ML2R Coding Nuggets
SVM Training Using 16 Lines of Plain Vanilla NumPy Code

Christian Bauckhage
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

Rafet Sifa
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

ABSTRACT

We consider L_2 support vector machines for binary classification. These are as robust as other kinds of SVMs but can be trained almost effortlessly. Indeed, having previously derived the corresponding dual training problem, we now show how to solve it using the Frank-Wolfe algorithm. In short, we show that it requires only a few lines of plain vanilla *NumPy* code to train an SVM.

1 INTRODUCTION

In an earlier theory nugget [5], we derived the dual problem of L_2 support vector machine training. Back then, we claimed that it is downright simple to train such an SVM and promised to show this later. In this coding nugget, we now make good on that promise.

First, we recall the basic application scenario for (kernel) L_2 SVMs and formalize the problem of training them. We then discuss how the Frank-Wolfe algorithm [8] allows for solving this training problem and finally implement the respective procedure in *NumPy*. Our main goal is to show that the required coding effort is minimal. In other words, this note is supposed to demonstrate that SVM training requires only a few lines of plain vanilla *NumPy* code.

Our discussion will assume that readers know about the theory behind SVMs. Those who would like to experiment with our code should be familiar with *NumPy* [13] and only need to

```
import numpy as np
import scipy.spatial as spt
```

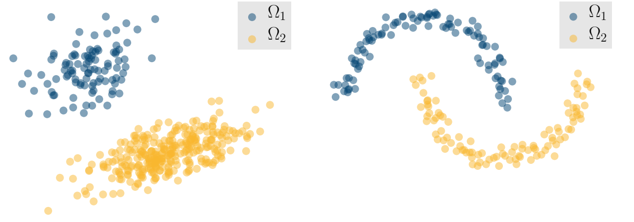
2 THEORY

To begin with, we recall the basic setting of binary classification. Our discussion will be slightly more general than in [5], because a more general perspective will allow us to train kernel SVMs which can be applied to linearly separable as well as to non-linearly separable data (see Fig. 1).

Consider a set of labeled training data $\{(\boldsymbol{x}_j, y_j)\}_{j=1}^{n}$ where the data $\boldsymbol{x}_j \in \mathbb{R}^m$ were sampled from two classes Ω_1 and Ω_2 and the labels

$$y_j = \begin{cases} +1, & \text{if } \boldsymbol{x}_j \in \Omega_1 \\ -1, & \text{if } \boldsymbol{x}_j \in \Omega_2, \end{cases}$$

indicate class membership. The problem of training a binary classifier for such data consists in estimating the parameters of a suitable function $y : \mathbb{R}^m \to \{-1, +1\}$ that can predict correct class labels for newly observed data $\boldsymbol{x}$.



(a) linearly separable training data (b) non-linearly separable training data

Figure 1: Two examples of training data $\boldsymbol{x}_j \in \mathbb{R}^2$ from from two classes. On the left, the two classes are linearly separable and can thus be dealt with using a linear classifier; in the example on the right, they are not and cannot.

A common ansatz for the function y is to assume that it is a linear classifier which computes

$$\begin{aligned} y(\boldsymbol{x}) &= \begin{cases} +1, & \text{if } \boldsymbol{w}^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) - b \geq 0 \\ -1, & \text{otherwise} \end{cases} \\ &= \operatorname{sign}\bigl(\boldsymbol{w}^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) - b\bigr) \end{aligned}$$

Here, $\boldsymbol{\varphi} : \mathbb{R}^m \to \mathbb{H}$ is a transformation or *feature map* which maps the input data into a (possibly infinite dimensional Hilbert) feature space which we denote by $\mathbb{H}$. The parameters of such a classifier that need to be learned from the training data are the *weight vector* $\boldsymbol{w} \in \mathbb{H}$ and the *bias* value $b \in \mathbb{R}$.

The problem of learning these parameters is usually formalized as an optimization problem. Depending on the optimization criterion, the resulting estimates may differ and the trained classifier will go by different names (naïve Bayes classifier, least squares classifier, linear discriminant classifier, support vector machine, ...).

But the more daunting question at this point is how to estimate a possibly infinite dimensional weight vector?

One common strategy is to devise training procedures where feature vectors only occur in form of inner products with other feature vectors. This way, we may invoke the kernel trick which replaces inner products in potentially infinite dimensional spaces by kernel function evaluations.

