Numerical Optimisation Project: Training Support Vector Machine using the Away-Step Frank-Wolfe algorithm

Stephane Ellis Department of Computer Science
University College London
London, UK
`ucabse3@ucl.ac.uk`

October 18, 2024

1 SVM classification and the underlying optimisation problem

1.1 Choose, describe the Classification Problem

The `make moons` from *sklearn* (Pedregosa et al. [2011]) is a synthetic dataset used for illustrating classification algorithms. It consists of two interleaving half circles (moons) and is often used to demonstrate the capability of classifiers to handle non-linear decision boundaries. The half circles represent two separate classes to which the data points belong. They are intertwined, making it a non-trivial classification problem. We split this dataset of 500 data points into 70/30 for the training and testing sets. Since this is a binary classifier, we assume that the training data is labeled $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where the data $\mathbf{x}_i \in \mathbb{R}^m$ have been sampled from two classes and the labels $y_i \in \{-1, +1\}$.

Training the classifier means estimating the parameters of a function $y : \mathbb{R}^m \to \{-1, +1\}$. We select an ansatz for a linear classifier $y(\mathbf{x}) = \operatorname{sign}(\mathbf{x}^T \mathbf{w} - \theta)$ with weight vector $\mathbf{w} \in \mathbb{R}^m$ and threshold value $\theta \in \mathbb{R}$. $y(\mathbf{x})$ is an SVM if we estimate its weights such that projection $\mathbf{x}_i^T \mathbf{w}$ of the training data from both classes are maximally separated (Bauckhage et al. [2022]). As there are various types of loss functions for max-min margin, SVMs come in different variations. We choose the L_2 SVMs dating back to Frieß and Harrison [1998].

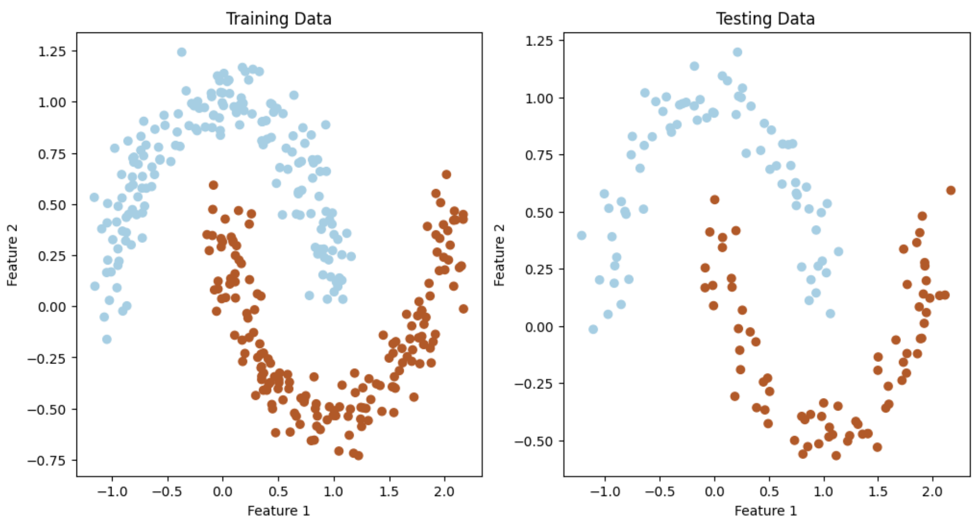

Figure 1: Visualisation of the classification problem

1.2 Choose the most appropriate SVM formulation for it (primal/dual, linear/nonlinear, choice of loss etc). Justify your choice for your classification problem.

The linear support vector machine for binary classification determines the max-min hyperplane between the training data for two given classes. If they are not linearly separable, slack variables are incorporated whose influence are controlled with a parameter $C \geq 0 \in \mathbb{R}$. In the case of an L_2 SVM, the slack variables are included in the primal objective in the form of a sum of squares. The primal problem of training an L_2 SVM comes with inequality constraints, which are different from least squares SVM. By evaluating the Karush-Kuhn-Tucker (KKT) conditions of optimality (see Bauckhage and Sifa [2021]), the *dual problem of training an L_2 SVM* can be derived:

$$\begin{aligned}
&\arg\min_{\boldsymbol{\mu}} \frac{1}{2}\boldsymbol{\mu}^T \left[\mathbf{X}^T\mathbf{X} \odot \mathbf{y}\mathbf{y}^T + \mathbf{y}\mathbf{y}^T + \frac{1}{C}\mathbf{I}\right] \boldsymbol{\mu} \\
&\text{s.t.} \quad \mathbf{1}^T\boldsymbol{\mu} = 1 \\
&\qquad\;\; \boldsymbol{\mu} \succeq \mathbf{0}
\end{aligned} \tag{1}$$

Here, $\mathbf{I}$ denotes the $n \times n$ identity matrix, $\mathbf{0}, \mathbf{1} \in \mathrm{R}^n$ are vectors of zeros and ones, $\boldsymbol{\mu} \in \mathrm{R}^{\mathrm{n}}$ is a vector of n Lagrange multipliers μ_i, and $\odot$ is the Hadamart product, or the element-wise product of matrices or vectors. Once the minimiser of (1) has been found, elements μ_s of $\boldsymbol{\mu}$, which are greater than zero are used to identify training data points that support the hyperplane, also known as *support vectors*. We then compute the weight vector $\mathbf{w} = \mathbf{X}\left[\mathbf{y} \odot \boldsymbol{\mu}\right]$ and threshold value $\theta = -\mathbf{1}^{\mathbf{T}}\left[\mathbf{y} \odot \boldsymbol{\mu}\right]$. Finally, the classifier becomes $y(\mathbf{x}) = \mathrm{sign}(\mathbf{x}^{\mathbf{T}}\mathbf{X}\left[\mathbf{y} \odot \boldsymbol{\mu}\right] + \mathbf{1}^{\mathbf{T}}\left[\mathbf{y} \odot \boldsymbol{\mu}\right]) = \mathrm{sign}(\left[\mathbf{x}^{\mathbf{T}}\mathbf{X} + \mathbf{1}^{\mathbf{T}}\right]\left[\mathbf{y} \odot \boldsymbol{\mu}\right])$.

In the training and application phases of an L_2 SVM, data vectors occur exclusively within inner products, namely the inner products $\mathbf{X}^{\mathbf{T}}\mathbf{X}$ in (1) and $\mathbf{x}^{\mathbf{T}}\mathbf{X}$ in the classifier $y(\mathbf{x})$. This property allows for invoking the Kernel trick and thus for treating non-linear settings. Consider a Mercer kernel $k : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, a non-linear classifier can be trained by replacing the Gram matrix $\mathbf{X}^{\mathbf{T}}\mathbf{X}$ with a kernel matrix $\mathbf{K}$ whose elements are given by $K_{ij} = k(x_i, x_j)$. We write the new expression for the classifier $y(\mathbf{x}) = \mathrm{sign}(\left[\mathbf{k}^{\mathbf{T}}(\mathbf{x}) + \mathbf{1}^{\mathbf{T}}\right]\left[\mathbf{y} \odot \boldsymbol{\mu}\right])$. The elements of the kernel vector $\mathbf{k}(\mathbf{x})$ are $k_i(\mathbf{x}) = k(\mathbf{x}, \mathbf{x_i})$.

