INTERNATIONAL DOCTORAL
SCHOOL OF THE USC

Ziad Akram
Ali Hammouri

PhD Thesis

Large-scale classification based on support vector machine

Santiago de Compostela, 2022

Doctoral Programme in Information Technology Research

TESE DE DOUTORAMENTO

LARGE-SCALE CLASSIFICATION BASED ON SUPPORT VECTOR MACHINE

Ziad Akram Ali Hammouri

ESCOLA DE DOUTORAMENTO INTERNACIONAL
DA UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
PROGRAMA DE DOUTORAMENTO EN
INVESTIGACIÓN EN TECNOLOXÍAS DA INFORMACIÓN

SANTIAGO DE COMPOSTELA
2022

DECLARACIÓN DO AUTOR DA TESE

Don Ziad Akram Ali Hammouri

Presento a miña tese, titulada **Large-scale classification based on support vector machine**, seguindo o procedemento adecuado ao Regulamento, e declaro que:

1. A tese abarca os resultados da elaboración do meu traballo.
2. De ser o caso, na tese faise referencia ás colaboracións que tivo este traballo.
3. Confirmo que a tese non incorre en ningún tipo de plaxio doutros autores nin de traballos presentados por min para a obtención doutros títulos.
4. A tese é a versión definitiva presentada para a súa defensa e coincide a versión impresa coa presentada en formato electrónico.

E comprométome a presentar o Compromiso Documental de Supervisión no caso de que o orixinal non estea na Escola.

En Santiago de Compostela, Marzo de 2022

Asdo. Ziad Akram Ali Hammouri
Autor da tese

AUTORIZACIÓN DO DIRECTOR DA TESE

Don **Manuel Fernández Delgado** Profesor Titular de Universidad da Área de Ciencias da Computación e Intelixencia Artificial, Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, en condición de director da presente tese, titulada **Large-scale classification based on support vector machine**,

INFORMA:

Que a presente tese correspóndese co traballo realizado por Don Ziad Akram Ali Hammouri, baixo a miña dirección, e autorizo a súa presentación, considerando que reúne os requisitos esixidos no Regulamento de Estudos de Doutoramento da USC, e que como director desta non incorre nas causas de abstención establecidas na Lei 40/2015.

En Santiago de Compostela, Marzo de 2022

Asdo. Manuel Fernández Delgado
Director da tese

I dedicate this work to my mother,
father and brother Khaled for their endless support.

I prefer to be a dreamer among the humblest, with visions to be realized, than a lord among those without dreams and desires.

Gibran Khalil Gibran

Acknowledgments

Firstly, the Ph.D. has been a truly life-changing experience for me where I learned and acquired different skills. It would not have been possible to do without the support and guidance that I received from remarkable people.

Secondly, I would like to express my special gratitude and appreciation to my supervisor Prof. Manuel Fernández Delgado. I would like to thank him for assisting me in my research and allowing me to grow and improve my research skills. Thank you for your patience, motivation, and immense knowledge and for the endless support of my Ph.D research. Your supervision assisted me in the entire research and writing of this thesis.

Finally, I would like to thank my family: my parents, sisters and brothers for their continuous support throughout writing this thesis and my life in general. I am also grateful to all my friends and in particular Marcos Matabuena.

This work has received financial support from the Consellería de Cultura, Educación e Universidade (accreditation 2019-2022 ED431G-2019/04) and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS - Centro Singular de Investigación en Tecnoloxías Intelixentes da Universidade de Santiago de Compostela as a Research Center of the Galician University System.

Marzo de 2022

Contents

Resumo en galego

Este tese preséntase como contribución para executar unha serie de métodos de aprendizaxe automática en datos de tamano grande nos que, no momento actual, non se poden executar por motivos de sobrecarga computacional. Específicamente, referímonos á popular máquina de vectores de soporte, coñecida polas súas siglas en inglés *support vector machine* (SVM). Trátase dun algoritmo considerado entre os mellores métodos existentes de aprendizaxe automática tanto para problemas de clasificación (predicción de valores discretos) como de regresión (predicción de valores continuos). A SVM é un método moi consolidado na literatura de aprendizaxe automática dende mediados dos anos 90 pero, como acontece con outros métodos competidores, adolece dun importante custe computacional en termos de tempo de execución e de requerimentos de memoria RAM. Isto limita a súa aplicabilidade a datos de tamano pequeno ou mediano, porque en datos de tamano grande simplemente non é capaz de rematar o seu entrenamento.

