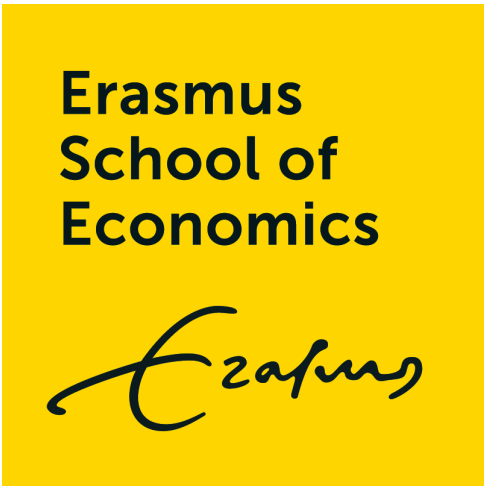
Feature Selection With GenSVM Using Group Lasso Regularization

Daniël de Bondt
416090
supervisor: Prof. dr. Patrick J.F. Groenen
second assessor: U. Karaca

Abstract

An extension is proposed to the GenSVM classification algorithm by replacing the square penalty term by a group lasso regularization. A majorization for this new penalty term is derived and the extended model is implemented in Python. This new technique is then tested on two data sets and compared with the regular GenSVM. The group lasso extension is shown to behave as a feature selector, setting row vectors, corresponding to a specific attribute, of the SVM coefficient matrix collectively to zero. In terms of performance, the group lasso GenSVM is shown to be competitive to the base model.

Bachelor Thesis Econometrics/Economics



Erasmus School of Economics
ERASMUS UNIVERSITY ROTTERDAM
February 3, 2020

Contents

1 Introduction

Over the past decades, immense technological advancements have been made around computers and their computational ability. This development greatly expanded the possibilities for many scientific areas of expertise, one of which is numerical optimization. The field of machine learning emerged: while previously a model would be built around the data, now the data is often the driving force in training the model, sometimes even determining structural relations.

A well-known subset of machine learning problems is classification, where every data point can be attributed to a certain class. The task of the model here is, using specific attributes or features of the data, to predict to which class an unknown data point belongs. Several popular classification methods exist in current literature. The Multinomial Logit gives a classic Econometric approach, allowing for economic interpretation. The K-nearest neighbours algorithm provides a simple intuitive alternative, yet lacks in memory and computational performance (Bhatia et al., 2010). Neural Networks are the current state-of-the-art solution for many different contexts, being especially applicable for abundant data with complex relations such as text or image recognition.

Support vector machines (SVMs), introduced by Cortes and Vapnik (1995), are another machine learning technique aimed at classification. The SVM originated for the binary case of two classes and aims to transform the data in such a way that both classes are linearly seperable. The essential idea of SVM are the support vectors, the observations that are incorrectly classified or close to the classification boundary. Correct classifications are sought after, but there is no measure as to how 'correct' the classification is. The loss function over which the model is optimized only takes into account the (almost) incorrect instances, the support vectors, and this leads to a sparse solution.

The aim of this research will be to extend the GenSVM algorithm introduced by Van den Burg and Groenen (2016). Instead of the squared penalty term in the GenSVM loss function, a group lasso penalty will be used. It is expected that this will cause the algorithm to implement a way of feature selection, selecting a subset of most useful attributes from the available data. For classification tasks within an economic context this would give additional interpretation of the independent variables at hand. In order to be able to include this extension, the GenSVM base algorithm will first have to be implemented separately from the existing package, and as such this will be a major preparatory step.

The rest of this paper will continue as follows. The next section will explore the existing literature surrounding SVMs and the group lasso penalty. Then section 3 will describe the different data sets that are used to test performance. Section 4 will lay out the methodology that is needed to build the GenSVM algorithm and introduce the proposed group lasso extension. Subsequently, section 5 displays the most interesting results and findings. Finally, section 6 will provide a discussion to these results and mention some limitations and further research.

2 Literature

An overview is given by Van den Burg and Groenen (2016) of several ways in which the SVM framework has been extended to a multiclass context in three specific ways. Heuristic approaches, combining multiple binary SVMs are the simplest and also most common, partly because of their availability through software packages. However, they can prove a computational burden. The second set of methods introduced is a generalization of these heuristics through error correcting codes. The main drawback of these however, is the difficulty in finding the correct corresponding coding matrix. The third proposed group of methods is the single machine approach. Here all boundaries are computed simultaneously. These methods often still lead to a very large dual problem, hurting computational time.

GenSVM is introduced by Van den Burg and Groenen (2016) as a single machine multiclass

SVM that generalizes to the binary SVM for K, the number of classes, equal to two. An accompanying iterative majorization method is also derived, to allow for solving the primal problem directly. GenSVM is compared with seven other multiclass SVM methods and was shown to be competitive, while yielding a significant computational efficiency improvement over most of the other methods.

The lasso (least absolute shrinkage and selection operator) method was coined by Tibshirani (1996) in an attempt to combine profitable aspects of both discrete subset selection and ridge regression. A penalty to the coefficients is applied, like in ridge regression, but instead of squaring the coefficients, the sum of the absolute value is taken. It shrinks the coefficients, similar to ridge regression, and while doing so it naturally sets some of them to 0, a way of internal subset selection. This provides a level of sparsity. The method was tested against subset selection and ridge regression for several regression scenarios with competitive yet varying results, highly dependent on the structure of the underlying model.

The lasso works by selecting individual coefficients. For more specific problems where explanatory variables are grouped into factors, Yuan and Lin (2006) have developed an extension, named the group lasso. Here the penalty term is defined as the sum over the ℓ_2 or Euclidean norms of all groups of coefficients. For the case where all groups contain only one element this simplifies to the regular lasso. Within groups however, the penalty behaves similar to the ridge regression, whose penalty involves the same ℓ_2 norm, but squared.

Yuan and Lin (2006) proposed an extended shooting algorithm to solve their group lasso regression. A coordinate descent approach is taken however by Wu et al. (2008) to solve a more general loss function $g(\theta)$ with lasso and group lasso error terms. Here, a first mention is made of majorizing the outer square root of the the group lasso as to help numerical optimization. Additionally, cross validation is brought forward as a way to determine the optimal value of parameter λ.

Yang and Zou (2015) introduce a more general groupwise-majorization-algorithm (GMD) for group lasso penalized learning problems that only requires a loss function to satisfy a quadratic majorization condition. What is of interest here, is that the algorithm is tested on two binary SVMs, which hints at the applicability of a group lasso error within an SVM framework. While the algorithm itself could probably be applied to a group lasso GenSVM, it is hard to predict its efficiency because of its generality.[1]

3 Data

For this research two data tasks will be used. Firstly a comparatively small dataset of breast tissue attributes and classifications, also used in Van den Burg and Groenen (2016) will be assessed to be able to compare the method to previous literature. The goal however, is to find a new application where the expected feature selection will be able to play an important role. For this reason, another dataset concerning drug use will be used. It contains a number of attributes and usage levels of several different drugs, out of which alcohol is selected for this analysis. This second dataset has been created by Fehrman et al. (2017) and an extensive analysis has already been made, to which this paper would be a nice addition. Both datasets can be obtained from the UCI Machine Learning Repository, Dua and Graff (2019). An overview of specifi statistics can be found in Table 1 below. The breast tissue data set has 10 different real valued attributes and its class sizes do not differ by too much, making it a prime candidate to test classification machine learning techniques. The alcohol use data set is somewhat less ideal because the input variables come from survey respondents and are originally categorical in nature. These categorical values are subsequently quantified into real numbers. This should be

[1] It is also way more convenient to make an adjustment to the IM algorithm of Van den Burg and Groenen (2016) than implementing the entire (GMD) algorithm for the GenSVM, which would be outside the scope of this thesis.

Table 1: Data sets summary statistics.

Data set	#instances	#attributes	#classes	smallest class size	largest class size
breast tissue	106	10	9	14	22
alcohol use	1885	12	7	34	795

kept in mind as it may cause some complications in the analysis. Additionally, the group sizes vary by quite a lot in this data set, which may be handled by using group corrected weights, but could still cause some unexpected results. Both data sets are normalized to average around zero. The breast tissue data is scaled into the interval [-1,1], where the alcohol use attributes have differing intervals around zero. While the alcohol data set might not be as clear compared to the breast tissue one, it poses as an example where these machine learning techniques could find a real world, economic application.

4 Methodology

This section lays out the methodology that is used for the proposed group lasso extension. The first part will concern the base GenSVM model and introduce the main idea and notation of the multiclass support vector machine. Next, the intuition behind the group lasso implementation for GenSVM is introduced, along which the lasso and group lasso methodogy will be presented. After this the exact derivation of the majorization for the group lasso is given, followed by a desciption of the practical implementation.