The KKT conditions are met at the solution of any constrained optimization problem, whether convex or not, and regardless of the types of constraints. This is true if the intersection of the set of feasible directions with the set of descent directions matches the intersection of the set of feasible directions for linearized constraints with the set of descent directions (see Fletcher [1987], McCormick [1983]). This technical regularity condition applies to all support vector machines since their constraints are always linear. Moreover, the SVM problem is convex, featuring a convex objective function and constraints that define a convex feasible region. For convex problems, provided the regularity condition is satisfied, the KKT conditions are both necessary and sufficient for w, b and α to be a solution (Fletcher [1987]).

In the context of an L_2 SVM, data vectors are exclusively represented within inner products. This property enables the application of the Kernel trick, allowing for the treatment of non-linear settings. By introducing a Mercer kernel $k : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, a non-linear classifier can be trained by replacing the Gram matrix $\mathbf{X}^{\mathbf{T}}\mathbf{X}$ with a kernel matrix $\mathbf{K}$, where $K_{ij} = k(x_i, x_j)$. Mercer kernels, denoted as K, are defined by specific properties: continuity, symmetry, and positive semi-definiteness. These properties ensure that the optimization problem associated with SVMs is a concave maximization problem, facilitating efficient and reliable optimization. The classifier's expression is then updated accordingly. Notably, the Gaussian kernel, a specific type of Mercer kernel, is often employed in practice, defined as:

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

Here, σ controls the width of the bell-shaped curve.

In conclusion, solving the SVM problem is equivalent to finding the a solution to the KKT conditions. For soft margin SVM, in which we make use of slack variables, strong duality holds and the duality gap is zero. This means that the optimal values of the primal and dual forms are the same.

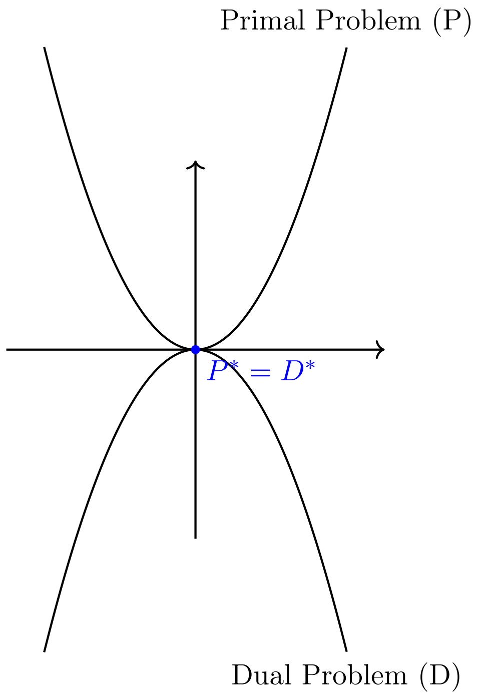

Figure 2: Diagram illustrating the case in which the strong duality holds. KKT conditions need to hold in order to have $(P^* - D^*) = 0$

2 Optimisation Method and Convergence Theory in the context of your problem:

2.1 Appropriate method for solution of your optimisation problem (justify with applicability, convergence type are rate, CPU time/memory efficiency.

Observing that the feasible set of the dual problem in (1) is the standard simplex $\Delta^{n-1} = \{\boldsymbol{\mu} \in \mathbb{R}^n \mid \boldsymbol{\mu} \succeq \mathbf{0} \wedge \mathbf{1}^T\boldsymbol{\mu} = 1\}$, which is a convex polytope, and letting $\mathbf{H} \equiv \mathbf{K} \odot \mathbf{y}\mathbf{y}^T + \mathbf{y}\mathbf{y}^T + \frac{1}{C}\mathbf{I}$ and the objective function $f(\boldsymbol{\mu}) \equiv -\frac{1}{2}\boldsymbol{\mu}^T\mathbf{H}\boldsymbol{\mu}$ for brevity, (1) can be written more succinctly as:

$$\arg\min_{\boldsymbol{\mu}\in\Delta^{n-1}} -f(\boldsymbol{\mu}) \tag{2}$$

This is now easily recognizable as a quadratic minimization problem over a compact convex set and therefore as a problem that can be solved using the Frank-Wolfe algorithm (Frank and Wolfe [1956]). Iterative versions of the Frank-Wolfe optimization are used as a baseline for L_2 SVM training.

For optimising over the product of probability simplices, project-gradient methods are a natural approach to take. They handle constraints by projecting the solution back onto the feasible set after each gradient step. The projection step, which requires optimizing a quadratic function over the constraint set, becomes computationally intractable as the number of variables increases.

Frank–Wolfe methods avoid projection by moving towards but not beyond an extreme point obtained via linear minimization, which ensures staying within the feasible region $\mathcal{X}$ (Braun et al. [2023]). In fact, most Frank–Wolfe algorithms add at most one extremal point per iteration to the representation. The condition is that the feasible region $\mathcal{X}$ must admit fast linear optimization, which is when FW is used. The number of linear minimizations is comparable to the number of more costly projections.

When the optimal solution lies at the boundary of polytope P, the convergence rate of iterates is sublinear: $f(x_t) - f(x^*) \leq (1/t)$ with x^* being an optimal solution. The iterates zig-zag between the vertices, which define the face containing the solution x^*.

The Away-Steps Frank Wolfe algorithm (AFW) offers significant improvements in convergence rates over the original Frank-Wolfe algorithm. For strongly convex functions, the AFW achieves linear convergence by removing less beneficial vertices of the polytope $\mathcal{X}$ from the active set. It focuses on the more important directions and avoids *zigzagging*. The AFW therefore introduces steps that move away from vertices in a given convex combination of x_t, as opposed to steps that move towards vertices of P. The away steps remove *weight* in the convex combination from undesirable vertices.

The AFW algorithm benefits from the geometric properties of the feasible region, such as the pyramidal width, which helps in deriving tight upper bounds on the primal gap. This ensures that the optimization progresses efficiently even in complex polytope structures.

While the basic FW requires storage for the current iterate x_t, gradient $\Delta f(x_t)$, chosen vertex v_t from polytope P, with each being a vector in $\mathbb{R}$, the AFW algorithm also requires keeping track of the active set of vertices and the away vertex v_t^A. The active set contains the vertices used to represent the current iterate x_t as a convex combination.

2.2 Describe the method in sufficient detail so it is clear how it works (state your sources, unify notation). You may want to introduce a method which was not on the syllabus and apply it to your optimisation problem. The bonus points awarded will depend on the level of difficulty of the chosen method and the theory involved.

We consider the general constrained convex optimization problem of the form:

$$\min_{\mathbf{x}\in\mathcal{M}} f(x), \qquad \mathcal{M} = \operatorname{conv}(\mathcal{A}) \qquad \text{with only access to:} \quad \operatorname{LMO}_{\mathcal{A}}(\mathbf{c}) \in \arg\min_{\mathbf{x}\in\mathcal{A}} \langle \mathbf{c}, \mathbf{x} \rangle \tag{3}$$

where $\mathcal{A} \subseteq \mathrm{R}^{\mathrm{d}}$ is a *finite* set of vectors called *atoms*, which do not have to be vertices (extreme points) of $\mathcal{M}$. The assumption is that the function f is $\mu -$ strongly convex with L-Lipschitz continuous gradient over $\mathcal{M}$.