A tese actual circunscríbese ao ámbito da SVM aplicada a problemas de clasificación. O capítulo 1 introduce unha concisa descripción da teoría que soporta este método, especialmente co obxectivo de definir unha nomenclatura que será empregada nos capítulos posteriores para definir as metodoloxías propostas neste traballo. Para isto, defínese a "SVM de marxe dura", enfocada á clasificación de datos que poden ser correctamente separados en dúas clases usando unha función linear. Neste contexto, establécese o obxectivo do entrenamento da SVM, que consiste en atopar a fronteira linear (hiper-plano) que maximiza as distancias aos datos das dúas clases. Preténdese minimizar así o risco de clasificar incorrectamente datos novos non usados durante o entrenamento. Esta é a denominada "minimización do risco estructural", enmarcada na teoría da aprendizaxe estatística. Xunto con este obxectivo, a SVM tamén persegue minimizar, que no caso da marxe dura significa anular, o erro de clasificación sobre os datos de entrenamento, o cal recibe o nome de "minimización do risco empírico" na teoría da SVM. O problema de optimización condicionado xerado pola SVM explícase no contexto do método dos multiplicadores de Lagrange e as condicións de Karush-Kuhn-Tucker, ambos elementos da teoría da optimización matemática. A resolución deste problema de optimización require a execución de algoritmos iterativos de cálculo numérico que en xeral resultan lentos. Ademais, o tempo requerido non so depende do tamano dos datos, senón tamén da complexidade do problema de clasificación, podendo o entrenamento ser máis lento en datos máis pequenos porque resultan máis "difíciles" de aprender.

O seguinte paso é o "clasificador de marxe branda", que considera problemas de clasificación binarios que non se poden separar correctamente usando unha función ou fronteira linear. Neste caso, é imposible acadar un erro de entrenamento nulo, de modo que hai que atopar un equilibrio ou compromiso entre a minimización dos riscos empírico e estructural, o cal da lugar á aparición do "hiper-parámetro de regularización" para ponderar ambos riscos. Hai que destacar tamén que, neste caso da marxe branda no que as clases non se poden separar cunha fronteira linear, non so o tempo requerido depende da complexidade dos datos, senón que o proceso de resolución iterativa podería non rematar satisfactoriamente, é decir, non atopar unha solución válida, ou incluso podería non rematar nunca, como acontece con datos grandes.

Neste punto aparece o segundo ingredente esencial no elevado desempeño da SVM, xunto coa minimización do risco estructural. Este segundo ingredente é o uso dunha proxección de tipo cerne non linear para potenciar a función de clasificación linear. Esta proxección emprégase para mapear os datos a un espazo de características de dimensionalidade meirande co orixinal, aumentando así as posibilidades de que unha función linear de clasificación acade resultados aceptábeis, noutras palabras, aumentando a separabilidade linear dos datos proxectados. A función cerne actúa como unha medida xeralizada de similaridade no espazo de características, e o denominado "truco do cerne" (*kernel trick* en inglés) permite efectuar esta proxección de modo implícito, sen necesidade dunha definición explícita para a mesma, xa que o cálculo dos parámetros entrenábeis da SVM so usa os valores da función cerne (similaridade xeralizada) aplicados sobre pares de datos. O único prezo a pagar é que o cerne debe ser adaptado aos datos dispoñíbeis, e isto require a sintonización dos seus hiper-parámetros. No caso do cerne gausiano, que é o máis empregado pola súa boa calidade, o único hiper-parámetro é o ancho da función gausiana.

Esta introducción á teoría da SVM remata explicando as estratexias para a súa extensión a problemas de clasificación multi-clase usando SVMs binarias. Isto introduce un custe computacional adicional que pode resultar importante cando o número de clases é elevado. Aínda que a tese actual centrarase exclusivamente en problemas de clasificación, a introducción describe tamén a extensión da SVM a problemas de regresión (*support vector regression* ou SVR). Para isto, emprégase unha función de custe ε-insensíbel, que so considera que a SVR comete un erro cando a distancia entre os valores verdadeiro e predito superan un umbral ε. O desenvolvemento matemático para problemas de regresión está relacionado co correspondente a clasificación, podéndose usar o mesmo tipo de cerne e sintonización de hiper-parámetros.