4.1 GenSVM

The GenSVM introduced by Van den Burg and Groenen (2016) tackles the task of classifying a data instance $\mathbf{x}_i$ of m known attributes with unknown label y_i, and assigning it to one of K classes. This is done by training a model around a known data set $\mathbf{X}$ of size $m \times n$, where n is the amount of data instances, and corresponding labels $\mathbf{y}$ of size $1 \times n$. This model consists of a mapping, or weight matrix $\mathbf{W}$ from m to a $(K-1)$ space. More explicitly, the transformation of a data instance $\mathbf{x}_i$ is done in the following way $\mathbf{s}_i' = \mathbf{x}_i'\mathbf{W} + \mathbf{t}'$, where the vector $\mathbf{t}$ of dimension $K-1$ denotes the bias terms. The resulting vector $\mathbf{s}_i'$ is the representation of data instance i in the $(K-1)$ space. To divide this $(K-1)$ space into K classes a K by $(K-1)$ coordinate matrix $\mathbf{U}_K$ of vertices is constructed with distance 1 between each two vertices as described in Van den Burg and Groenen (2016). Each vertex of $\mathbf{U}_K$ corresponds to one of the classes and predictions can be made by selecting the closest vertex to a transformed unknown data point. This matrix also helps us define the miss classification error of instance i with respect to the boundary between class k and j in terms of a projection on the difference between these vertices in the following way:

$$q_i^{(kj)} = (\mathbf{x}_i'\mathbf{W} + \mathbf{t}')(\mathbf{u}_k - \mathbf{u}_j). \tag{1}$$

This error is then propagated through a Huber hinge loss to make sure correct classifications within a certain error margin get zero loss. The Huber hinge is flexible in the κ parameter, allowing the model to adapt to various different data structures, and is given by:

$$h(q) = \begin{cases} 1 - q - \frac{(\kappa+1)}{2} & \text{if } q \leq -\kappa, \\ \frac{1}{2(\kappa+1)}(1-q)^2 & \text{if } q \in (-\kappa, 1], \\ 0 & \text{if } q > 1. \end{cases} \tag{2}$$

Another level of flexibility is added by summing these Huber errors using the ℓ_p norm. This ℓ_p norm determines how much multiple errors within one instance contribute to the total loss.

Additionally the ρ_i parameter is introduced to allow for weighting observations, for example to correct for group sizes. All of the above, combined with the quadratic regularization term, results in the following loss function:

$$L_{(MSVM)}(\mathbf{W}, \mathbf{t}) = \frac{1}{n} \sum_{k=1}^{K} \sum_{i \in G_k} \rho_i \left(\sum_{j \neq k} h^p \left(q_i^{(kj)} \right) \right)^{1/p} + \lambda \operatorname{tr} \mathbf{W}'\mathbf{W}. \tag{3}$$

In order to minimize this loss function, Van den Burg and Groenen (2016) derive an iterative majorization (IM) algorithm. IM works by means of finding an auxiliary majorization function that is greater than the original function and touches the original function at a supporting point, denoted with an overline dash, in this case $\overline{\mathbf{V}}$. If this majorization is chosen to be an easily optimized function, optimization of this majorization function will step-wise also optimize the original function in an efficient manner. A more detailed introduction into IM and a full derivation of the majorization function of GenSVM can be found in Van den Burg and Groenen (2016).

For the GenSVM majorization an extra notation is introduced:

$$\mathbf{V} = [\mathbf{t}\ \mathbf{W}']',$$

$$\mathbf{z}_i' = [1\ \mathbf{x}_i'],$$

$$\boldsymbol{\delta}_{kj} = \mathbf{u}_k - \mathbf{u}_j,$$

$$\text{such that } q_i^{(kj)} = \mathbf{z}_i' \mathbf{V} \boldsymbol{\delta}_{kj}.$$

Using this notation, equation (3) turns into:

$$L_{(MSVM)}(\mathbf{V}) = \frac{1}{n} \sum_{k=1}^{K} \sum_{i \in G_k} \rho_i \left(\sum_{j \neq k} h^p \left(q_i^{(kj)} \right) \right)^{1/p} + \lambda \operatorname{tr} \mathbf{V}'\mathbf{J}\mathbf{V}. \tag{4}$$

A majorization of this loss function, the exact derivation of $\mathbf{A}$ and $\mathbf{B}$ can be found in Van den Burg and Groenen (2016), is derived to be:

$$L_{(MSVM)}(\mathbf{V}) \leq \operatorname{tr} \mathbf{V}'(\mathbf{Z}'\mathbf{A}\mathbf{Z} + \lambda \mathbf{J})\mathbf{V} - 2 \operatorname{tr} (\overline{\mathbf{V}}'\mathbf{Z}\mathbf{A} + \mathbf{B}')\mathbf{Z}\mathbf{V} + \Gamma^{(3)}, \tag{5}$$

where $\Gamma^{(3)}$ is the collection of constants, not dependent on $\mathbf{V}$ and thus irrelevant for the optimization. Taking the derivative with respect to $\mathbf{V}$ and setting to zero gives the following linear system:

$$(\mathbf{Z}'\mathbf{A}\mathbf{Z} + \lambda \mathbf{J})\mathbf{V} = \mathbf{Z}'\mathbf{A}\mathbf{Z}\overline{\mathbf{V}} + \mathbf{Z}'\mathbf{B}, \tag{6}$$

the solution of which determines the Iterative Majorization (IM) update. It can be easily obtained through Gaussian elimination as this system is of the desired form $\mathbf{A}\mathbf{X} = \mathbf{B}$.[2]

4.2 Group Lasso Introduction

GenSVM makes use of a quadratic penalty term to help regularize the coefficient matrix $\mathbf{V}$. By shrinking the weights to a certain extent this prevents overfitting and improves out-of-sample performance. While the penalty term is not part of the core SVM model, it does affect the way it is trained and the optimal solution that is found. Changing this penalty term would thus significantly change the behaviour of the algorithm.

[2] This $\mathbf{A}$ and $\mathbf{B}$ are general matrices, not to be confused with the ones from the derivation in equations (5) and (6)

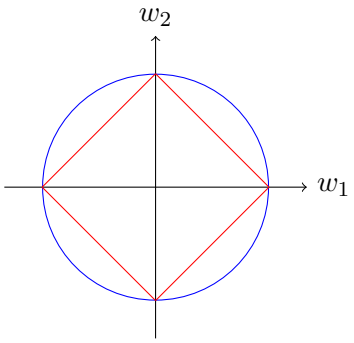


Figure 1: Graphical comparison of the contour lines, both equal to a constant, say 1, for the ℓ_1 (red) and ℓ_2 (blue) norm over two coefficients w_1 and w_2.

The quadratic penalty can be regarded as a squared ℓ_2 norm:

$$\lambda \operatorname{tr} \mathbf{V}'\mathbf{J}\mathbf{V} = \lambda \sum_{i=1}^{m} \sum_{j=1}^{K-1} w_{ij}^2 = \lambda \left\| (\mathbf{W}) \right\|_2^2 . \tag{7}$$

One natural tweak to this would be to instead use the ℓ_1 norm, defined as $\|\mathbf{W}\|_1 \sum_{i=1}^{m} \sum_{j=1}^{K-1} |w_{ij}|$. In fact, this is exactly what Tibshirani (1996) proposes within the context of OLS regression models by introducing the least absolute shrinkage and selection operator (Lasso). It keeps the desirable property of stability from ridge regression (OLS with a squared penalty), while also performing a form of feature selection by setting some parameters to zero. This can be made more intuitive by examining Figure 1. The contour lines for both the ℓ_1 (red) and ℓ_2 (blue) norm are displayed. Imagine now, that a convex loss function, whose global optimum we will assume lies outside both contour lines, were to be optimized within this constraint. For the ℓ_2 norm this constraint optimum could lie anywhere on the circle, since any place on the circle has a unique tangent that could match with the loss contours. For the ℓ_2 norm however, there is a very big chance that the constraint optimum will fall on one of the four corners, since they have a wide range of possible tangents to match with the loss contours. All of these corners lie on an axis, meaning either of the coefficients (w_1 or w_2) is set to zero.