Lemma 1.7 (Equivalence of smoothness and Lipschitz-continuous gradients). Let $f : \mathcal{X} \to \mathbb{R}$ be a differentiable convex function on a full dimensional convex domain $\mathcal{X}$. Then f is L-smooth if and only if its gradient ∇f is L-Lipschitz continuous.

Since the set of vectors $\mathcal{A}$ is finite, $\mathcal{M}$ is a convex and bounded polytope.

Original Frank Wolfe The Frank-Wolfe (FW) algorithms only require optimising *linear* functions, or first-order approximations of the objective function f (Frank and Wolfe [1956]) at x_t, given by $f(x_t) + \langle \Delta f(x_t), x - x_t \rangle$. At each iteration, the linear optimisation (LMO) yields an extreme point v_t, which forms the *descent direction* $v_t - x_t$ together with the current iterate x_t. This is an alternative to the negative gradient direction in the Euclidean norm (Braun et al. [2023]). $\mathcal{M}$ is only accessed through the linear minimization oracle. The next iterate x_{t+1} is found by moving in this direction by performing a line-search on f between x_t and v_t.

The first oracle, called the *First-Order Oracle* (FOO) gives information about the function f when queried with a point $x \in \mathcal{X}$. It returns the function value $f(x)$ and the gradient $\Delta f(x)$ of f at x. The objective function is given by:

$$\mathcal{D}(\boldsymbol{\mu}) = \frac{1}{2}\boldsymbol{\mu}^T \left[\mathbf{K} \odot \mathbf{y}\mathbf{y}^T + \mathbf{y}\mathbf{y}^T + \frac{1}{C}\mathbf{I}\right] \boldsymbol{\mu}$$

The gradient of objective function is given by:

$$\Delta(-\mathcal{D}(\boldsymbol{\mu})) = -\Delta\mathcal{D}(\boldsymbol{\mu}) = \left[\mathbf{K} \odot \mathbf{y}\mathbf{y}^T + \mathbf{y}\mathbf{y}^T + \frac{1}{C}\mathbf{I}\right] \boldsymbol{\mu}$$

where $\boldsymbol{\mu}$ will be used interchangeably with x as we will iterate over the Lagrange multipliers.

The second oracle is the *Linear Minimization Oracle* (LMO) gives information about the domain $\mathcal{M}$. When queried with a linear function c, the LMO returns an extreme point $v \in \mathcal{X}$ minimising c, i.e. $v = \text{argmin}_{x \in \mathcal{X}} \langle c, x \rangle$.

It is important to highlight the sparsity of the iterates. In iteration t of the algorithm, the iterate can be represented as a sparse convex combination of at most $t+1$ atoms $\mathcal{S}^{(t)} \subseteq \mathcal{A}$ of domain $\mathcal{M}$, which we write as $\mathbf{x}^{(t)} = \sum_{\mathbf{v} \in \mathcal{S}^{(t)}} \alpha_{\mathbf{v}}^{(t)} \mathbf{v}$. $\mathcal{S}^{(t)}$ represents the active set, containing the previously discovered atoms s_c for $c < t$ that have non-zero weight $\alpha_{s_c}^{(t)} > 0$ in the expansion. Tracking the active set is not necessary in the original FW algorithm, but it is in the Away-Steps Frank-Wolfe (AFW) algorithm, which maintains $\mathcal{S}^{(t)}$.

Away-Steps Frank Wolfe It is well-known that the Frank Wolfe algorithm tends to stagnate near the solution a^* (Guélat and Marcotte [1986]) due to the so called *zigzagging* phenomenon. When the optimum lies well inside a face, at the boundary of $\mathcal{M}$, no vertices are available in the approximate direction of the optimum as the algorithm approaches the face. The FW algorithm has no choice but move in progressively worse directions. The convergence rate of the iterates is sublinear: $f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) \leq O(1/t)$ with x^* being an optimal solution.

Algorithm 1 Away-steps Frank-Wolfe algorithm: AFW($x^{(0)}$; A; ϵ)

```
1: Let $x^{(0)} \in A$, and $S^{(0)} := \{x^{(0)}\}$ (so that $\alpha_v^{(0)} = 1$ for $v = x^{(0)}$ and 0 otherwise)
2: for $t = 0 \dots T$ do
3:     Let $s_t := \text{LMO}_A(-\nabla f(x^{(t)}))$ and $d_t^{\text{FW}} := s_t - x^{(t)}$ (the FW direction)
4:     Let $v_t \in \arg\max_{v \in S^{(t)}} \langle -\nabla f(x^{(t)}), v \rangle$ and $d_t^{\text{A}} := x^{(t)} - v_t$ (the away direction)
5:     if $g_t^{\text{FW}} := \langle -\nabla f(x^{(t)}), d_t^{\text{FW}} \rangle \leq \epsilon$ then return $x^{(t)}$ (FW gap is small enough, so return)
6:     end if
7:     if $\langle -\nabla f(x^{(t)}), d_t^{\text{FW}} \rangle \geq \langle -\nabla f(x^{(t)}), d_t^{\text{A}} \rangle$ then
8:         $d_t := d_t^{\text{FW}}$, and $\gamma_{\max} := 1$ (choose the FW direction)
9:     else
10:        $d_t := d_t^{\text{A}}$, and $\gamma_{\max} := \frac{\alpha_{v_t}}{1 - \alpha_{v_t}}$ (choose away direction; maximum feasible step-size)
11:    end if
12:    Line-search: $\gamma_t \in \arg\min_{\gamma \in [0, \gamma_{\max}]} f(x^{(t)} + \gamma d_t)$
13:    Update $x^{(t+1)} := x^{(t)} + \gamma_t d_t$ (and accordingly for the weights $\alpha^{(t+1)}$, see text)
14:    Update $S^{(t+1)} := \{v \in A : \alpha_v^{(t+1)} > 0\}$
15: end for
```

The *away* direction d_t^A is defined in Algorithm 1 (line 4) by finding the atom v_t in $\mathcal{S}^{(t)}$ that maximises the potential of descent given by $g_t^A := \langle -\nabla f(x^{(t)}), \mathbf{x^{(t)}} - \mathbf{v_t} \rangle$. This search is over the small active set $\mathcal{S}^{(t)}$. The maximum step-size $\gamma_{\max}$ ensures that the new iterate $x^{(t+1)} := x^{(t)} + \gamma_t d_t^A$ remains in $\mathcal{M}$. This guarantees that the convex representation is maintained. Using the conservative maximum step-size of line 10 ensures we do not need the oracle computing the true maximum feasible step-size, requiring to know when we cross the boundary of $\mathcal{M}$ along a chosen line.

The FW gap g_t^{FW} is an upper bound on the unknown suboptimality and can be used as a stopping criterion:

$$g_t^{\mathrm{FW}} := \langle -\nabla f(x^{(t)}), d_t^{\mathrm{FW}} \rangle \geq \langle -\nabla f(x^{(t)}), x^\star - x^{(t)} \rangle \geq f(x^{(t)}) - f(x^\star) \quad \text{(by convexity)}$$

If $\gamma_t = \gamma_{\max}$, then we call this step a drop step, as it fully removes the atom v_t from the currently active set of atoms $S^{(t)}$ (by setting its weight to zero).