Unha vez exposta a teoría da SVM para clasificación (denotada co acrónimo SVC), o capítulo 2 presenta a aproximación proposta nesta tese, denominada fast support vector classifier (FSVC) para convertir a SVC nun algoritmo eficiente que poda ser executado sobre un elevado número de datos, con dimensionalidade elevada e con moitas clases. Este método FSVC publicouse no artigo [16]. Preliminarmente, realízase unha revisión das causas da ineficiencia da SVC con datos grandes, derivadas directamente da teoría descrita no capítulo 1. Estas causas inclúen a execución dun proceso de entrenamento con complexidade crecente co cubo do número de datos, requerindo un espazo de almacenamento que medra co seu cadrado. O almacenamento dos vectores de soporte (datos seleccionados no entrenamento da SVC) e máis dos seus coeficientes asociados tamén require un espazo de almacenamento considerábel para datos grandes. Outro aspecto problemático é a sintonización dos hiper-parámetros, incluíndo a regularización, que afortunadamente pode non ser sintonizada xa que non é moi sensíbel, e os hiper-parámetros do cerne, como o ancho no caso do cerne gausiano, que si ten grande relevancia na calidade da aprendizaxe. A elevada dimensionalidade dos datos tamén afecta ao cálculo das distancias entre datos, necesarias para calcular o cerne. Finalmente, en problemas con moitas clases debe entrenarse un elevado número de SVCs binarias, que medra co cadrado do número de clases, ralentizando enormemente a execución da SVC.

A partires dos problemas diagnosticados realízase unha exposición das ideas aportadas nesta tese para resolvelos. En primeiro lugar, proponse un algoritmo de entrenamento eficiente definido por unha fórmula analítica pechada que calcula os parámetros entrenábeis da SVM directamente a partir dos datos, sen necesidade de ningún método numérico iterativo de optimización que poda resultar lento ou incluso non rematar nunca, en caso de non atopar solucións válidas, como acontece coa SVC. Concretamente, o método proposto permite calcular os coeficientes do cerne aplicado aos datos de entrenamento e teste, e máis o termo independente da función de clasificación linear no espazo de características.

Este método de entrenamento resulta moito máis rápido que a SVC clásica, pero ten o inconvinte de que require tódolos datos de entrenamento, o cal resulta lóxico tratándose dunha expresión pechada. Nembargantes, isto non é aceptábel cando se traballa con datos grandes. Comparada co entrenamento eficiente, o método clásico de entrenamento da SVC ten a vantaxe de que so require unha parte dos datos de entrenamento (os vectores de soporte), aínda que normalmente representa unha porcentaxe importante do total dos datos. Polo tanto, o entrenamento eficiente proposto non é dabondo, xa que a pesar de ser máis rápido, require almacenar tódolos datos, e non so os vectores de soporte. Por esta razón, a tese actual propón

tamén un método eficiente para o cálculo do cerne, consistente en substituir os datos de entrenamento orixinais por unha representación comprimida dos mesmos. Con este obxectivo, créase unha colección de prototipos asociados a cada clase, que se van actualizando a medida que se percorren os datos de entrenamento. O número de prototipos está limitado, de maneira que o consumo de memoria e o número de iteracións non medra indefinidamente para datos grandes. Inicialmente, empréganse os datos de cada clase para definir os seus prototipos asociados, ata acadar o número máximo de prototipos por clase. A partir dese momento, cada dato actualiza o seu prototipo máis cercano entre os da clase á que pertence. Esta actualización require calcular as distancias entre o dato e os prototipos da súa clase, o cal é eficiente e escala co tamano dos datos porque o número de prototipos por clase está limitado e ten un valor reducido. A ponderación entre o prototipo actual e o dato novo ven definida polo número de datos usados ata o momento para actualizar o prototipo. O obxectivo é que cada prototipo sexa a media dos datos desta clase para os cais este prototipo foi o máis cercano. Finalmente, os prototipos de cada clase actúan como promedios dos seus datos orixinais, agrupados por similaridades, pero dunha maneira máis eficiente que almacenando os propios datos, o cal non é posíbel cando os datos son grandes.