Unlike an OLS setting, where every coefficient β_i corresponds to the effect of one explanatory variable and where lasso was first developed, the GenSVM coefficient matrix $\mathbf{W}$ consists of rows of coordinate coefficients that correspond to a single input variable or attribute. A single element of this matrix is only partly (in one dimension) responsible for the effect of the specific attribute on classification performance. In order to be able to apply the lasso its feature selection in a meaningful way, one would have to group these rows of coordinate coefficients together and penalize them in such a way that entire rows of coefficients are pushed to zero. This is where the group lasso comes into play, first developed by Yuan and Lin (2006). The group lasso penalty is given by:

$$\lambda \sum_{i=1}^{M} \left\| \mathbf{w}_i' \right\|_2 = \lambda \sum_{i=1}^{M} \left(\sum_{j=1}^{K-1} w_{ij}^2 \right)^{1/2} , \tag{8}$$

with $\mathbf{w}_i'$ being the vector of parameters corresponding to group i, in this case the i'th row of $\mathbf{W}$ with elements w_{ij}. If all parameters are in the same group ($M = 1$), this acts similar to a ridge regression, except that the norm is not squared. If all parameters are in their own group (The size of all $\mathbf{w_i}$ is 1), it simplifies to the lasso described above. Plugging this penalty term

back into the loss function of (3) gives the following equation:

$$L_{(GL-MSVM)}(\mathbf{W}, \mathbf{t}) = \frac{1}{n} \sum_{k=1}^{K} \sum_{i \in G_k} \rho_i \left(\sum_{j \neq k} h^p \left(q_i^{(kj)} \right) \right)^{1/p} + \lambda \sum_{i=1}^{M} \left\| \mathbf{w}_i' \right\|_2 \tag{9}$$

4.3 Majorization

Similar to Van den Burg and Groenen (2016) the goal is to derive a quadratic majorization function for our newly established regularization term. Upon further inspection of this term in (8), the only non-linear, non-quadratic part would be the square root. It is therefore sufficient to find a majorization of this operation, after which a term quadratic in $\mathbf{W}$ is left.
The most straightforward majorizing function of the square root in its simplest form $f(x) = \sqrt{x}$ would be a linear function $g(x, \overline{x}) = bx + c$, as illustrated in figure (2). Here, $\overline{x}$ denotes the supporting point, as used in the IM algorithm.

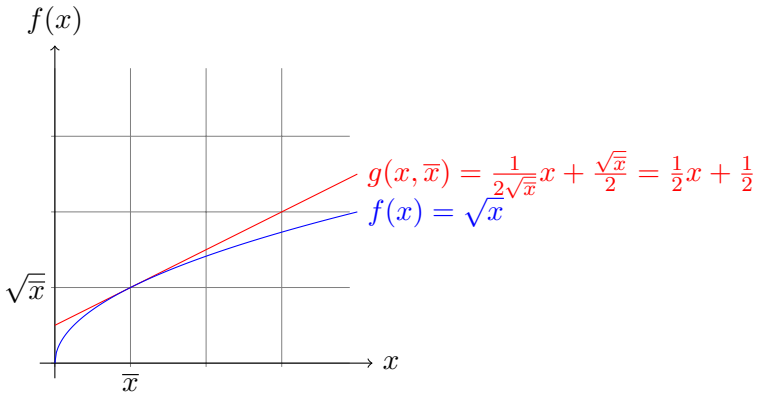


Figure 2: Illustration of a linear majorization function of $f(x) = \sqrt{x}$ where the supporting point $\overline{x} = 1$

To construct the specific majorization function, the following conditions should be met:

$$\begin{aligned} f(\overline{x}) &= g(x, \overline{x}), \\ f(x) &\leq g(x, \overline{x}) \quad \forall x \in \chi. \end{aligned}$$

Using the fact that these conditions imply $\nabla f(\overline{x}) = \nabla g(x, \overline{x})$ a linear majorizing function can be derived for $f(x) = \sqrt{x}$ as:

$$g(x, \overline{x}) = \frac{1}{2\sqrt{\overline{x}}} x + \frac{\sqrt{\overline{x}}}{2}, \tag{10}$$

where especially the term linear in x, $\frac{1}{2\sqrt{\overline{x}}}$, is of great importance, as this is what remains after taking a derivative with respect to x. This will become clearer later. Applying this majorization to the Group Lasso regularization term using the norm notation in (8) gives

$$g(\mathbf{W}, \overline{\mathbf{W}}) = \lambda \sum_{i=1}^{M} \frac{1}{2 \left\| \overline{\mathbf{w}'}_i \right\|_2} \mathbf{w}_i' \mathbf{w}_i + \Gamma^{(4)}, \tag{11}$$

where $\Gamma^{(4)}$ denotes arbitrary constant terms. This expression is to be regarded with some caution though, as the group lasso is expected to push some coefficient vectors $\mathbf{w}_i'$ to zero. In such a case, dividing by the ℓ_2 norm of the supporting vector will cause computational trouble. For this reason a safeguard is built into the implementation and in practice the denominator

in (11) is replaced by $2\max(\|\overline{\mathbf{w}}'_i\|_2, \epsilon)$ for some value ϵ arbitrarily close to zero. In this case $\epsilon = 10^{-32}$ is chosen.

Rewriting the above majorization function into matrix notation and replacing $\mathbf{W}$ with $\mathbf{V} = [\mathbf{t}\ \mathbf{W}']'$, yields:

$$g(\mathbf{V}, \overline{\mathbf{V}}) = \lambda \operatorname{tr} \mathbf{D}\mathbf{V}\mathbf{V}' + \Gamma^{(4)} = \lambda \operatorname{tr} \mathbf{V}'\mathbf{D}\mathbf{V} + \Gamma^{(4)}, \tag{12}$$

$$\begin{aligned} \text{with } \mathbf{D}^{(M+1)\times(M+1)} &= \operatorname{diag}(0, \frac{1}{2\|\overline{\mathbf{w}}'_1\|_2}, \frac{1}{2\|\overline{\mathbf{w}}'_2\|_2}, ...), \\ &= \operatorname{diag}(0, \frac{1}{2\|\overline{\mathbf{v}}'_2\|_2}, \frac{1}{2\|\overline{\mathbf{v}}'_3\|_2}, ...). \end{aligned}$$

Here, $\overline{\mathbf{v}}'_i$ denotes the i'th row of the supporting solution $\overline{\mathbf{V}}$ with the index shifted one place down compared to $\overline{\mathbf{W}}$ because of the top row of constants. This can subsequently be implemented in the majorization of the entire loss function (9) using (5) while replacing $\lambda\mathbf{J}$ with $\lambda\mathbf{D}$:

$$L_{(GL-MSVM)}(\mathbf{V}) \leq \operatorname{tr} \mathbf{V}'(\mathbf{Z}'\mathbf{A}\mathbf{Z}+\lambda\mathbf{D})\mathbf{V} - 2\operatorname{tr}(\overline{\mathbf{V}}'\mathbf{Z}\mathbf{A} + \mathbf{B}')\mathbf{Z}\mathbf{V} + \Gamma^{(4)}, . \tag{13}$$

And similar to the GenSVM derivation, finding the next $\mathbf{V}$ equates to solving the following linear system, originating after taking the derivative with respect to $\mathbf{V}$ and equating to zero:

$$(\mathbf{Z}'\mathbf{A}\mathbf{Z}+\lambda\mathbf{D})\mathbf{V} = \mathbf{Z}'\mathbf{A}\mathbf{Z}\overline{\mathbf{V}} + \mathbf{Z}'\mathbf{B}. \tag{14}$$

4.4 Computational Implementation

In order to incorporate the Group Lasso penalty term into the GenSVM framework, the algorithm from Van den Burg and Groenen (2016) has been implemented independently from the already existing package in C. This has been done in the Python programming language because of both the ease of use and the abundance of powerful auxiliary packages. A class called My_GenSVM is created that contains the entire model including methods for initialization, optimization and many auxiliary computation methods. The full code base for this class can be found in Appendix B[3]. The test runs and analysis is done using Jupyter notebooks, as this allows to store results and variables in a work environment as well as providing easy access to useful visualization methods such as the matplotlib package.

5 Results

As a baseline it makes sense to first validate the rewritten GenSVM implementation by checking if it provides the same results as the package provided by Van den Burg and Groenen (2016). While the package was originally coded in C, it does provide two shells for the higher level languages of Python and R. Unfortunately though, the Python package proved unable to be successfully installed and thus the comparison is to be made between the new Python implementation and the existing package in R. One resulting difficulty is that the random seed is hard to control across the different platforms. This challenge is overcome by exporting the optimal $\mathbf{V}$ from R at iteration 0 and using this as the starting $\mathbf{V}$ for the Python code. The rest of the optimization is completely deterministic with no randomness and as such, the two programs should yield nearly equivalent results, with the only exception being some possible rounding errors. An experiment is run for both programs on the breast tissue data set with the following parameters: $\kappa = 0$, $p = 1$, $\lambda = 2^{-12}$, $\epsilon = 10^{-6}$, using unit weights and the base quadratic penalty term, the result of which can be found in Table 2 below.

[3]This also includes a link to the relevant GitHun repo

Table 2: Comparison of the GenSVM package in R and the new implementation in Python using the breast tissue data set, for $\kappa = 0$, $p = 1$, $\lambda = 2^{-12}$, $\epsilon = 10^{-6}$ and unit weights.