2.3 Discuss if local or global convergence can be expected for your problem

We assume the objective function f is smooth over a compact set $\mathcal{M}$. This means its gradient is Lipschitz continuous with constant L. Let $M := \operatorname{diam}(\mathcal{M})$. Let $\mathbf{d}_t$ be the direction in which line-search is executed in line 12 of Algorithm 1. Using the following lemma from Nesterov [2004]:

Lemma 2.3.1 (Descent Lemma). Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable with a Lipschitz gradient, i.e., $\exists L > 0$ such that

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|, \quad \forall x, y \in \mathbb{R}^n.$$

Then,

$$f(y) \leq f(x) + \nabla f(x)^T (y - x) + \frac{1}{2} L\|x - y\|^2, \quad \forall x, y \in \mathbb{R}^n.$$

We have:

$$f(\mathbf{x}^{(t+1)}) \leq f(\mathbf{x}^{(t)} + \gamma \mathbf{d_t}) \leq f(\mathbf{x}^{(t)}) + \gamma \langle \nabla f(\mathbf{x}^{(t)}), \mathbf{d}_t \rangle + \frac{\gamma^2}{2} L\|\mathbf{d_t}\|^2 \quad \forall \gamma \in [0, \gamma_{\max}] \tag{4}$$

We let $\mathbf{c}_t := -\nabla f(\mathbf{x}(t))$ and let $h_t := f(\mathbf{x}^{(t)} - f(\mathbf{x}^*$ be the suboptimality error. Suppose that $\gamma_{\max \geq \langle \mathbf{c}_t, \mathbf{d}_t \rangle / (L\|\mathbf{d}_t\|^2)}$, set $\gamma = \gamma_t^*$ to minimise the RHS of (4). By re-organising, we get a lower bound on the progress:

$$h_t - h_{t+1} \geq \frac{\langle \mathbf{r}_t, \mathbf{d}_t \rangle^2}{2L\|\mathbf{d}_t} \|^2 = \frac{1}{2L} \langle \mathbf{r}_t, \hat{\mathbf{d}}_t \rangle^2 \tag{5}$$

where $\hat{\mathbf{d}}_t := \mathbf{d}_t / \|\mathbf{d}_t\|$ is the normalised vector. Let $\boldsymbol{\epsilon}_t := \mathbf{x}^* - \mathbf{x}^{(t)}$ be the error vector. By $\mu -$ strong convexity of f, we have:

$$f(x(t) + \mathbf{e}_t) \geq f(x(t)) + \mathbb{E}\left[Drf(x(t)); \mathbf{e}_t\right] + \frac{1}{2} \mu \|\mathbf{e}_t\|^2, \quad \forall \mathbf{e}_t \in [0, 1] \tag{6}$$

Sure, here is the paraphrased version of the text you provided along with the LaTeX code to render it:

The right-hand side (RHS) of the inequality can be minimized with respect to (without constraints) by setting $\gamma := \frac{\langle r_t, e_t \rangle}{\mu \|e_t\|^2}$. We can choose any value on the left-hand side (LHS) to maintain a valid bound, and by setting $\gamma = 1$, we derive $f(x^*)$. Rearranging this gives:

$$h_t \leq \frac{\langle r_t, \hat{e}_t \rangle^2}{2\mu}$$

Combining this with equation (5), we get:

$$h_t - h_{t+1} \geq \frac{\mu}{L} \frac{\langle r_t, \hat{d}_t \rangle^2}{\langle r_t, \hat{e}_t \rangle^2} h_t$$

This inequality is quite general and applies to any line-search method in direction d_t. To achieve linear convergence, we need to ensure the term in front of h_t on the RHS is positively bounded. Assuming the solution x^* lies within the relative interior of M, with a distance of at least $\delta > 0$ from the boundary, we have $\langle r_t, d_t \rangle \geq \delta \|r_t\|$ for the Frank-Wolfe direction d_{FW_t}. Combining this with $\|d_t\| \leq M$, we get a linear convergence rate with a constant $1 - \frac{\mu}{L} \left(\frac{\delta}{M} \right)^2$. However, if x^* is on the boundary, $\langle \hat{r}_t, \hat{d}_t \rangle$ approaches zero, leading to sublinear convergence due to the zig-zagging phenomenon. (Lacoste-Julien and Jaggi [2015])

AFW has *global linear convergence.* The crucial insight for proving the global linear convergence of the Away-Step Frank-Wolfe (AFW) algorithm involves relating $\langle c_t, d_t \rangle$ to the pairwise Frank-Wolfe direction $d_t^{\text{PFW}} := s_t - v_t$. Based on the direction selection in lines 6 to 10 of Algorithm 1, we have:

$$2\langle c_t, d_t \rangle \geq \langle c_t, d_t^{\text{FW}} \rangle + \langle c_t, d_t^{\text{A}} \rangle = \langle c_t, d_t^{\text{FW}} + d_t^{\text{A}} \rangle = \langle c_t, d_t^{\text{PFW}} \rangle$$

Thus, $\langle c_t, d_t \rangle \geq \frac{1}{2} \langle c_t, d_t^{\text{PFW}} \rangle$. The key property of the pairwise FW direction is that $\langle c_t, d_t^{\text{PFW}} \rangle$ is bounded away from zero by a quantity dependent only on the geometry of $\mathcal{M}$ (unless at the optimum), termed the pyramidal width of $\mathcal{A}$. (Lacoste-Julien and Jaggi [2015])

For steps where $\gamma_t^* \leq \gamma_{\max}$, this underpins the linear convergence of AFW. If $\gamma_{\max}$ is too small, AFW performs a drop step, reducing the active set size by one and ensuring these steps are infrequent (no more than half the time) (Lacoste-Julien and Jaggi [2015]). This forms the basis of the global linear convergence proof for AFW. The remaining part addresses boundary cases, and similar techniques apply to pairwise FW, albeit with potentially problematic swap steps (Lacoste-Julien and Jaggi [2015]).

2.4 Discuss theoretical local convergence rates predicted for your problem

The best rate for the vanilla FW and AFW algorithms is *linear convergence*: $(\log(1/\epsilon))$. The special cases are: problems in which the optimum lies in the interior of the feasible region (see Guélat and Marcotte [1986]; Beck and Teboulle [2004]) or when the feasible region is uniformly or strongly convex (see Levitin and Polyak [1966]).

Whenever the optimal solution x^* is contained in the interior of $\mathcal{X}$, denoted by $\text{Int}(\mathcal{X})$, the vanilla FW algorithm (employing line search or short step rule), converges linearly. This is the case when the assumption that the optimum solution set Ω^* is contained in the relative interior of a face, is violated, assumption is also violated if vertex of $\mathcal{P}$ is optimal solution (we do not assume polytope domain here or unique optimal solution).

The *drop step* is when the active set shrinks $|S^{t+1}| < |S^t|$ and an atom is dropped.