Among the many possible classifiers, SVMs are interesting not the least because they naturally lend themselves to kernel methods. Alas, SVM training involves solving a constrained optimization problem which may be difficult. Then again, SVMs come in different flavors [7, 14, 16] and we next work out that L_2 SVMs [1, 9, 11] can be trained rather easily.

2.1 Primal and Dual L_2 SMV Training Problem

Recall that support vector machines for binary classification learn the max-margin hyperplane between (transformed) training data from two classes. Since a hyperplane is given by $\boldsymbol{w}^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) - b = 0$, the idea is thus to estimate $\boldsymbol{w}$ and b such that the *margin* $\rho \in \mathbb{R}$ between the plane and the nearest $\boldsymbol{\varphi}(\boldsymbol{x}_j)$ from either class is as large as possible. Yet, if the two classes are not linearly separable, there is no separating hyperplane. This is usually dealt with by trying to find a plane such that the individual margins $\rho - \xi_j$ are maximized where the ξ_j are elements of a vector $\boldsymbol{\xi} \in \mathbb{R}^n$ of *slack variables*.

Given these prerequisites, we recall that the primal problem of training an L_2 SVM consists in solving

$$\begin{aligned} \boldsymbol{w}_*, b_*, \rho_*, \boldsymbol{\xi}_* = \underset{\boldsymbol{w}, b, \rho, \boldsymbol{\xi}}{\operatorname{argmin}} \ & \tfrac{1}{2}\bigl(\boldsymbol{w}^\intercal \boldsymbol{w} + C \cdot \boldsymbol{\xi}^\intercal \boldsymbol{\xi}\bigr) - \rho \\ \text{s.t.} \ & \boldsymbol{\Psi}^\intercal \boldsymbol{w} - b \cdot \boldsymbol{y} - \rho \cdot \mathbf{1} + \boldsymbol{\xi} \geq \mathbf{0} \end{aligned} \tag{1}$$

where $C \geq 0 \in \mathbb{R}$ is a parameter for tuning the slack and $\mathbf{0}, \mathbf{1} \in \mathbb{R}^n$ denote the vectors of all zeros and ones, respectively. The vector

$$\boldsymbol{y} = \bigl[y_1, y_2, \ldots, y_n\bigr]^\intercal \tag{2}$$

gathers the given training labels y_j and the columns of the matrix

$$\boldsymbol{\Psi} = \bigl[y_1 \cdot \boldsymbol{\varphi}(\boldsymbol{x}_1), y_2 \cdot \boldsymbol{\varphi}(\boldsymbol{x}_2), \ldots, y_n \cdot \boldsymbol{\varphi}(\boldsymbol{x}_n)\bigr] \tag{3}$$

contain the feature space representations of the training data $\boldsymbol{x}_j$ weighted by their respective label values y_j.

We furthermore recall that the corresponding dual problem of training an L_2 SVM consists in solving

$$\begin{aligned} \boldsymbol{\mu}_* = \underset{\boldsymbol{\mu}}{\operatorname{argmax}} \ & -\tfrac{1}{2}\, \boldsymbol{\mu}^\intercal \Bigl[\boldsymbol{\Psi}^\intercal \boldsymbol{\Psi} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Bigr] \boldsymbol{\mu} \\ \text{s.t.} \ & \mathbf{1}^\intercal \boldsymbol{\mu} = 1 \\ & \boldsymbol{\mu} \geq \mathbf{0} \end{aligned} \tag{4}$$

Here, $\boldsymbol{I}$ denotes the $n \times n$ identity matrix and the vector

$$\boldsymbol{\mu} = \bigl[\mu_1, \mu_2, \ldots, \mu_n\bigr]^\intercal \tag{5}$$

is a vector of Lagrange multipliers. Its optimal choice $\boldsymbol{\mu}_*$, i.e. the solution to (4), allows for computing the optimal parameters $\boldsymbol{w}_*$ and b_* of the SVM we are after because we recall that

$$\boldsymbol{w}_* = \boldsymbol{\Psi}\boldsymbol{\mu}_* \tag{6}$$

$$b_* = -\boldsymbol{y}^\intercal \boldsymbol{\mu}_* \tag{7}$$

From the point of view of "pen and paper mathematics", we could already conclude our discussion because (6) and (7) characterize the optimal parameters of an L_2 SVM.