Program	Iterations	Runtime (s)	Final loss	v_{11}	v_{21}
Package in R	1782	0.261838	0.5045575814538870	-9.4337379	-2.5625487
Python	1782	162.362478	0.5045575814539558	-9.43373792	-2.56254874

Both programs took the same 1782 iterations to reach a sufficient solution. The obtained losses are very similar, only differing by a value smaller than 10^{-13}, insignificant with a stopping criterion of $\epsilon = 10^{-6}$. The coefficients are equal, with Python reporting an extra decimal of accuracy. The only stark difference would be the computational time, where the R package (with a C backend) is about 600 times faster than the Python implementation. From these results it can be concluded that the Python implementation was done correctly, yielding the same result as the established GenSVM package. One big downside however, is the computational inefficiency of the Python code. While this does not pose any direct issues, it will limit the ability to use thorough cross-validation in the optimization of parameters.

Now that the validity of the base code is established, the group lasso extension can be further examined and tested. One expected property of the group lasso is the feature selection by means of setting groups of coefficients, in this case attribute vectors, to zero. This can be illustrated using profile plots, where for a range of values for λ the ℓ_2 norm over specific attributes is plotted. Figures 3 and 4 display these profile plots for $\lambda \in \{2^{-14}, 2^{-13}, ..., 2^4\}$ on the breast tissue data set for the base GenSVM and the group lasso extension respectively. The in- and out-of-sample hitrates are also displayed on the right axis to indicate what values for λ are relevant in terms of performance. For retrieving the out-of-sample hitrate a simple validation technique is used, where a random 20% of the data is kept apart as a test set.[4]

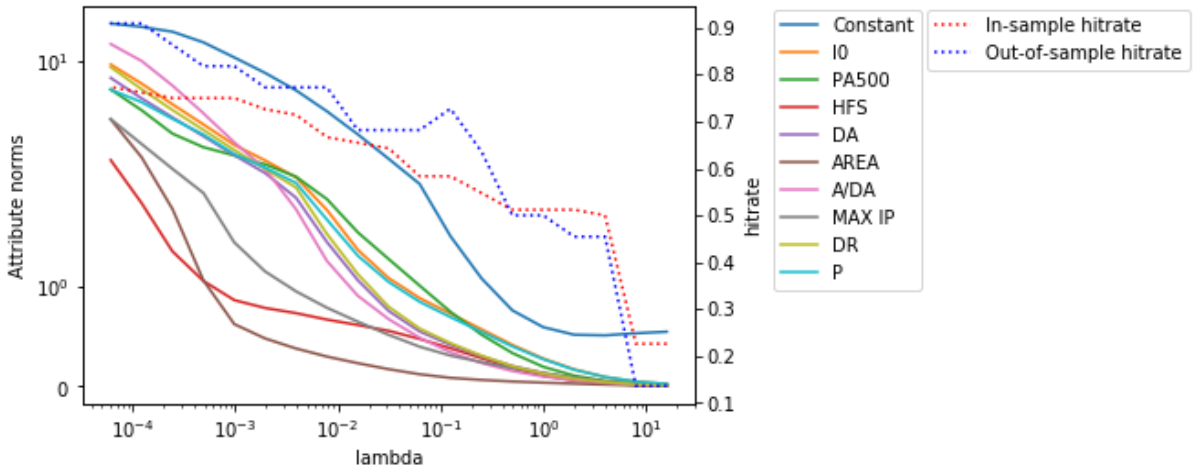


Figure 3: Profile plots of the attribute norms for the base GenSVM algorithm on the breasttissue dataset, for different values of lambda. The in- and out-of-sample hitrates are also displayed on the right axis. $p = 1$, $\kappa = 0$, $\epsilon = 10^{-6}$, unit weights

As lambda increases, the coefficients are penalized more heavily and should thus shrink, this holds for both the regular GenSVM and the group lasso extension. For the regular GenSVM, with a quadratic penalty, one would expect this process to occur continuously. This hypothesis is confirmed by the findings in figure 3. Figure 4 displays a very different picture however.

[4]This could of course be improved by a more sophisticated (k-fold) cross validation, but this is omitted due to computational constraints

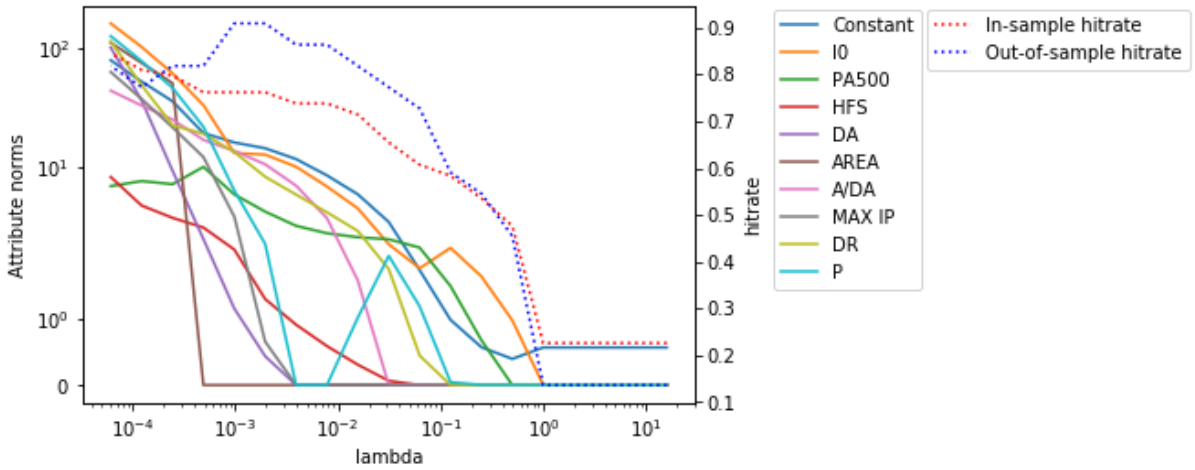


Figure 4: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix (corresponding with specific attributes) for the extended GenSVM algorithm with Group Lasso penalties on the breasttissue dataset, for different values of lambda. The in- and out-of-sample hitrates are also displayed on the right axis. $p = 1$, $\kappa = 0$, $\epsilon = 10^{-6}$, unit weights

It can clearly be seen that the group lasso makes a trade-off between entire attribute vectors and doing so, the weakest predictor falls first, here AREA, the area under the spectrum. Eventually all coefficient vectors get set to zero one after the other as λ increases. What is interesting to note is that for the optimal value for λ in terms of out-of-sample performance (blue), which lies around 2^{-10} and 2^{-8}, there are already 3 attributes at or close to zero. This means that the model actually has better predictive accuracy without these attributes included. It can be argued that the feature selection that the group lasso brings to the table thus improves the model's performance, or at least for this data set. Furthermore it can be seen that the base model has an optimal value of λ slightly lower than the group lasso extension. This can be explained by the square root in the group lasso penalty term that, apart from changing the structure, also decreases the overall level of the penalty. This allows for a comparatively larger λ in the extension to get a penalty level similar to the base model.

What was also found during the experiments surrounding these profile plots was that the group lasso extension behaves quite unexpectedly to certain kinds of warm starts. Van den Burg and Groenen (2016) describe warm starts as a way to speed up training by simply starting the optimization at the optimal solution of the previous set of parameters. However, in applying this technique for a series of λ in descending order this yielded much different results compared to the same experiment with a cold random starting solution. The high λ in the first few runs set a lot of attribute vectors to zero, but it appears that in consequent runs with lower λ it remains very difficult to escape these zero coefficient values. It seems the solution space is very flat around these solutions, often resulting in the first next step already satisfying the ϵ stopping criterion and thus yielding little improvement albeit sufficient by the algorithm's terms. The relevant profile plots describing this phenomenon can be found in Appendix A.1.

Next, the second data set will be examined. The alcohol use data set has much varying class sizes and it thus makes sense to use weights ρ_i that correct for these group sizes and these weights are used for all results concerning this data. For this data set, the profile plots seem much less interpretable. Starting with the profile plot for the group lasso extension displayed below in figure 5.