Theorem 1. *Suppose that f has an L-Lipschitz gradient and is μ-strongly convex over $M = conv(A)$. Let $M = diam(M)$ and $\delta = PWidth(A)$ as defined by (9). Then the suboptimality h_t of the iterates of the Away-Step Frank-Wolfe (AFW) algorithm decreases geometrically at each step that is not a drop step nor a swap step (i.e., when $\gamma_t < \gamma_{max}$, called a 'good step'), that is*

$$h_{t+1} \leq (1 - \rho) h_t, \quad \textit{where} \quad \rho := \frac{\mu}{4L} \left(\frac{\delta}{M} \right)^2.$$

Let $k(t)$ be the number of 'good steps' up to iteration t. For AFW, we have $k(t) \geq t/2$. This yields a global linear convergence rate of

$$h_t \leq h_0 \exp(-\rho k(t)).$$

If $\mu = 0$ (general convex), then $h_t = O(1/k(t))$ instead. See Theorem 8 in Appendix D for an affine invariant version and proof.

Note that none of the existing linear convergence results showed that the duality gap was also linearly convergent. The result for the gap follows directly from the simple manipulation of (2); putting the FW gap to the LHS and optimizing the RHS for $\gamma \in [0, 1]$.

Theorem 2. *Suppose that f has an L-Lipschitz gradient over M with $M := diam(M)$. Then the FW gap g_t^{FW} for AFW is upper bounded by the primal error h_t as follows:*

$$g_t^{FW} \leq h_t + \frac{LM^2}{2} \quad \textit{when} \quad h_t > \frac{LM^2}{2}, \quad g_t^{FW} \leq M\sqrt{2h_tL} \quad \textit{otherwise.}$$

For AFW, we require that ∇f is L-Lipschitz over the larger domain $\mathcal{M} + \mathcal{M} - \mathcal{M}$.

The Away-Step Frank-Wolfe (AFW) algorithm achieves linear convergence under certain conditions, particularly when the optimal solution lies in the interior of the feasible region or when the feasible region is uniformly or strongly convex. The algorithm benefits from geometrically decreasing suboptimality at each step that is not a drop step or a swap step. The theoretical analysis confirms that at least half of the iterations result in a significant reduction in the primal error, ensuring a global linear convergence rate. Additionally, the duality gap, which is crucial for measuring convergence, is also shown to be upper bounded by the primal error, further reinforcing the efficiency of the AFW method. This comprehensive understanding highlights the robustness of the AFW algorithm in achieving rapid convergence in optimization problems with strongly convex and Lipschitz gradient conditions.

3 Solution and discussion of the results:

3.1 Solve + state relevant parameter choices. Discuss obtained solution

The parameters chosen include the standard deviation parameter for the Gaussian kernel, set to $\sigma = 2$, the regularization parameter set to $C = 100$, the maximum number of iterations $t_{max} = 10000$ and the convergence threshold set to $\epsilon = 0.01$.

The obtained solution from the AFW algorithm is the vector of dual variables, or Lagrange multipliers, given by: $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, with each α_i satisfying $0 \leq \alpha_i \leq C$. From the dual variables, we can compute the primal parameters, i.e., the set of weights $\mathbf{w}$ and the threshold b.

3.2 Provide relevant convergence plots

3.3 Discuss theoretical vs. empirical convergence rates with checks (e.g. active trust region, was step size 1 attained etc.)

In Figure 3, the beginning of the convergence seems to be sublinear, but overall, this plot of the log of the gap vs. iterates confirms that the convergence is indeed linear for the optimiser on the `make_moons` dataset. The

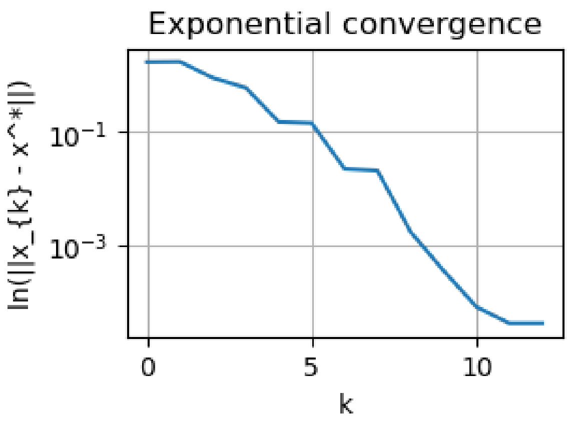

Figure 3: Exponential Convergence Plot

empirical convergence therefore aligns with the theory that the AFW algorithm converges linearly globally. It can be noted that finding the decision boundary for a dataset containing only two features will converge faster than for a more realistic dataset containing a large number of features.

3.4 Performance of method in terms of complexity, CPU time, memory used

The recorded memory usage of the AFW algorithm is approximately 6.48 MB. This includes the memory allocated for storing variables, intermediate results, and other necessary data structures. This is found using the `memory profiler` module (mem). The CPU time used is 1.48 seconds using the tim package.

3.5 Classification result

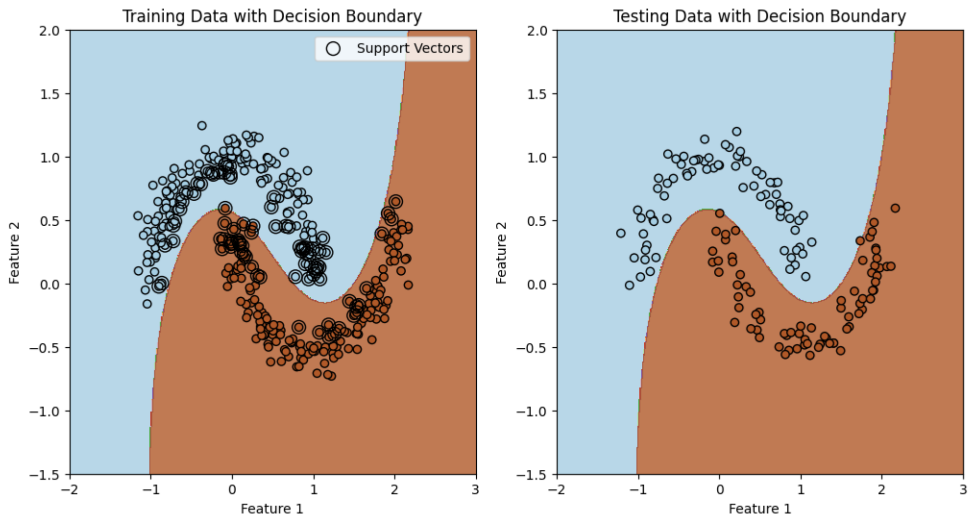

Figure 4: Visualisation of the decision boundary and the support vectors

The performance of the trained SVM model is evaluated on the test dataset. Precision for class -1 is 0.99 and 1.00 for class 1. The overall accuracy is 0.99.