However, from the point of view of practical computation, we still have work to do. First of all, we have not yet worked out how to actually solve (4) for $\boldsymbol{\mu}_*$. Second of all, we need to address the disquieting fact that the columns of matrix $\boldsymbol{\Psi}$ in (4) are feature space vectors. Since these may potentially be infinite dimensional, it is not yet clear how we could practically solve (4) or evaluate the expression in (6). Next, we first address the latter issue and then the former.

2.2 Invoking the Kernel Trick

Observe that the occurrence of $\boldsymbol{\Psi}$ in (4) is in form of an $n \times n$ **Gram matrix** $\boldsymbol{\Psi}^\intercal \boldsymbol{\Psi}$ whose elements amount to

$$\bigl[\boldsymbol{\Psi}^\intercal \boldsymbol{\Psi}\bigr]_{ij} = y_i \cdot \boldsymbol{\varphi}(\boldsymbol{x}_i)^\intercal \boldsymbol{\varphi}(\boldsymbol{x}_j) \cdot y_j \tag{8}$$

Hence, if we consider a **Mercer kernel** $k : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ where

$$k\bigl(\boldsymbol{x}_i, \boldsymbol{x}_j\bigr) = \boldsymbol{\varphi}(\boldsymbol{x}_i)^\intercal \boldsymbol{\varphi}(\boldsymbol{x}_j) \tag{9}$$

we can equivalently write the elements of $\boldsymbol{\Psi}^\intercal \boldsymbol{\Psi}$ in terms of kernel evaluations, namely

$$\bigl[\boldsymbol{\Psi}^\intercal \boldsymbol{\Psi}\bigr]_{ij} = y_i \cdot k\bigl(\boldsymbol{x}_i, \boldsymbol{x}_j\bigr) \cdot y_j \tag{10}$$

$$= y_i \cdot K_{ij} \cdot y_j \tag{11}$$

$$= K_{ij} \cdot y_i \cdot y_j \tag{12}$$

Indeed, if we collect the numbers K_{ij} in an $n \times n$ kernel matrix $\boldsymbol{K}$, we can rewrite the whole Gramian as

$$\boldsymbol{\Psi}^\intercal \boldsymbol{\Psi} = \boldsymbol{K} \odot \boldsymbol{y}\boldsymbol{y}^\intercal \equiv \boldsymbol{G} \tag{13}$$

where $\odot$ denotes the **Hadamard product** or element-wise product of two matrices.

Expressed in terms of kernel evaluations, the dual problem of training an L_2 SVM thus becomes solving

$$\begin{aligned} \boldsymbol{\mu}_* = \underset{\boldsymbol{\mu}}{\operatorname{argmax}} \ & -\tfrac{1}{2}\, \boldsymbol{\mu}^\intercal \Bigl[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Bigr] \boldsymbol{\mu} \\ \text{s.t.} \ & \mathbf{1}^\intercal \boldsymbol{\mu} = 1 \\ & \boldsymbol{\mu} \geq \mathbf{0} \end{aligned} \tag{14}$$

This now appears doable because computing matrix $\boldsymbol{G}$ does not require evaluating feature space inner products. Rather, its elements can be determined by evaluating kernel functions such as the linear kernel or the Gaussian kernel for pairs of given training data points.

With respect to practically computing the optimal weight vector, we observe that plugging $\boldsymbol{w}_* = \boldsymbol{\Psi}\boldsymbol{\mu}_*$ into the classifier function $y(\boldsymbol{x}) = \operatorname{sign}\bigl(\boldsymbol{w}^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) - b\bigr)$ provides us with

$$y(\boldsymbol{x}) = \operatorname{sign}\bigl(\boldsymbol{\mu}_*^\intercal \boldsymbol{\Psi}^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) - b_*\bigr) \tag{15}$$