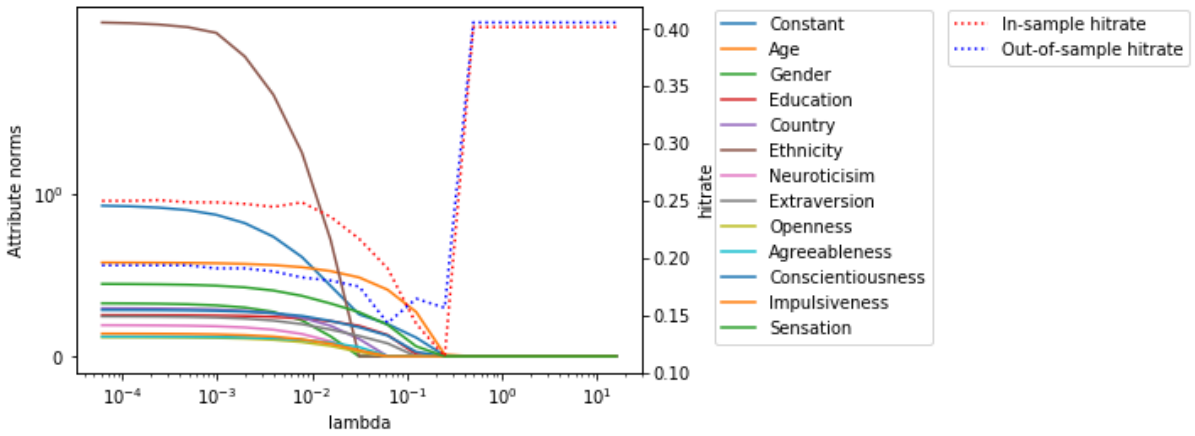


Figure 5: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix (corresponding with specific attributes) for the extended GenSVM algorithm with Group Lasso penalties on the alcohol use dataset, for different values of lambda. The in- and out-of-sample hitrates are also displayed on the right axis. $p = 1$, $\kappa = 0$, $\epsilon = 10^{-6}$, group weights

Here, a similar occurrence as in figure 4 can be observed where the attribute norms are pushed to zero. Most surprisingly, these zero coefficients seem to have a better performance in terms of hitrate than an actual fitted model. This leads to believe hitrate might actually not be as good as a performance measure, since a constant prediction for a frequent class appears to outperform an actual data based prediction. This same phenomenon was described by Van den Burg and Groenen (2016) and the adjusted Rand Index (ARI) was brought forward as a solution to this issue. The ARI uses random predictions as a base line and describes the extra discriminatory power the model predictions have compared to this random base line. The same results as figure 5, but now with the ARI, are displayed in figure 6 below.

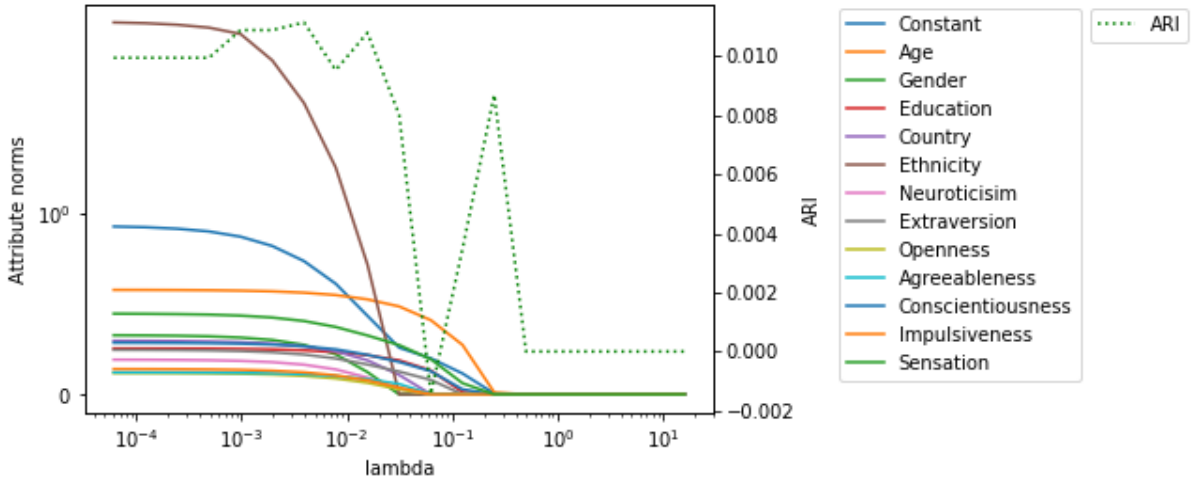


Figure 6: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix (corresponding with specific attributes) for the extended GenSVM algorithm with Group Lasso penalties on the alcohol use dataset, for different values of lambda. The adjusted Rand Index is also displayed on the right axis. $p = 1$, $\kappa = 0$, $\epsilon = 10^{-6}$, group weights

While the ARI does favor a model with relevant coefficients, other than the hitrate, it still seems very inconsistent even yielding a negative value for $\lambda = 2^{-4}$. This may be due to the specific test and training set division. When we examine the results for the base GenSVM

model on this data set though, displayed in figure 7, this inconsistency does not show. All coefficients follow a smooth downwards sloping path, similar to the first data set (figure 3). While the ARI does differ among different values of λ, there is a clear optimal $\lambda = 10^{-6}$. Upon the realization that the adjusted Rand Index would act as a better performance measure, it would make sense to reevaluate the earlier results for the breast tissue data set including an ARI. This has been done and results are shown in Appendix A.2, but no significant difference has been found between ARI and hitrates for this data set.

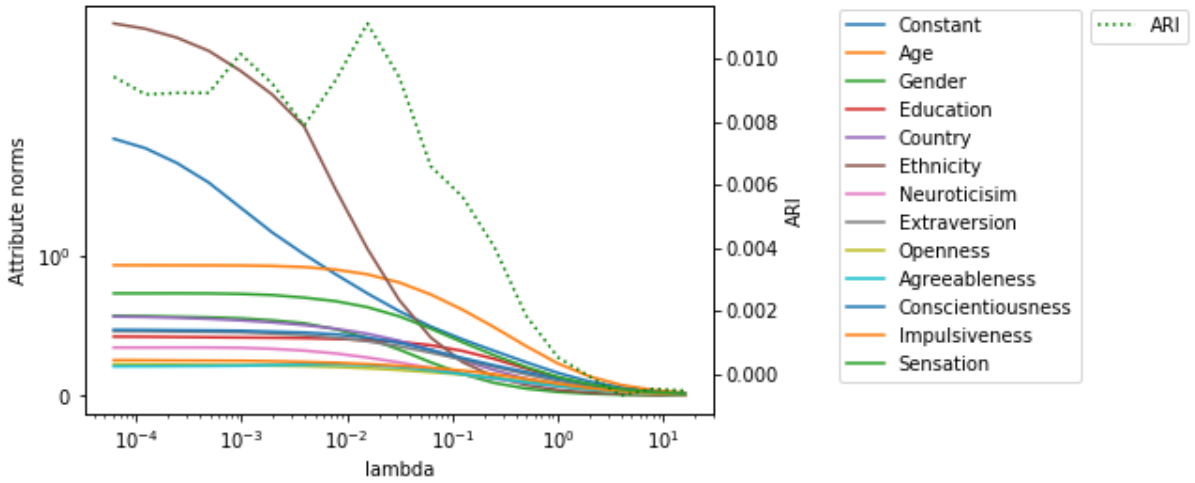


Figure 7: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix (corresponding with specific attributes) for the base GenSVM algorithm on the alcohol use dataset, for different values of lambda. The adjusted Rand Index is also displayed on the right axis. $p = 1$, $\kappa = 0$, $\epsilon = 10^{-6}$, group weights

The preceding profile plots have given an insight into the inner workings of the group lasso penalty extension and its effects on training the GenSVM. While it is nice to see the coefficients behave like what was expected, this does not yet show the real relevance of a group lasso penalty. To this end the performance will be evaluated between the base GenSVM and the group lasso extension. Both models are evaluated for different combinations of parameters, $\kappa \in \{-0.9, 0.5, 5\}$, $p \in \{1.0, 1.5, 2.0\}$ and $\lambda \in \{2^{-14}, 2^{-13}, ..., 2^{-5}\}$, for both of the two data sets. For each pair of κ and p the optimal λ is reported in Tables 3 and 4 along with the corresponding ARI value on the test set, the maxima of which are underlined.

Table 3: Comparison of the GenSVM model and the group lasso extension using the breast tissue data set, for $\epsilon = 10^{-6}$ and using group weights.

Parameters		Base GenSVM		Group lasso	
κ	p	ARI	optimal λ	ARI	optimal λ
-0.9	1.0	0.797989	2^{-14}	0.797989	2^{-10}
-0.9	1.5	<u>0.827560</u>	2^{-14}	<u>0.827560</u>	2^{-10}
-0.9	2.0	<u>0.827560</u>	2^{-14}	<u>0.827560</u>	2^{-9}
0.5	1.0	<u>0.827560</u>	2^{-14}	<u>0.827560</u>	2^{-10}
0.5	1.5	<u>0.827560</u>	2^{-14}	<u>0.827560</u>	2^{-11}
0.5	2.0	<u>0.827560</u>	2^{-14}	<u>0.827560</u>	2^{-11}
5.0	1.0	0.773067	2^{-14}	<u>0.827560</u>	2^{-12}
5.0	1.5	0.724097	2^{-14}	<u>0.827560</u>	2^{-13}
5.0	2.0	0.699062	2^{-12}	0.793073	2^{-13}

Table 4: Comparison of the GenSVM model and the group lasso extension using the alcohol use data set, for $\epsilon = 10^{-6}$ and using group weights.