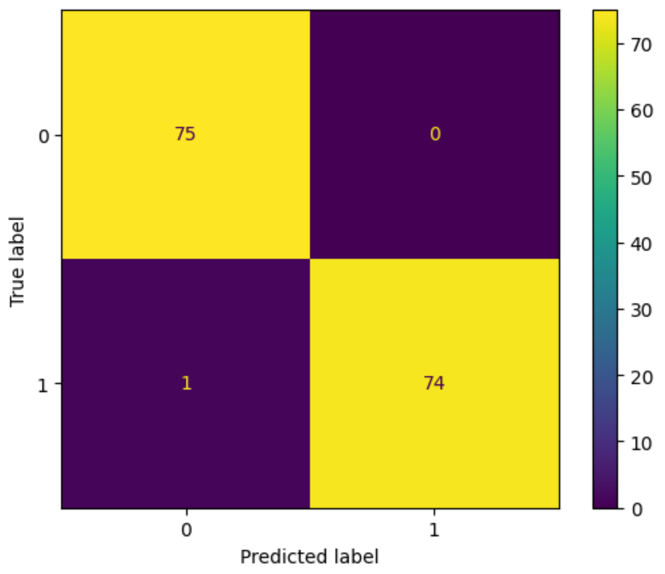

Figure 5: Confusion matrix for the classification result

References

memory_profiler: A module for monitoring memory usage of a python program. `https://pypi.org/project/memory-profiler/`. Accessed: 2024-06-05.

Python time module. `https://docs.python.org/3/library/time.html`. Accessed: 2024-06-05.

Christian Bauckhage and Rafet Sifa. Ml2r theory nuggets: The dual problem of l2 support vector machine training. *ML and AI Lab, Computer Science, University of Bonn*, June 2021.

Christian Bauckhage, Helen Schneider, Benjamin Wulff, and Rafet Sifa. Gradient flows for l2 support vector machine training, 08 2022.

Amir Beck and Marc Teboulle. A conditional gradient method with linear rate of convergence for solving convex linear systems. *Mathematical Methods of Operations Research*, 59(2):235–247, 2004. doi: 10.1007/s001860400350.

Gábor Braun, Alejandro Carderera, Cyrille W. Combettes, Hamed Hassani, Amin Karbasi, Aryan Mokhtari, and Sebastian Pokutta. Conditional gradient methods, 2023.

Roger Fletcher. *Practical Methods of Optimization.* Wiley, 2nd edition, 1987. ISBN 978-0471915478.

M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1–2): 95–110, 1956. doi: 10.1002/nav.3800030109.

T.T. Frieß and R.F. Harrison. The kernel adatron with bias unit: Analysis of the algorithm (part 1). Technical Report ACSE Research Report 729, Dept. of Automatic Control and Systems Engineering, University of Sheffield, 1998.

Jacques Guélat and Patrice Marcotte. Some comments on wolfe's "away step". *Mathematical Programming*, 35:110–119, 1986.

Simon Lacoste-Julien and Martin Jaggi. On the global linear convergence of frank-wolfe optimization variants. *Advances in Neural Information Processing Systems*, 28:496–504, 2015. URL `https://papers.nips.cc/paper/5860-on-the-global-linear-convergence-of-frank-wolfe-optimization-variants`.

E. S. Levitin and B. T. Polyak. Constrained minimization methods. *USSR Computational Mathematics and Mathematical Physics*, 6(5):1–50, 1966. doi: 10.1016/0041-5553(66)90055-8.

Garth P. McCormick. *Nonlinear Programming: Theory, Algorithms, and Applications*. John Wiley Sons, 1983. ISBN 978-0471876250.

Yurii Nesterov. *Introductory Lectures on Convex Optimization*. Kluwer Academic Publishers, 2004.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. Available online, 2011. URL `https://scikit-learn.org/stable/`. Version 0.24.2.