Looking at this expression, we further note that the elements of the vector $\boldsymbol{\Psi}^\intercal \boldsymbol{\varphi}(\boldsymbol{x})$ are given by

$$\bigl[\boldsymbol{\Psi}^\intercal \boldsymbol{\varphi}(\boldsymbol{x})\bigr]_j = y_j \cdot \boldsymbol{\varphi}(\boldsymbol{x}_j)^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) \tag{16}$$

$$= y_j \cdot k\bigl(\boldsymbol{x}_j, \boldsymbol{x}\bigr) \tag{17}$$

$$= y_j \cdot k_j(\boldsymbol{x}) \tag{18}$$

If we now collect the numbers $k_j(\boldsymbol{x})$ in a kernel vector $\boldsymbol{k}(\boldsymbol{x})$, we realize that the computations of our classifier can be written as

$$y(\boldsymbol{x}) = \operatorname{sign}\Bigl(\boldsymbol{\mu}_*^\intercal \bigl(\boldsymbol{k}(\boldsymbol{x}) \odot \boldsymbol{y}\bigr) - b_*\Bigr) \tag{19}$$

This, however, is to say that, in the application phase of the trained classifier, it is not necessary to evaluate feature space inner products involving $\boldsymbol{w}_*$. Instead, we can evaluate kernel functions for given training data points and a new input data point.

Algorithm 1 General Frank-Wolfe algorithm for solving (22).

```
guess a feasible point
        $\boldsymbol{\mu}_0 \in \Delta^{n-1}$
for $t = 0, \ldots, t_{\max}$
    determine the step direction
        $\boldsymbol{s}_t = \underset{\boldsymbol{s} \in \Delta^{n-1}}{\operatorname{argmin}} \; -\boldsymbol{s}^\intercal \nabla \mathcal{D}(\boldsymbol{\mu}_t)$
    update the current estimate
        $\boldsymbol{\mu}_{t+1} = \boldsymbol{\mu}_t + \frac{2}{t+2} \cdot \big[\boldsymbol{s}_t - \boldsymbol{\mu}_t\big]$
```

Algorithm 2 Specific Frank-Wolfe algorithm for solving (22).

```
initialize
        $\boldsymbol{\mu}_0 = \frac{1}{n}\mathbf{1}$
for $t = 0, \ldots, t_{\max}$
    determine
        $i = \underset{j \in \{1,\ldots,n\}}{\operatorname{argmin}} \; \Big[\big[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \frac{1}{C}\boldsymbol{I}\big]\,\boldsymbol{\mu}_t\Big]_j$
    update the current estimate
        $\boldsymbol{\mu}_{t+1} = \boldsymbol{\mu}_t + \frac{2}{t+2} \cdot \big[\boldsymbol{e}_i - \boldsymbol{\mu}_t\big]$
```

2.3 Frank-Wolfe for L_2 SVM Training

Finally addressing the question of how to solve the kernelized dual problem of training an L_2 SVM in (14), we note that its feasible set is the standard simplex

$$\Delta^{n-1} = \Big\{ \boldsymbol{\mu} \in \mathbb{R}^n \;\Big|\; \boldsymbol{\mu} \geq \mathbf{0} \wedge \mathbf{1}^\intercal \boldsymbol{\mu} = 1 \Big\} \tag{20}$$

We further note that its objective function

$$\mathcal{D}(\boldsymbol{\mu}) = -\tfrac{1}{2}\,\boldsymbol{\mu}^\intercal \Big[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Big]\,\boldsymbol{\mu} \tag{21}$$

is quadratic and concave (due to the scaling factor of $-1/2$). But this is to say that maximizing $\mathcal{D}(\boldsymbol{\mu})$ is the same as minimizing $-\mathcal{D}(\boldsymbol{\mu})$. We can therefore rewrite our training problem as

$$\boldsymbol{\mu}_* = \underset{\boldsymbol{\mu} \in \Delta^{n-1}}{\operatorname{argmin}} \; -\mathcal{D}(\boldsymbol{\mu}) \tag{22}$$

which we now clearly recognize as a convex minimization problem over a compact convex set. These are exactly the kind of problems the Frank-Wolfe algorithm [8] was designed for.

The general procedure (presented with respect to the ingredients of our problem) is summarized in Algorithm 1: given an initial feasible guess $\boldsymbol{\mu}_0$ as to the solution, each iteration of the algorithm determines which point $\boldsymbol{s}_t$ in the feasible set Δ^{n-1} minimizes the inner product $-\boldsymbol{s}^\intercal \nabla \mathcal{D}(\boldsymbol{\mu}_t)$. This point is then used to update the current estimate $\boldsymbol{\mu}_{t+1} = \boldsymbol{\mu}_t + \frac{2}{t+2} \cdot [\boldsymbol{s}_t - \boldsymbol{\mu}_t]$.