Parameters		Base GenSVM		Group lasso	
κ	p	ARI	optimal λ	ARI	optimal λ
-0.9	1.0	<u>0.016258</u>	2^{-12}	<u>0.014627</u>	2^{-8}
-0.9	1.5	0.012678	2^{-14}	0.013527	2^{-11}
-0.9	2.0	0.012606	2^{-14}	0.011972	2^{-12}
0.5	1.0	0.012530	2^{-6}	0.012692	2^{-6}
0.5	1.5	0.011172	2^{-10}	0.011135	2^{-8}
0.5	2.0	0.010734	2^{-14}	0.011262	2^{-10}
5.0	1.0	0.012294	2^{-12}	0.013104	2^{-10}
5.0	1.5	0.011335	2^{-12}	0.011239	2^{-14}
5.0	2.0	0.011373	2^{-14}	0.012777	2^{-11}

It appears that for the smaller breast tissue data set (Table 3) both models peak at an ARI of about 0.83, achieved by many different pairs of parameters. This lack of discrimination between the models is probably explained by the size of the test set. The breast tissue data set only has 106 instances and thus a 20% test set will contain no more than 21 testable predictions. It is thus very likely that even differently trained models will make similar predictions for all these 21 cases. What can be noted though, is that the group lasso is slightly more flexible for its choice of parameters, achieving the optimal ARI for the pairs of (5.0,1.5) and (5.0,2.0) where the base GenSVM does not. Also it uses a substantially higher optimal λ, which was already established earlier. For the alcohol usage data set it turns out that both models share an optimal parameter configuration of $\kappa = -0.9$ and $p = 1$. What can also be noted is that the base GenSVM model peaks at a slightly better performance with an ARI of 0.016258, compared to the 0.014627 for the group lasso model. Among the spectrum of parameter settings and data sets however, both models produce quite similar results. The group lasso can at least be said to compete with the original GenSVM algorithm.

6 Discussion

An extension to the GenSVM is proposed by means of a group lasso penalty term. The majorization of this penatly term is derived and implemented within the existing GenSVM algorithm. The algorithm was reprogrammed in Python and it was shown that this implementation works equivalently to the original GenSVM package. From the literature it was expected that this extension would act as a means of feature selection, where the coefficient vectors of certain attributes would be set to zero. This expectation has been confirmed by the findings from experiments on both the breast tissue and alcohol use data sets that were applied in this research. From the profile plots it can clearly be seen that certain attribute vectors are pushed to zero. In some cases the optimal model configuration in terms of ARI or hitrate also contained one or more of these attribute vectors at zero, showing the relevance of this feature selection mechanism in practice.

The general performance of the new group lasso model has also been tested against the base model with a quadratic penalty. The base GenSVM outperformed the group lasso model for

the alcohol usage data set, but both models achieved similar results for the smaller breast tissue data set.

The research and results of this paper are mainly limited by the small range of test cases, caused by a lack of computational power. A mere two datasets are examined and cross validation is kept to only keeping a fifth of the data in a separate test set. While the experiments show some promising results they could be extended and additionally validated by testing on a wide variety of data sets and using a more sophisticated, nested or k-fold, cross validation technique. Given greater computational power this would further explore the possibilities of a group lasso GenSVM in practice. Another possible extension would be to implement the GMD algorithm (Yang and Zou, 2015) for the GenSVM and compare the performance. While this does add to the group lasso penalty application, it could be regarded as an entirely separate project.

To conclude, the proposed group lasso GenSVM extension has proven to be capable of feature selection. It can perform competitively compared to the base GenSVM and provides an extra option for tackling classification problems when the task at hand suits the feature selection property.

References

Bhatia, N. et al. (2010). Survey of nearest neighbor techniques. *arXiv preprint arXiv:1007.0085*.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Dua, D. and Graff, C. (2019). UCI machine learning repository.

Fehrman, E., Muhammad, A. K., Mirkes, E. M., Egan, V., and Gorban, A. N. (2017). The five factor model of personality and evaluation of drug consumption risk. In *Data Science*, pages 231–242. Springer.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Van den Burg, G. J. and Groenen, P. J. (2016). Gensvm: A generalized multiclass support vector machine. *Journal of Machine Learning Research*, 17(224):1–42.

Wu, T. T., Lange, K., et al. (2008). Coordinate descent algorithms for lasso penalized regression. *The Annals of Applied Statistics*, 2(1):224–244.

Yang, Y. and Zou, H. (2015). A fast unified algorithm for solving group-lasso penalize learning problems. *Statistics and Computing*, 25(6):1129–1141.

Yuan, M. and Lin, Y. (2006). Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67.

A Additional Results

A.1 Warm Starts

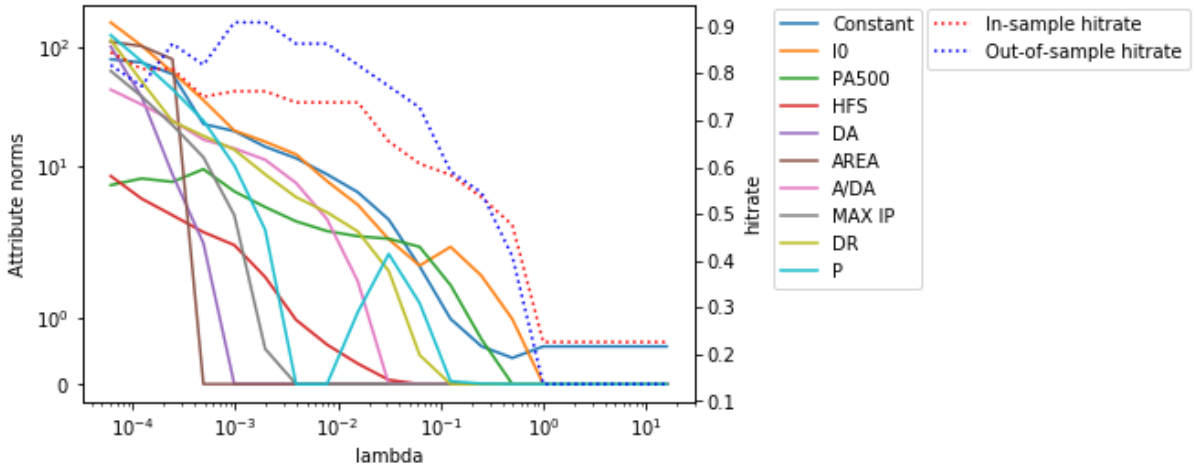


Figure 8: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix similar to figure 4, but now using warm starts to speed up the training. The value for λ is iterated over in ascending order, starting with a small value and ending with the largest.

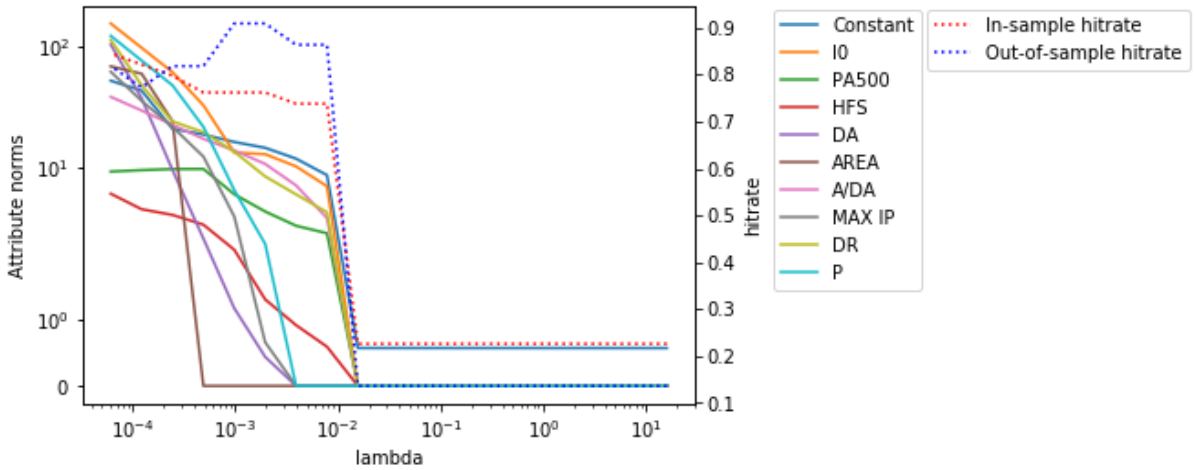


Figure 9: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix similar to figure 4, but now using warm starts to speed up the training. The value for λ is iterated over in descending order, starting with a large value and ending with the smallest.