A Appendix

```

source: ML2R Coding Nuggets
SVM Training Using 16 Lines of Plain Vanilla NumPy Code by Christian Bauckhage

    def squaredEDM(matX, matY):
    return spt.distance.cdist(matX.T, matY.T , 'sqeuclidean')

def computeGaussianKernelMatrix(matX, sigma =1.):
    return np.exp( -0.5 * squaredEDM(matX, matX) / sigma **2)

def computeGaussianKernelVector(vecX, matX, sigma =1.):
    if vecX.ndim == 1: vecX = np.reshape(vecX, (-1, 1))
    return np.exp( -0.5 * squaredEDM(vecX , matX )/ sigma **2)

    def fwL2SVM(matH, tmax=1000):
    m, n = matH.shape
    matI = np.eye(n)
    vecM = np.ones(n) / n
    for t in range(tmax):
        indx = np.argmin(matH @ vecM)
        vecM += 2 / (t + 2) * (matI[indx] - vecM)
    return vecM

def fwL2SVM_V2(matH, tmax=1000):
    _, n = matH.shape
    vecM = np.ones(n) / n
    for t in range(tmax):
        indx = np.argmin(matH @ vecM)
        vecM -= 2 / (t + 2) * vecM
        vecM[indx] += 2 / (t + 2)

    return vecM

    def trainL2SVM(matX, vecY, C =100., tmax =10000):
    matK = computeGaussianKernelMatrix(matX, 2.0) # transformed train data into other
        space

    matY = np.outer(vecY, vecY)
    matG = matK * matY
    matH = matG + matY + np.eye(matX.shape[1]) / C

    return fwL2SVM(matH, tmax)

    def runL2SVM(matXtst, matXs, vecYs, vecMs):
    bias = -vecYs @ vecMs
    vecK = computeGaussianKernelVector(matXtst, matXs, 2.0)
    print(np.sign((vecK * vecYs) @ vecMs - bias))
    return np.sign((vecK * vecYs) @ vecMs - bias)

    from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X, y = make_moons(n_samples = 500, noise=0.1, random_state=42)

matX, matX_test, vecY, vecY_test = train_test_split(X, y, test_size=0.3, random_state=42)

# visualise training dataset
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(matX[:, 0], matX[:, 1], c=vecY, cmap=plt.cm.Paired)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Training Data')

# Visualize the testing dataset
plt.subplot(1, 2, 2)
plt.scatter(matX_test[:, 0], matX_test[:, 1], c=vecY_test, cmap=plt.cm.Paired)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Testing Data')

plt.show()

print('Training Data:')
print('matX_train:', matX)
print('vecY_train:', vecY)

print('\nTesting Data:')
print('matX_test:', matX_test)
print('vecY_test:', vecY_test)
```

```python

# the code assumes we are using Transposed train matrices
import psutil
from time import process_time_ns
# Start recording time
start_time = process_time_ns()

# Record initial CPU and memory usage
process = psutil.Process()
initial_cpu_time = process.cpu_times()
initial_memory_info = process.memory_info()

vecM = trainL2SVM(matX.T, vecY.T)

# Calculate elapsed time
finish_time = (process_time_ns() - start_time) / 1e9

# Record final CPU and memory usage
final_cpu_time = process.cpu_times()
final_memory_info = process.memory_info()


# Calculate CPU and memory usage
cpu_time_used = (final_cpu_time.user + final_cpu_time.system) - (initial_cpu_time.user +
    initial_cpu_time.system)
memory_used = final_memory_info.rss - initial_memory_info.rss

supps = np.where(vecM > 0, True, False) # support vector
matXs = matX.T[:, supps]
vecYs = vecY.T[supps]
vecMs = vecM.T[supps]

vecYtst = runL2SVM(matX_test.T, matXs, vecYs, vecMs)
# Plot decision boundary
xx, yy = np.meshgrid(np.linspace(-2, 3, 500), np.linspace(-1.5, 2, 500))
grid = np.c_[xx.ravel(), yy.ravel()]
grid_predictions = runL2SVM(grid.T, matXs, vecYs, vecMs).reshape(xx.shape)

plt.figure(figsize=(12, 6))

# Plot training data with decision boundary
plt.subplot(1, 2, 1)
plt.contourf(xx, yy, grid_predictions, alpha=0.8, cmap=plt.cm.Paired)
plt.scatter(matX[:, 0], matX[:, 1], c=vecY, cmap=plt.cm.Paired, edgecolors='k')
plt.scatter(matXs[0, :], matXs[1, :], edgecolors='k', facecolors='none', s=100, label='
    Support Vectors')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Training Data with Decision Boundary')
plt.legend()

# Plot testing data with decision boundary
```

```
plt.subplot(1, 2, 2)
plt.contourf(xx, yy, grid_predictions, alpha=0.8, cmap=plt.cm.Paired)
plt.scatter(matX_test[:, 0], matX_test[:, 1], c=vecY_test, cmap=plt.cm.Paired, edgecolors=
    'k')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Testing Data with Decision Boundary')

plt.show()

print('Training Data:')
print('matX_train:', matX)
print('vecY_train:', vecY)

print('\nTesting Data:')
print('matX_test:', matX_test)
print('vecY_test:', vecY_test)

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from time import process_time_ns

source: https://github.com/santiviquez/frank-wolfe-svm

def loss(X, y, a, C):
    """
    Evaluates the dual lagrangian at a particular value of a (alpha)

    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        a (np.array): array of the lagrange multipliers (alpha)
        C (float): Regularization parameter. The strength of the regularization
        is inversely proportional to C. Must be strictly positive.
        The penalty is a squared l2 penalty.

    Returns:
        loss (float): value of dual lagrangian
    """
    N = y.shape[0]
    aa = a.reshape(N,1)
    yy = y.reshape(N,1)
    term_1 = (np.matmul(yy,yy.T)*np.matmul(aa,aa.T)*np.matmul(X,X.T)).sum()
    term_2 = np.sum(a * a)
    loss = (0.5) * term_1 + (1 / (2 * C)) * term_2
    #loss = int(loss)
    return loss

    def grad(X, y, a, C):
    """
    Computes the gradient of the of the dual loss function

    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        a (np.array): array of the lagrange multipliers (alpha)
        C (float): Regularization parameter. The strength of the regularization
        is inversely proportional to C. Must be strictly positive.
        The penalty is a squared l2 penalty.

    Returns:
        gradient (np.array): gradient of the dual lagrangian
    """
    gradients = y * np.matmul(a*y,np.matmul(X,X.T)).T + (a/C)
    #print(gradients)
    #print(a)
    return gradients

    # attempt to define the Hessian of the dual function
def hessian(X, y, a,  C): # no Lagrangian multipliers as inputs
    """
    Computes the Hessian of the dual loss function

    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        a (np.array): array of the lagrange multipliers (alpha)
        C (float): Regularization parameter. The strength of the regularization
        is inversely proportional to C. Must be strictly positive.
        The penalty is a squared l2 penalty.

    Returns:
        hessian (np.array): Hessian of the dual lagrangian
    """
    N = len(y)
    XTX = np.matmul(X, X.T)
    hessian = np.outer(y, y) * XTX + np.identity(N) / C
    return hessian

    def fw_oracle(X, y, a, C):
    """
    Computes the Frank-Wolfe oracle defined as argmin(s) <s, grad(f(x))>
    where s is a feasible solution that belongs to C.


    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        a (np.array): array of the lagrange multipliers (alpha)
        C (int): positive integer, SVM budget parameter, vertix of the FW set.
    Returns:
        s (np.array): s that minimizes argmin(s) <s, grad(f(x))>
    """
    #s = np.zeros(len(y))
    #gradient = grad(X, y, a, C)
    #abs_gradient_ = abs(gradient)
    #max_idx = np.nonzero(abs_gradient_ == max(abs_gradient_))[0][0]
    #s[max_idx] = - np.sign(gradient[max_idx]) #* C
    #return s

    s = np.zeros(len(y))
    gradient = grad(X, y, a, C)
    i = np.argmin(gradient)
    s[i] = 1

    #print('grad',gradient)
    return s


def fw(X, y, k, C):
    """
    Performs k Frank-Wolfe updates

    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        k (int): number of iterations
        C (int): positive integer, SVM budget parameter, vertix of the FW set.

    Returns:
        a (np.array): lagrange multipliers that minimazes the dual lagrangian
        history (dictionary): history of the training loss
    """
    history = {}
    train_loss = 0
    iterates = {'xs': np.zeros((len(X), k+2)),
        'alphas': np.zeros(k+2)}

    a = np.ones(len(y))/len(y)

    iterates['xs'][:, 0] = a


    for k in range(0, k):
        gamma = 2 / (k + 2)
        s = fw_oracle(X, y, a, C)
        a = (1 - gamma) * a + gamma * s

        if k % 1 == 0:
            train_loss = loss(X, y, a, C)
            history[k] = train_loss

        iterates['xs'][:, k+1] = a
        iterates['alphas'][k+1] = gamma

    return a, history, iterates

def away_fw_oracle(X, y, a, S, C):
    """
    Computes the Frank-Wolfe oracle defined as argmax(s) <s, grad(f(x))>
    where s is a feasible solution.

    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        a (np.array): array of the lagrange multipliers (alpha)
        C (int): positive integer, SVM budget parameter, vertix of the FW set.
    Returns:
        s (np.array): s that minimizes argmax(s) <s, grad(f(x))>