This conditional gradient scheme guarantees that updates never leave the feasible set. Moreover, one can show that the estimate $\boldsymbol{\mu}_t$ in iteration t is $O(1/t)$ away from the optimal solution [6]. This provides a convenient criterion for choosing the number $t_{\max}$ of iterations to be performed. For instance, if we had reason to believe that a precision of 0.001 is good enough for our problem, it would be sufficient to run $t_{\max} \in O(1000)$ iterations.

What is particularly interesting for our specific problem in (22) is that its rather special form allows for very efficient computations when running Frank-Wolfe. For instance, regarding the initial guess $\boldsymbol{\mu}_0$, we note that the midpoint of Δ^{n-1} is contained in Δ^{n-1} and therefore feasible. Hence, we simply let

$$\boldsymbol{\mu}_0 = \tfrac{1}{n}\mathbf{1} \tag{23}$$

We also note that finding the step direction $\boldsymbol{s}_t$ in iteration t involves the gradient of $-\mathcal{D}(\boldsymbol{\mu}_t)$ which amounts to

$$\nabla\Big(-\mathcal{D}(\boldsymbol{\mu}_t)\Big) = -\nabla\mathcal{D}(\boldsymbol{\mu}_t) = \Big[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Big]\,\boldsymbol{\mu}_t \tag{24}$$

Given this expression for the gradient, we realize that we have to solve the following optimization problem

$$\boldsymbol{s}_t = \underset{\boldsymbol{s} \in \Delta^{n-1}}{\operatorname{argmin}} \; \boldsymbol{s}^\intercal \Big[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Big]\,\boldsymbol{\mu}_t \tag{25}$$

in each Frank-Wolfe iteration. Since this may still look daunting, we next observe that the objective function in (25) is linear in $\boldsymbol{s}$ and needs to be minimized over the compact convex set Δ^{n-1}. However, a minimum of a linear function over a compact convex set is necessarily attained at a vertex of said set. As the vertices of Δ^{n-1} coincide with the standard basis vectors $\boldsymbol{e}_j$ of $\mathbb{R}^n$, we therefore only have to determine which of these standard basis vectors minimizes the objective. Hence, (25) simplifies to

$$\boldsymbol{e}_i = \underset{\boldsymbol{e}_j \in \mathbb{R}^n}{\operatorname{argmin}} \; \boldsymbol{e}_j^\intercal \Big[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Big]\,\boldsymbol{\mu}_t \tag{26}$$

Next, we note that the inner product between a standard basis vector $\boldsymbol{e}_j$ and an arbitrary vector $\boldsymbol{v}$ amounts to $\boldsymbol{e}_j^\intercal \boldsymbol{v} = v_j$ where v_j is the j-th entry of $\boldsymbol{v}$. This then means that (26) can be solved without actually having to compute any inner product. We merely need to determine the index of the smallest component of the gradient, namely

$$i = \underset{j \in \{1,\ldots,n\}}{\operatorname{argmin}} \; \Big[\Big[\boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \tfrac{1}{C}\boldsymbol{I}\Big]\,\boldsymbol{\mu}_t\Big]_j \tag{27}$$

Given these considerations, the update step in the general Frank-Wolfe algorithms thus specializes to

$$\boldsymbol{\mu}_{t+1} = \boldsymbol{\mu}_t + \frac{2}{t+2} \cdot \big[\boldsymbol{e}_i - \boldsymbol{\mu}_t\big] \tag{28}$$

and we summarize all our problem specific adaptations or simplifications of the Frank-Wolfe procedure in Algorithm 2.

2.4 Intermediate Summary

We have now seen how to formalize the problem of training a kernel L_2 SVM (equation (22)), how to solve this problem (Algorithm 2), and how to use this solution to let the SVM classify newly observed data (equations (7) and (19)).

In the next section, we will put these theoretical insights into practice and demonstrate how to implement all of this in *NumPy*. As claimed in the title of this nugget, this will only require a few lines of code.

Listing 1: training a (linear) L_2 SVM

```python
1  def trainL2SVM(matX, vecY, C=100., tmax=10000):
2      matK = computeLinearKernelMatrix(matX)
3  
4      matY = np.outer(vecY, vecY)
5      matG = matK * matY
6      matH = matG + matY + np.eye(matX.shape[1]) / C
7  
8      return fwL2SVM(matH, tmax)
```

3 PRACTICE

In this section, we demonstrate how to practically train an run kernel L_2 support vector machines for binary classification. We structure our discussion into three parts: 1) training, 2) preparation of application, and 3) application.