Unha das causas máis importantes da lentitude da SVC é a necesidade de sintonizar os hiper-parámetros, xa que isto require probar con distintos valores e seleccionar aqueles que proporcionan mellores resultados. Pero isto implica entrenar e testear a SVC con cada valor, o cal ralentiza enormemente o seu funcionamento. No caso da SVC con cerne gausiano, os hiper-parámetros son a regularización e o ancho do cerne. A tese actual contribúe cun método de sintonización que resulta moi eficiente porque non require executar a SVC para seleccionar o valor óptimo. Dado que o entrenamento eficiente antes mencionado non emprega proceso de optimización ningún, o hiper-parámetro de regularización desaparece en FSVC, quedando somentes a sintonización do ancho do cerne gausiano. Para seleccionar o seu valor, faise uso do concepto de función “cerne ideal”, entendida como unha función de similaridade xeralizada que separa correctamente as dúas clases, aplicándose sobre dous datos e valendo 1 se os dous son da mesma clase e 0 en caso contrario. O método proposto selecciona o ancho da gausiana que proporciona un cerne máis similar ao ideal. Para isto, calcula a matriz de cerne ideal e as matrices de cerne gausiano para cada valor do ancho nunha colección de valores pre-definida na literatura e considerada estándar. O ancho mellor é o que da unha diferencia media menor, en valor absoluto, entre ambas matrices. O tamano das mesmas está determinado polo número de datos, de modo que para executar este método en datos grandes

é necesario usar os prototipos no canto dos datos orixinais. Hai que notar que este método de sintonización é válido para calquera tipo de cerne, non so para o cerne gausiano, e tamén para outros métodos baseados en cernes, como por exemplo *kernel principal component analysis* (Kernel PCA).

É amplamente coñecido na literatura que con datos de elevada dimensionalidade o uso de cernes non lineares, como o gausiano, non aumenta o acerto comparado cun cerne linear, aínda que resulta moito máis lento. Isto débese a que a dimensionalidade orixinal xa é moi elevada, e proxectar a un espazo de meirande dimensión non aporta nada. O uso dun cerne linear en FSVC leva a unha simplificación dos cálculos, que pasan a ser extremadamente eficientes. Con respecto á clasificación multi-clase, FSVC usa por defecto a aproximación *one-vs-one* (OVO), con varias FSVCs binarias que separan entre pares de clases. O número de FSVCs binarias medra cadráticamente co número de clases, o cal resulta inaceptábel cando este último é moi elevado. Neste caso, FSVC pode usar a estratexia *one-vs-all* (OVA), que so necesita tantas FSVCs binarias como clases, o cal resulta moito máis eficiente en datos grandes con moitas clases.

O capítulo 2 reporta o algoritmo FSVC completo, incluíndo o cálculo eficiente do cerne mediante prototipos, a sintonización eficiente do ancho do cerne gausiano, a clasificación multi-clase usando as estratexias OVO e OVA para números de clases baixo e alto, respectivamente, e as etapas de entrenamento e teste da FSVC binaria, usando por defecto un cerne gausiano ou linear, este último preferíbel con datos multi-dimensionais. Tamén se realiza unha análise da complexidade computacional de FSVC, que é linear nos números de entradas e de datos de entrenamento e teste, e cadrático ou linear no número de clases usando OVO ou OVA, respectivamente.

O traballo experimental empregou unha extensa colección de 22 datos de pequeno tamano (menos de 15000 datos, dimensionalidade ata 617 e ata 26 clases) e outros 22 datos de tamano grande (31.5 millóns de datos, dimensionalidade ata 30000 e ata 131 clases). Analízase detalladamente o comportamento dos distintos elementos de FSVC: entrenamento eficiente, cálculo eficiente do cerne, sintonización eficiente do ancho de cerne, cerne linear para dimensionalidades elevadas e aproximación OVA para datos con moitas clases. Executáronse 4 versións alternativas de SVC: con e sen entrenamento eficiente, cálculo eficiente do cerne (é decir, uso de prototipos) e sintonización eficiente do ancho do cerne gausiano. Tamén se executaron 6 versións de FSVC: con e sen entrenamento eficiente, cálculo eficiente do cerne (prototipos), sintonización eficiente, cerne linear para datos de elevada dimensionalidad, aproximación