    """
    gradient = grad(X, y, a, C)
    dot = []
    for alpha in S:
        # dot.append(np.dot(alpha.reshape(len(alpha), 1), gradient))
        dot.append(np.dot(alpha, gradient))
        #dot.append(np.dot(alpha.reshape(1, len(alpha)), gradient.reshape(len(gradient))))

    idx = np.argmax(dot)
    v = S[idx]
    return v

# 'history' contains loss
def away_fw(X, y, k, C, epsilon):
    # store loss history
    history = {}
    train_loss = 0

    iterates = {
        'xs': np.zeros((len(X), k+2)),
        'alphas': np.zeros(k+2)}

    # initialise Lagrangian multipliers 'a' with equal values summing to 1
    a = np.ones(len(y))/len(y)
    # initialise set 'S' with initial multipliers 'a'
    S = [a]

    iterates['xs'][:, 0] = a

    # perform loop for 'k' iterations
    for k in range(0, k):
        # store training loss for each iteration
        if k % 1 == 0:  # k % 5
            train_loss = loss(X, y, a, C)
            history[k] = train_loss

        s = fw_oracle(X, y, a, C) # Frank Wolfe direction 's'
        v = away_fw_oracle(X, y, a, S, C) # Away steps Frank Wolfe direction 'v'

        # determine alpha based on whether 'v' is same as initial 'a'
        if (v == S[0]).all():
            alpha = 1
        else:
            alpha = 0

        # compute FW-direction
        d_fw = s - a
        # compute Away-direction
        d_a = a - v

        # compute gradient in the FW direction
        g_fw = np.dot(-grad(X, y, a, C), d_fw)

        # check for convergence
        if g_fw <= epsilon:
            return a, history

        # determine direction 'd' and maximum step size 'gamma_max'
        if g_fw >= np.dot(-grad(X, y, a, C), d_a):
            d = d_fw
            gamma_max = 1

        else:
            d = d_a
            gamma_max = alpha / (1 - alpha + 0.0001)

        # compute current loss
        loss_ = loss(X, y, a, C)

        # Search for the optimal step size 'gamma' along the direction 'd'
        gamma_search = np.linspace(0, gamma_max, 25)

        for gamma_iter in gamma_search:
            tmp = loss(X, y, a + gamma_iter * d, C)
            # if a lower loss is found, update the loss and gamma
            if tmp < loss_:
                loss_ = tmp
                gamma = gamma_iter
            else:
                gamma = 0

        # update 'a' using the optimal step size 'gamma
        a = a + gamma * d

        iterates['xs'][:, k+1] = a
        iterates['alphas'][k+1] = gamma

        # update set S based on whether 'gamma' is 1
        if gamma == 1:
            S=[s]
        else:

            # add 's' to set 'S' if it's not already in it
            check=True
            for s_check in S:
                if (s != s_check).all():
                    check=False
                    break
            if check:
                S.append(s)

        #for away step
        #The idea is that in each iteration, we not only add a new atom s,
        #but potentially also remove an old atom (provided it is bad with respect to our
           objective).
        if gamma == gamma_max:

            eq=False
            for s_check in S:
                if (v == s_check).all():
                    S=S.remove(v)
            alpha =(1+gamma)*alpha
        else:
          alpha=(1+gamma)*alpha+gamma

    return a, history, iterates

def svm(X, y, k=10, C=1, type="hard", method="fw",epsilon=0.01):
    """
    Recovers the primal of the dual SVM loss function formulation.
    Learns the w and b of the separating hyperplane by using Frank-Wolfe
    and Frank-Wolfe variations

    Args:
        X (np.matrix): data matrix
        y (np.array): array of 1s, -1s representing the classification labels
        C (int): positive integer, SVM budget parameter, vertix of the FW set.
        type (string): can be hard or soft depending of the desiared margin
        method (string): optimization algorithm. Can be fw, away_fw,
        pairwise_fw, fully_corrective_fw
        k (int): positive integer, number of iteration
    Returns:
        w (np.array): w vector of the separating hyperplane
        b (int): intercept
        history (dictionary): history of the training loss
    """

    if method == "fw":
        alpha, history, iterates = fw(X, y, k, C)
    if method == "away_fw":
        alpha, history, iterates = away_fw(X, y, k, C, epsilon)


    w = np.dot(alpha.T * y , X)
    chi = alpha / C

    b = (1 - chi) / y - np.dot(w, X.T)
    b=b.T

    b = np.mean(b[np.where(alpha > 0.00001)])


    return w, b, history, alpha, iterates


def predict(X, w, b):
  y_hat = np.sign(np.dot(w,X.T) + np.full(X.shape[0],b))
  return np.array(y_hat) #[0]


def accuracy(y, y_hat):
  acc = abs(y_hat[y_hat == y]).sum() / len(y) * 100
  return acc

  import psutil
from time import process_time_ns

# Start recording time
start_time = process_time_ns()

# Record initial CPU and memory usage
process = psutil.Process()
initial_cpu_time = process.cpu_times()
initial_memory_info = process.memory_info()

# Execute your function
w, b, history, chi, iterates = svm(X_train, y_train, k=1000, C=1, type="soft", method="
    away_fw", epsilon=0.5)  # chi = alpha

# Calculate elapsed time
finish_time = (process_time_ns() - start_time) / 1e9

# Record final CPU and memory usage
final_cpu_time = process.cpu_times()
final_memory_info = process.memory_info()

# Predict and evaluate
y_hat = predict(X_test, w, b)
acc = accuracy(y_test, y_hat)
f1 = f1_score(y_test, y_hat)
```

```python

# Calculate CPU and memory usage
cpu_time_used = (final_cpu_time.user + final_cpu_time.system) - (initial_cpu_time.user +
    initial_cpu_time.system)
memory_used = final_memory_info.rss - initial_memory_info.rss

def convergenceHistory(info, xMin, F, p, H=None):
    """
    Compute norms of errors of iterates, function values, and gradients (in p-norm).

    Parameters:
    - info: dictionary with optimization history returned by an optimization function
        - 'xs': iterates (numpy array)
        - 'alphas': step sizes (optional)
    - xMin: true minimum, if empty last entry of info['xs'][:, -1] is used
    - F: dictionary with function-related information
        - 'f': function handler
        - 'df': gradient handler
        - 'd2f': Hessian handler (optional)
    - p: p >= 1: p-Euclidean norm || x ||_p,
         p='M': M weighted 2-norm || x ||_M ie (x^T M x)^{1/2} with M = H or if H = [], M
             = F['d2f'](xMin)
    - H: optional, Hessian matrix

    Returns:
    - con: dictionary with p-norms of convergence of
        - 'x': iterates i.e. || x_k - xMin ||_p, k - iteration index
        - 'f': difference in function values i.e. (f(x_k) - f(xMin)), k - iteration index
        - 'df': gradients i.e. || f(x_k) - f(xMin) ||_p, k - iteration index
    """
    con = {}

    # take last point of iterations if true minimum xMin is not provided
    if xMin is None: # if included, xMin needs to be defined as np.array()
        xMin = info['xs'][:, -1] # final point

    if p == 'M':
        p = 2
        if H is not None:
            M = H
        else:
            M = F['d2f'](xMin) # Hessian handler where xMin is argument of the function
                found in dictionary F

        # convergence of iterates || x_k - xMin ||_M
        xMin = xMin[:, np.newaxis]
        print(xMin.shape[0], xMin.shape[1])
        err = info['xs'] - xMin
        #err = info['xs'] - np.array([xMin] * info['xs'].shape[1]) # removed .T at the end

        con['x'] = np.zeros(info['xs'].shape[1]) # initialise list inside 'con' dictionary

        #for k in range(err.shape[1]):
            #con['x'][k] = np.sqrt(err[:, k] @ M @ err[:, k])
        con['x'] = [np.sqrt(err[:, k] @ M @ err[:, k]) for k in range(err.shape[1])]


    else: # convergence of iterates || x_k - xMin ||_p
        #initialise con['x']
        con['x'] = np.zeros(info['xs'].shape[1])
        con['x'] = np.sum(np.abs(info['xs'] - np.array([xMin] * info['xs'].shape[1]).T)**p
            , axis=0)**(1/p)

    if F is not None:
        # converge of function values: f(x_k) - f(xMin)
        con['f'] = np.zeros(info['xs'].shape[1])

        # each iteration point is stored inside each row of info['xs']
        for k in range(info['xs'].shape[1]):
            con['f'][k] = F['f'](info['xs'][:,k]) - F['f'](xMin)

        # convergence of gradient: || f(x_k)||_p
        con['df'] = np.zeros(info['xs'].shape[1])
        for k in range(info['xs'].shape[1]):
            con['df'][k] = np.sum(np.abs(F['df'](info['xs'][:,k]))**p)**(1/p)

    return con
C=1
grad_f = lambda x: grad(X_train, y_train, x, C)
loss_f = lambda x: loss(X_train, y_train, x, C)
hessian_f = lambda x: hessian(X_train, y_train, x, C)

F = {'f': loss_f, 'df':grad_f, 'd2f':hessian_f}
```