3.1 Training

In order to train an L_2 SVM for binary classification, we need labeled training data. In what follows, we therefore assume that the training data points have been collected into a data matrix $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ and that the corresponding label values are available in $\boldsymbol{y} \in \{-1, +1\}^n$. We further assume that this matrix and vector are implemented in terms of *NumPy* arrays

```python
matX = ...
vecY = ...
```

For linearly separable data such as those in Fig. 1(a), we may work with a simple linear kernel

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^\intercal \boldsymbol{x}_j \tag{29}$$

To train a corresponding SVM, i.e. to determine the solution $\boldsymbol{\mu}_*$ to the problem in (22), we may then simply call

```python
vecM = trainL2SVM(matX, vecY)
```

using function `trainL2SVM` as defined in Listing 1. Its arguments are the training data points and labels, the slack tuning parameter C, and the maximum number of Frank-Wolfe iterations $t_{\max}$.

Inside of this function, we first compute an $n \times n$ linear kernel matrix $\boldsymbol{K} = \boldsymbol{X}^\intercal \boldsymbol{X}$ for the training data in $\boldsymbol{X}$. To this end, we apply function `computeLinearKernelMatrix` in Listing 2. Given $\boldsymbol{K}$, we can compute the matrix $\boldsymbol{G} = \boldsymbol{K} \odot \boldsymbol{y}\boldsymbol{y}^\intercal$ which we defined in (13). Given $\boldsymbol{G}$, we can then compute the matrix $\boldsymbol{H} = \boldsymbol{G} + \boldsymbol{y}\boldsymbol{y}^\intercal + \frac{1}{C}\boldsymbol{I}$ which features prominently in the objective function of the dual L_2 SVM training problem. At this point, we have prepared all the ingredients required to run the Frank-Wolfe algorithm for L_2 SVM training. We therefore call a corresponding function `fwL2SVM` and return its result.

Function `fwL2SVM` is defined in Listing 3 and is a straightforward *NumPy* implementation of the pseudo code in Algorithm 2.

Note: This implementation aims at readability rather than at efficiency. The astute reader will have noticed that we implement the step direction $\boldsymbol{e}_i$ for the update in equation (28) as the i-th row of the $n \times n$ identity matrix $\boldsymbol{I}$. If n is large, this helper matrix will consume considerable amounts of memory which may hamper the whole procedure. Function `fwL2SVM_V2` in Listing 3 realizes a much, much more memory efficient but also less readable implementation of the Frank-Wolfe algorithm. Can you see how and why it works?

Listing 2: computing a linear kernel matrix and vector

```python
def computeLinearKernelMatrix(matX):
    return matX.T @ matX

def computeLinearKernelVector(vecX, matX):
    if vecX.ndim == 1: vecX = np.reshape(vecX, (-1,1))
    return vecX.T @ matX
```

Listing 3: Frank-Wolfe procedures for $L2$ SVM training

```python
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
```

Added together, we only needed 16 lines of readable code to set up and accomplish L_2 SVM training (6 lines in Listing 1, 2 lines in Listing 2, and 8 lines in Listing 3). Moreover, our code is plain vanilla *NumPy* code and does not require any special purpose functions from external libraries. Training linear support vector machines is really as simple as that.

But what if we wanted to train an SVM do deal with non-linearly separable data such as in Fig. 1(b)? Here, we may, for instance, work with the Gaussian kernel

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2}{2\,\sigma^2}\right) \tag{30}$$

Will this require substantially more coding efforts? No, not at all!

To set up a corresponding Gaussian kernel matrix $\boldsymbol{K}$, we can use function `computeGaussianKernelMatrix` in Listing 4. For instance, letting $\sigma = 2$, we would then simply have to replace line 2 in Listing 1 by

```python
matK = computeGaussianKernelMatrix(matX, 2.0)
```

That is it! The difference between training an L_2 SVM with a linear- and a Gaussian kernel really just boils down to how the corresponding kernel matrix is computed.