A.2 Breast Tissue Profiles with ARI

From the below figures it can clearly be seen that the ARI follows the general path of the out-of-sample hitrate for this breast tissue data set, both with quadratic and with group lasso penalty terms. Note that once all coefficients are shrunk to zero the model has no explanatory power anymore as the ARI is shown to also be zero, even though there will still be some lucky hits. This displays a strong argument in favor of using the adjusted Rand Index, but the general results from figures 3 and 4 remain valid.

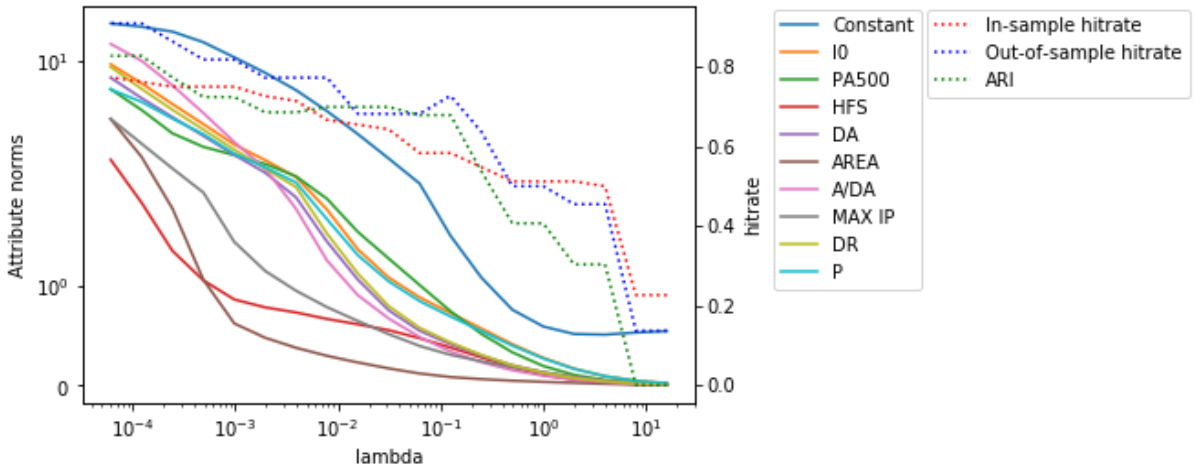


Figure 10: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix for the breast tissue data set similar to figure 3, but now with the ARI included.

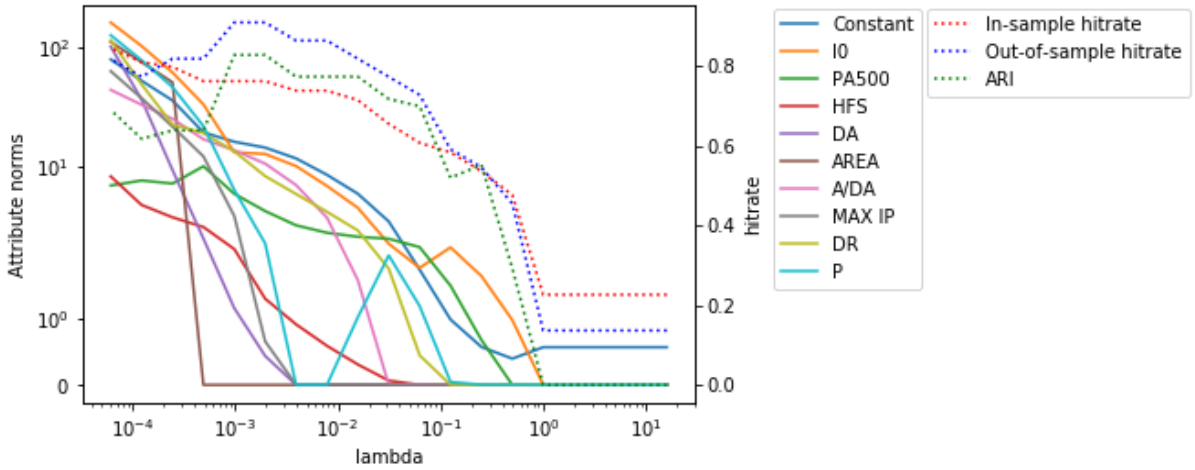


Figure 11: Profile plot of the ℓ_2 norm of rows of the optimal $\mathbf{V}$ matrix for the breast tissue data set using group lasso error terms similar to figure 4, but now with the ARI included.