OVA para datos con moitas clases, e cerne linear con OVA para datos multi-dimensionais con moitas clases. Comparando as distintas versións de SVC, o factor que máis reduce o acerto é o entrenamento eficiente, mentres a sintonización eficiente incluso aumenta o acerto e os prototipos reducen lixeiramente o acerto. Considerando a sintonización eficiente, para algúns datos o ancho que minimiza a diferencia media absoluta entre as matrices de cerne ideal e gausiana pode non proporcionar o mellor acerto, caso no cal selecciónase o ancho que maximiza a derivada desta diferencia.

O método FSVC comparouse coa SVC gausiana e linear implementadas polas librarías LibSVM e LibLinear, consideradas os estándares de ouro para SVM con cernes gausiano e linear, respectivamente, e máis co direct kernel perceptron (DKP), que é unha aproximación eficiente anteriormente proposta para a SVC. O acerto de FSVC é moi cercano á SVC con cerne gausiano e ó DKP, e superior á SVC linear. Ademais, FSVC é moito máis rápida que as outras alternativas. Cos datos grandes ata 31 millóns, 30,000 entradas e 131 clases, onde a SVC gausiana ou o DKP non poden ser executados, a FSVC execútase en tódolos datos mentres a SVC linear falla nos máis grandes, aínda que está deseñada específicamente para datos de elevada dimensionalidade.

A FSVC tamén se comparou con varias implementacións eficientes de SVC para datos grandes, incluindo *primal estimated sub-gradient solver for SVM* (Pegasos-SVM), *SVM with sublinear importance-sampling bi-stochastic algorithm* (SVM-SIMBA), *indefinite core vector machine* (ICVM) e *doubly stochastic gradient* (DSG), que obtiveron menos acerto que FSVC sendo máis lentos e non podendo ser executados nos datos grandes nos que FSVC si se executou. A sintonización eficiente revelouse como a parte do algoritmo FSVC que máis contribúe a acelerar a execución comparada con SVC, mentres o entrenamento eficiente e o cálculo eficiente do cerne (prototipos) son similares entre si pero contribúen bastante menos. Tamén se comparou FSVC co método *steady-state genetic algorithm for instance selection* (SGA), un popular algoritmo evolucionario para a selección de patróns de entrenamento, revelando que a selección xenética é lenta e non pode aplicarse con datos moi grandes, resultando limitada a datos pequenos onde a SVC clásica xa se executa sen reducción dos datos.

Un dos aspectos importantes de FSVC é o manexo da memoria RAM, xa que isto permite a súa execución en datos dun tamano case arbitrariamente elevado sen que se produzan erros por falta de memoria. É decir, aínda que o entrenamento sexa lento, non fallará por causa dunha memoria insuficiente. Para isto, realizouse un estudo do consumo de memoria de FSVC, elaborándose un algoritmo que selecciona automáticamente o tamano do bloque de datos a ler

dende disco. Este bloque é maior ou menor dependendo da memoria dispoñíbel e do tamano dos datos, evitando fallos de memoria e acelerando ou ralentizando a execución con datos pequenos ou grandes. A eficacia deste método probouse executando exitosamente FSVC nos datos grandes incluso con memoria limitada, mentres a SVC linear fallou na maioría destes datos. Esta regulación do tamano dos bloques de datos permite executar FSVC sobre datos grandes incluso en dispositivos con baixa memoria ou potencia computacional.