Speaking of computing kernel matrices, we note that function `computeGaussianKernelMatrix` calls the function `squaredEDM` which computes a matrix of squared Euclidean distances between the columns of the two matrices passed as arguments. For an in-depth explanation of the mechanism behind this function, we refer to our earlier discussion in [2].

Listing 4: computing a Gaussian kernel matrix and vector

```python
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)
```

Listing 5: running a (linear) L_2 SVM

```python
1 def runL2SVM(matXtst, matXs, vecYs, vecMs):
2     bias = -vecYs @ vecMs
3     vecK = computeLinearKernelVector(matXtst, matXs)
4 
5     return np.sign((vecK * vecYs) @ vecMs - bias)
```

3.2 Preparation of Application

Above, we just saw how to practically solve the dual kernel L_2 SVM training problem, i.e. how to determine an optimal vector $\boldsymbol{\mu}_*$ of Lagrange multipliers. Yet, before we naïvely set out to use this vector to implement the classifier in (19), we note the following:

The solution $\boldsymbol{\mu}_*$ to the training problem in (22) is typically *sparse*! That is, most of its entries are typically 0.

However, entries of $\boldsymbol{\mu}_*$ that are 0 will not contribute to the value of the expression $\boldsymbol{\mu}_*^\intercal\left(\boldsymbol{k}(\boldsymbol{x})\odot\boldsymbol{y}-\boldsymbol{y}\right)$ that occurs in (19). Put differently, entries μ_j of $\boldsymbol{\mu}_*$ that are 0 will zero out the corresponding y_j and $k(\boldsymbol{x}_j, \boldsymbol{x}) \cdot y_j$. Indeed, this is well known and even reflected in the name "support vector machine", because those training data points $\boldsymbol{x}_j$ for which $\mu_j \neq 0$ are called the *support vectors* of the separating hyperplane that is learned by the mathematical machinery.

In order to determine the support vectors, their labels, and their Lagrange multipliers, we may therefore use

```python
supps = np.where(vecM>0, True, False)
matXs = matX[:,supps]
vecYs = vecY[supps]
vecMs = vecM[supps]
```

This provides us with corresponding arrays of considerably reduced size and thus allows for faster computations during application.

3.3 Application

In order to apply an L_2 SVM for binary classification, we need test- or application data. We therefore assume that corresponding data points $\boldsymbol{x}$ have been gathered in a data matrix $\boldsymbol{X}_{\text{tst}} \in \mathbb{R}^{m \times n_{\text{tst}}}$ which is implemented as a *NumPy* array

```python
matXtst = ...
```

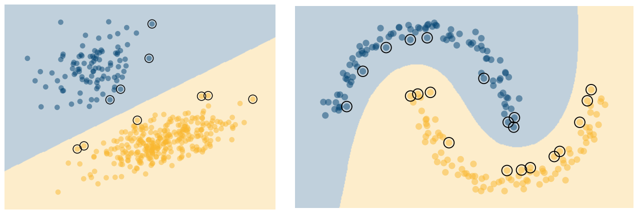
Given this data, we have to compute a vector $\boldsymbol{y}_{\text{tst}} \in \{-1, +1\}^{n_{\text{tst}}}$ whose entries contain label predictions for each of the columns of $\boldsymbol{X}_{\text{tst}}$. To this end, we simply use

```python
vecYtst = runL2SVM(matXtst, matXs, vecYs, vecMs)
```

where function `runL2SVM` is defined in Listing 5. Inside of this function, we compute the bias value b_* according to equation (7) and an array whose columns represent the kernel vectors $\boldsymbol{k}(\boldsymbol{x})$ for the columns of matrix $\boldsymbol{X}_{\text{tst}}$. Our implementation again considers a linear kernel and thus uses function `computeLinearKernelVector` in Listing 2. If we wanted to work with Gaussian kernels, we could instead use function `computeGaussianKernelVector` in Listing 4.

Given `bias` and `vecK`, we can evaluate our classifier for all the given input data; this happens in line 5 which contains a straightforward *NumPy* implementation of the mathematics in (19).

Illustrations of the kind of results we obtain from this approach are shown in Fig. 2. As the classifiers in this figure were trained on the 2D training data in Fig. 1, their decision boundaries can easily be visualized in 2D plots. The classifier on the left is an L_2 SVM with a linear kernel, the classifier on the right is an L_2 SVM with a Gaussian kernel (with $\sigma = 2$). Looking at didactic results like these corroborates that L_2 SVMs perform robust and reliably; looking at the code we discussed in this nugget corroborates that they are easy to train and to apply.