B Code Implementation

My apologies for the inconvenient font. Since the code contains a few longer the lines this was the best way to fit everything on the page without changing the code structure. A more comprehensive code base including examples can be found on my Github: https://github.com/DanieldeBondt/GL-GenSVM

```
import numpy as np
import time
import sklearn.metrics

class My_GenSVM:
    """The main SVM object"""

    def __init__(self, x, y, rho="unweighted", lamb=10 ** -8, kappa=0, p=1, epsilun=10 ** -6, description="unspecified"
                 extension=False, max_iter=10000, burn_in=51, seed=124):
        self.x = x
        self.y = y
        # Sadly the lambda variable name is unavailable in python, since it has its own functionality, thus lamb
        self.lamb = lamb
        self.kappa = kappa
        self.p = p
        self.epsilun = epsilun
        self.description = description
        self.extension = extension
```

```
        self.max_iter = max_iter
        self.burn_in = burn_in
        np.random.seed(seed)

        # n = number of data instances, m = number of features/attributes
        self.n, self.m = x.shape
        # k = number of classes
        self.k = int(np.max(y))-int(np.min(y))+1
        # rho determines the weights of different classes as to how they impact the total error
        if rho == "unweighted":
            self.rho = np.ones(self.k)
        elif rho == "weighted":
            self.rho = self.weighted_roh()

        # Z is the n by (m+1) matrix of data including the intercept of ones
        self.Z = np.concatenate((np.ones((self.n, 1)), self.x), 1)

        # J is the m+1 diagonal matrix used to transform V to W
        self.J = np.diag(np.ones(self.m + 1))
        self.J[0][0] = 0

        # U_k is the K by (K-1) matrix of vertex coordinates
        self.U_k = self.generate_u_k(self.k)

    # The main iterative majorization algorithm
    def fit_im(self, starting_v=None, printing=True):
        starting_time = time.time()
        # V_hat is the first supporting point of variables t and W' of dimensions (M+1) by (K-1)
        if starting_v is not None:
            v_hat = starting_v
        else:
            v_hat = np.random.randn(self.m + 1, self.k - 1)

        # Initialize other starting values
        losses = []
        hitrates = []
        t = 1
        doubling = False
        loss = self.compute_loss(v_hat)
        loss_prev = (1 + 2 * self.epsilun) * loss
        print("Starting loss:", loss)

        # Loop until convergence is reached
        while (loss_prev - loss)/loss > self.epsilun and t <= self.max_iter:
            alpha_is = np.zeros((self.n, 1))
            beta_is = np.zeros((self.n, self.k-1))
            for i in range(self.n):
                # Find class of instance i
                class_i = int(self.y[i])

                # Initialize local data structures
                supporting_qs = np.zeros((self.k, 1))
                hubers = np.zeros((self.k, 1))
                nonzeros = 0

                # Compute qs (projection distances), their huber hinges and determine epsilon (nonzeros)
                for j in range(1, self.k+1):
                    if j == class_i:
                        continue
                    supporting_q = self.compute_q(i, class_i, j, v_hat)
                    supporting_qs[j-1] = supporting_q
                    hubers[j-1] = self.huber(supporting_q)
                    if hubers[j-1] != 0:
                        nonzeros += 1
                if nonzeros > 1:
                    epsilon = 0
                else:
                    epsilon = 1

                # Initialize local data structures
                small_a = np.zeros((self.k, 1))
                small_b = np.zeros((self.k, 1))

                # Compute a, b and omega and subsequently alpha and beta
                if epsilon:
                    for j in range(1, self.k+1):
                        if j == class_i:
                            continue
                        small_a[j - 1], small_b[j - 1] = self.compute_a_b(supporting_qs[j - 1], 1)
                    alpha_is[i] = self.compute_alpha_simple(small_a, class_i)
                    beta_is[i][:] = self.compute_beta_simple(small_a, small_b, supporting_qs, class_i)
                else:
                    omega = self.compute_omega(hubers, self.p)
                    for j in range(1, self.k+1):
                        if j == class_i:
                            continue
                        small_a[j - 1], small_b[j - 1] = self.compute_a_b(supporting_qs[j - 1], self.p)
                    alpha_is[i] = self.compute_alpha_omega(small_a, omega, class_i)
                    beta_is[i][:] = self.compute_beta_omega(small_a, small_b, supporting_qs, omega, class_i)

            # Construct majorization matrices A, B and for the extension D
            A = np.diag(alpha_is.flatten())
            B = beta_is
            if self.extension:
                D = self.compute_D(v_hat)
                system_a = np.matmul(np.matmul(self.Z.T, A), self.Z) + np.multiply(self.lamb, D)
            else:
                system_a = np.matmul(np.matmul(self.Z.T, A), self.Z) + np.multiply(self.lamb, self.J)
            system_b = np.matmul(np.matmul(np.matmul(self.Z.T, A), self.Z), v_hat) + np.matmul(self.Z.T, B)
```

```
            # Solve the linear system to get V+ (new_V)
            new_V = np.linalg.solve(system_a, system_b)

            # Only start step doubling after 50 iterations burn-in
            if doubling:
                new_V = 2*new_V-v_hat

            # Update loss, store or print diagnostics and set new supporting point V_hat
            loss_prev = loss
            loss = self.compute_loss(new_V)
            losses.append(loss)
            hitrate = self.compute_hitrate(new_V, self.x, self.y)
            hitrates.append(hitrate)
            if printing or (np.mod(t-1, 100) == 0 and t>1):
                print("Iteration: ", t-1)
                print("loss: ", loss)
                print("In_sample_IRA: ", self.compute_ari(new_V, self.x, self.y))
            v_hat = new_V

            # Check if burn-in is over for step doubling
            if t == self.burn_in:
                doubling = True
            t += 1

        total_time = time.time()-starting_time
        print("Training_time: ", total_time)
        print("Iterations: ", t-1)
        print("Final_loss: ", loss)
        return v_hat, losses, hitrates, total_time

    # Given a solution V, predicts the labels of given x using the SVM
    def predict_data(self, V, x):
        t_star = V[0, :]
        W_star = np.delete(V, (0), axis=0)
        s_proj = np.matmul(x, W_star) + t_star

        distances = []
        for row in self.U_k:
            distances.append(np.linalg.norm(s_proj-row))
        label = np.argmin(distances) + 1
        return label

    def compute_ari(self, V, x, y):
        predictions = np.zeros(y.shape)
        for i in range(len(y)):
            predictions[i] = self.predict_data(V, x[i])
        ari = sklearn.metrics.adjusted_rand_score(y.flatten(), predictions.flatten())
        return ari

    # This function computes the hitrate between true and predicted labels
    # Kind of obsolete since packages can do it more efficiently
    def compute_hitrate(self, V, x, y):
        hits = 0
        misses = 0
        for i in range(len(y)):
            prediction = self.predict_data(V, x[i])
            if prediction == int(y[i]):
                hits += 1
            else:
                misses += 1
        hitrate = hits / (hits + misses)
        return hitrate

    # Computes the loss L_MSVM or L_GL-MSVM as described by formulas in the paper
    def compute_loss(self, V):
        summed_loss = 0
        for k in range(1, self.k+1):    # k from 1 to K
            class_range = self.find_class_indices(k)    # find G_k
            for index in class_range:                   # i in G_k
                norm_sum = 0
                for j in range(1, self.k+1):
                    if j == k:
                        continue
                    q_i = self.compute_q(index, k, j, V)
                    huber_q = self.huber(q_i)
                    norm_sum += huber_q**self.p
                summed_loss += self.rho[k - 1] * norm_sum ** (1 / self.p)
        if self.extension:
            # This line could give a sqrt warning, caused by negative non diagonal elements of VV', but since the
            # trace is taken (only over diagonal elements), this error can be ignored.
            regularizer = self.lamb * np.sqrt(np.matmul(self.J, np.matmul(V, V.T))).trace()
        else:
            regularizer = self.lamb * np.matmul(V.T, np.matmul(self.J, V)).trace()

        return summed_loss/self.n + regularizer

    def compute_alpha_simple(self, small_a, class_i):
        # epsilon = 1, we can use the simple majorization
        alpha = (1/self.n) * self.rho[class_i - 1] * np.sum(small_a)
        return alpha

    def compute_alpha_omega(self, small_a, omega, class_i):
        # epsilon = 0, we need to apply omega
        alpha = (1/self.n) * self.rho[class_i - 1] * np.sum(small_a * omega)
        return alpha

    def compute_beta_simple(self, small_a, small_b, supporting_qs, class_i):
        # One row of B, 1 by K-1
```

```
        def beta_map(a, b, q): return b-a*q
        sum = np.zeros((1, self.k-1))
        for j in range(1, self.k+1):
            if j == class_i:
                continue
            element = beta_map(small_a[j-1], small_b[j-1], supporting_qs[j-1])
            delta = self.U_k[class_i-1, :] - self.U_k[j-1, :]
            sum += np.multiply(delta, element)
        return np.multiply(1 / self.n * self.rho[class_i - 1], sum)

    def compute_beta_omega(self, small_a, small_b, supporting_qs, omega, class_i):
        # One row of B, 1 by K-1
        def beta_map(a, b, q): return omega*(b-a*q)
        sum = np.zeros((1, self.k-1))
        for j in range(1, self.k+1):
            if j == class_i:
                continue
            element = beta_map(small_a[j-1], small_b[j-1], supporting_qs[j-1])
            delta = self.U_k[class_i-1, :] - self.U_k[j-1, :]
            sum += np.multiply(delta, element)
        return np.multiply(1 / self.n * self.rho[class_i - 1], sum)

    def compute_a_b(self, x, p):
        # a and b are computed as from Table 4, Appendix C in Van den Burg and Groenen (2016)
        a = 0
        b = 0

        if p != 2 and x <= (p + self.kappa - 1) / (p - 2) :
            a = 1 / 4 * p ** 2 * (1 - x - (self.kappa + 1) / 2) ** (p - 2)                    # (22)
            b = a * x + 0.5 * p * (1 - x - (self.kappa + 1) / 2) ** (p - 1)                   # (20)
        elif x <= - self.kappa:
            a = 1 / 4 * p * (2 * p - 1) * ((self.kappa + 1) / 2) ** (p - 2)                   # (19)
            b = a * x + 0.5 * p * (1 - x - (self.kappa + 1) / 2) ** (p - 1)                   # (20)
        elif x <= 1:
            a = 1 / 4 * p * (2 * p - 1) * ((self.kappa + 1) / 2) ** (p - 2)                   # (19)
            b = a * x + p / (1 - x) * ((1 - x) / np.sqrt(2 * (self.kappa + 1))) ** (2 * p)    # (17)
        elif x > 1:
            if p == 2:
                a = 1 / 4 * p * (2 * p - 1) * ((self.kappa + 1) / 2) ** (p - 2)               # (19)
                b = a*x                                                                       # given
            else:
                a = 1 / 4 * p ** 2 * (p / (p - 2) * (1 - x - (self.kappa + 1) / 2)) ** (p - 2)  # (23)
                b = a*((p*x+self.kappa-1)/(p-2)) + 0.5*p*(p/(p-2)*(1-x-(self.kappa+1)/2))**(p-1)  # (24)
        return a, b

    def compute_omega(self, hubers, p):
        def p_power(x): return x ** p
        omega = (1/p)*np.sum(np.apply_along_axis(p_power, 0, hubers))**(1/p-1)
        return omega

    def compute_q(self, i, y_i, j, V):
        return np.matmul(np.matmul(self.Z[i, :], V), self.U_k[y_i-1, :].T - self.U_k[j-1, :].T)

    # This computes the majorization matrix D for the Group Lasso penalty extension
    def compute_D(self, v_hat):
        diagonal_elements = np.zeros(self.m + 1)
        for i in range(1, self.m+1):
            # If the denominator is zero we need to set it so some positve number close to zero to prevent dividing
            # by zero.
            denom = max(10**-12, (2*np.linalg.norm(v_hat[i, :])))
            element = 1/denom
            diagonal_elements[i] = element

        D = np.diag(diagonal_elements)
        return D

    def find_class_indices(self, k):
        indices = []
        for i in range(len(self.y)):
            if self.y[i] == k:
                indices.append(i)
        return indices

    def huber(self, q):
        output = 0
        if q <= -self.kappa:
            output = 1 - q - (self.kappa+1)/2
        elif q <= 1:
            output = 1/(2*(self.kappa+1))*((1-q)**2)
        return output

    def weighted_roh(self):
        weights = np.zeros(self.k)
        new_y = self.y.flatten().astype(int)
        counts = np.bincount(new_y)
        if 0 not in new_y:
            counts = counts[1:]
        for i in range(self.k):
            weights[i] = self.n/(counts[i]*self.k)
        return weights

    @staticmethod
    def generate_u_k(classes):
        U = np.zeros((classes, classes-1))
        for k in range(1, classes+1):
            for l in range(1, classes):
                if k <= l:
                    U[k-1][l-1] = -1/(np.sqrt(2*(l**2+l)))
                elif k == l + 1:
```

```
                U[k-1][l-1] = 1/(np.sqrt(2*(l**2+1)))
        return U

    def print_model(self):
        print("This is the multiclass SVM for "+self.description)
```