O método proposto para a sintonización eficiente dos hiper-parámetros do cerne gausiano en FSVC é a aportación desta tese que máis contribúe a acelerar a execución de FSVC comparada con SVC. Por este motivo, o capítulo 3 presenta un estudo máis detallado deste algoritmo, denominado ideal kernel tuning (IKT), como método de sintonización do ancho do cerne gausiano para a SVC clásica. Este traballo foi enviado para a súa publicación [15], atópandose neste momento en segunda revisión. A sección 3.1 realiza unha revisión dos métodos existentes na literatura para a selección de modelos na SVM. A idea básica de IKT (sección 3.2) consiste en seleccionar o ancho que minimiza a diferencia media en valor absoluto entre as matrices de cerne ideal e de cerne gausiano, esta última dependente do ancho. Dado que este método baséase nas dúas matrices, ambas con tamano definido polo número de datos, a súa formulación para a SVC orixinal require o uso de prototipos que permitan a súa execución sobre datos grandes. Polo tanto, a sintonización eficiente débese executar conxuntamente co cálculo eficiente do cerne descrito no capítulo 2, é decir, xunto co uso de prototipos. Nalgúns datos, a diferencia mínima entre as matrices de cerne ideal e cerne gausiano correspóndese cos valores mínimo ou máximo do ancho da gausiana. Nestes casos o acerto máximo non se acada na diferencia mínima, senón na máxima derivada da diferencia, de modo que o ancho selecciónase como o que maximiza a derivada desta diferencia. Na clasificación multi-clase, todas as SVCs binarias usan o mesmo valor do ancho, seleccionado considerando as dúas clases máis poboadas.

O traballo experimental (sección 3.3) compara sobre un conxunto de 37 problemas de clasificación de ata 1 millón de patróns o método proposto IKT con outras 5 técnicas competidoras na literatura, todos eles usados para seleccionar o ancho do cerne para entrenar unha SVC clásica. Estas técnicas son o *kernel density estimation* (KDE), que selecciona o ancho óptimo maximizando unha medida de separabilidade entre clases baseada no cerne gausiano; o método estándard *grid-search* (GS), que selecciona o ancho que maximiza o acerto sobre un conxunto separado de datos; a optimización xenética; a optimización mediante *particle-swarm optimization* (PSO); e a optimización bayesiana. Os métodos GS, xenético, PSO e

bayesiano requiren entrenar e testear varias veces a SVC para seleccionar o ancho da gausiana, e polo tanto espérase que sexan máis lentos.

O IKT acada un dos mellores acertos, so lixeiramente por debaixo dos métodos xenético e bayesiano. Ademais, IKT é 105 veces máis rápido que KDE, 163 veces máis rápido que GS, e miles de veces máis rápido que o xenético, PSO e bayesiano. Esta diferencia en velocidade aumenta co tamano dos datos. De feito, os métodos xenético e PSO non poden ser executados nos 6 datos máis grandes da colección. Considerando o consumo de memoria RAM, o método KDE, que é o segundo máis rápido logo de IKT, require 400 veces máis memoria que IKT, mentres que os outros métodos requiren memorias similares a IKT. De feito, IKT non require máis de 10MB en ningún caso, mentres KDE chega a usar ata 25GB nos datos máis grandes. Globalmente, IKT acada o mesmo acerto pero sendo moitísimo máis rápido que as alternativas existentes e mantendo un baixo consumo de memoria. Polo tanto, é unha opción moi interesante para datos de calquera tipo, pequenos, medianos ou grandes, aínda que neste último caso, unha vez seleccionado o ancho óptimo, a SVC clásica non podería ser executada por usar demasiados datos de entrenamento.

O traballo futuro inclúe extender a formulación de FSVC a problemas de regresión; deseñar aproximacións máis eficientes que OVO e OVA para datos grandes con moitas clases; definir un entrenamento eficiente que acede mellores taxas de acerto sen perder velocidade de execución; e empregar a información proporcionada pola matriz de cerne para mellorar a eficiencia da SVC.

Palabras chave: aprendizaxe automática, clasificación, máquina de vectores de soporte, datos grandes, sintonización de hiper-parámetros, cerne gausiano, dimensionalidade elevada, clasificación multi-clase.

CHAPTER 1

THE SUPPORT VECTOR MACHINE

The current thesis belongs to the field of machine learning, a field of artificial intelligence that is classical but has recently achieved great interest from the research community. Machine learning algorithms or models are oriented to extract information from data, in two main paradigms. The supervised methods use data in order to predict some value (named often as "output") using other values (named usually as "inputs"). The value to be predicted is used as a label in order to adjust the method (during a stage named "training") in such a way that it reproduces the relation between inputs and output. This label is what provides "supervision". The unsupervised methods extract information from the available data without any kind of label, in order to create groups of data that are similar in some sense, as in the "clustering" methods or to reduce the number of inputs (data dimensionality) by removing redundant or unuseful information. There are also additional machine learning paradigms, such as reinforcement learning, where the learning machine is rewarded or punished depending on the quality of its predictions without supervised labels, or semi-supervised learning, that combines unlabeled and a small amount of labeled data in order to refine the quality of unsupervised learning in scenarios where labels are very expensive to achieve.