(a) result for the data in Fig. 1(a) (b) result for the data in Fig. 1(b)

Figure 2: Visualization of the decision boundaries of two SVM classifiers trained on the data in Fig. 1. The classifier on the left is an L_2 SVM with a linear kernel, the one the right is an L_2 SVM with a Gaussian kernel. Both panels highlight the support vectors which define the decision boundaries.

4 CONCLUSION

In this note, we fulfilled an earlier promise and demonstrated that L_2 support vector machines are downright easy to train. The key insight was that we can use the Frank-Wolfe algorithm to solve the dual training problem we derived in [5]. Our corresponding implementation amounted to only 16 lines of plain vanilla *NumPy* code and did not require any special purpose libraries whatsoever.

An interesting aspect we did not discuss is that Frank-Wolfe optimization can be understood in terms of recurrent neural network computations [3]. This, in turn, allows for running neural networks that train support vector machines [15] and we will study this in detail in an upcoming note.

Compared to modern deep learning frameworks, support vector machines constitute rather simple baseline approaches. Nevertheless, they achieve very good results in many data scientific applications, especially in settings where training data is scarce [4, 10, 12, 17]. Corresponding practical examples will be discussed in later notes as well.

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (**ML2R**) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01IS18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] C.M. Alaiz and J.A.K. Suykens. 2018. Modified Frank-Wolfe Algorithm for Enhanced Sparsity in Support Vector Machine Classifiers. *Neurocomputing* 320, Dec (2018).

[2] C. Bauckhage. 2014. NumPy / SciPy Recipes for Data Science: Squared Euclidean Distance Matrices. researchgate.net. https://dx.doi.org/10.13140/2.1.4426.1127.

[3] C. Bauckhage. 2017. A Neural Network Implementation of Frank-Wolfe Optimization. In *Proc. ICANN*.

[4] C. Bauckhage and K. Kersting. 2013. Data Mining and Pattern Recognition in Agriculture. *KI – Künstliche Intelligenz* 27, 4 (2013).

[5] C. Bauckhage and R. Sifa. 2021. *ML2R Theory Nuggets: The Dual Problem of L2 Support Vector Machine Training*. Technical Report. MLAI, University of Bonn.

[6] K.L. Clarkson. 2010. Coresets, Sparse Greedy Approximation, and the Frank-Wolfe Algorithm. *ACM Trans. on Algorithms* 6, 4 (2010).

[7] C. Cortes and V. Vapnik. 1995. Support Vector Networks. *Machine Learning* 20, 3 (1995).

[8] M. Frank and P. Wolfe. 1956. An Algorithm for Quadratic Programming. *Naval Research Logistics Quarterly* 3, 1–2 (1956).

[9] T.T. Frieß and R.F. Harrison. 1998. *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.

[10] H. Kondratiuk and R. Sifa. 2021. Swords, Data and Balls: Extracting Extreme Behavioural Prototypes with Kernel Minimum Enclosing Balls. In *Proc. CoG*. IEEE.

[11] O.L. Mangasarian and D.R. Musicant. 2001. Lagrangian Support Vector Machines. *J. of Machine Learning Research* 1 (2001), 161–177.

[12] M. Neumann, L. Hallau, B. Klatt, K. Kersting, and C. Bauckhage. 2014. Erosion Band Features for Cell Phone Image Based Plant Disease Classification. In *Proc. ICPR*. IEEE.

[13] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).

[14] B. Schölkopf, A.J. Smola, R.C. Williamson, and P.L. Bartlett. 2000. New Support Vector Algorithms. *Neural Computation* 12, 5 (2000).

[15] R. Sifa, D. Paurat, D. Trabold, and C. Bauckhage. 2018. Simple Recurrent Neural Networks for Support Vector Machine Training. In *Proc. ICANN*.

[16] J.A.K. Suykens and J.P.L. Venderwalle. 1999. Lest Squares Support Vector Machine Classifiers. *Neural Processing Letters* 9, 3 (1999).

[17] Y. Wu, C. Thurau, and C. Bauckhage. 2010. The Good, the Bad, and the Ugly: Predicting Aesthetic Image Labels. In *Proc. ICPR*. IEEE.