Within the supervised paradigm, classification is one of the kinds of problems most studied in the literature. The objective of classification is to assign a discrete (class) label as output to a vector of input data, that is used by the machine to predict the class label, expectably with a reasonable accuracy or reliability as if it were an oracle. The other relevant kind of problems in machine learning is commonly known as regression, that is conceptually similar to classification but the output to be predicted is a continue numeric value instead a discrete class label.

Indeed, both classification and regression can be considered as the same problem: to predict the values of an unknown function (output) using other data (inputs). This prediction requires to learn (i.e., to calculate a value for) a set of trainable parameters from a collection of examples, each composed by the the value of the unknown function and by the corresponding input data. During training, the set of parameters are adjusted to fit the predicted function to the collection of examples. This collection is expected to be representative enough of the function to be learnt, in order to allow a good accuracy in the prediction issued by the machine learning model. When the value to be predicted is a discrete class label, classification is achieved. From this point of view, regression can be considered more general than classification or including classification as a specific case. However, classification is much more popular in the practice than regression, so in general many machine learning techniques are more oriented to classification than to regression, while others are equally used for both tasks. There is a wide variety of well-known machine algorithms that are able to learn to predict a function, either continuous or discrete, such as k-nearest neighbors, linear and logistic regression, decision trees, neural networks and ensembles of different families, such as bagging, boosting and random forests, among others.

In the current thesis we will focus on one of the classifiers that achieve state-of-the-art performance: the so called support vector machine (SVM). This algorithm was initially proposed in 1960's, but it achieved a wide popularity in the middle 1990's, when Corinna Cortes and Vladimir Vapnik [5] formulated it for general machine learning. Since then, the SVM became widely used both for classification (support vector classifier, SVC) and regression (support vector regression, SVR), due to its high performance, specially using radial basis kernels, that make it a state-of-the-art machine learning method over a wide variety of applications [1] because: 1) it optimizes a convex function without local extremes that may drive training into sub-optimal solutions, as happens with some neural networks; and 2) it is easy of use and configure, mainly with respect to the hyper-parameter tuning, because a standard collection of hyper-parameter values is valid for most applications [20].

1.1 Hard-margin SVM

In the following, the notation defined by [40] and listed in Table 1.1, will be used. Let $\{\mathbf{x}_n\}_{n=1}^{N}$ be the set of training data (henceforth named as patterns). Vectors are in lowercase bold font, as columns, while matrices are designed with bold uppercase symbols. Let N be the number

of training patterns. Let $\mathbf{x}_n = (x_{n1}, \ldots, x_{nI})$ be the n-th training pattern, where I is the number of inputs, or the dimensionality of the input pattern. In the following, we will consider a two-class (or binary) SVC, so let $Q = 2$ be the number of classes. Let $c_n \in \{1,2\}$ be the class label of $\mathbf{x}_n$. We also define $y_n = -1$ (resp. $y_n = 1$) for $\mathbf{x}_n$ with $c_n = 1$ (resp. $c_n = 2$).

Symbol	Meaning	Symbol	Meaning
N	Number of training patterns	I	Number of inputs
$\mathbf{x}_n$	n-th traning pattern	x_{ni}	i-th input of $\mathbf{x}_n$
Q	No. of classes	c_n	Class label $\in \{1 \ldots Q\}$ of $\mathbf{x}_n$
y_n	True output $\in \{\pm 1\}$ for $\mathbf{x}_n$	$\mathcal{H}$	SVC hyperplane
$\mathbf{w}$	Vector that defines $\mathcal{H}$	b	Offset of $\mathcal{H}$
$\mathbf{x}$	Test pattern	T	Number of test patterns
ρ	Margin of $\mathcal{H}$	H_n	Function for $\mathcal{H}$: $H_n = \mathbf{w}^T\mathbf{x}_n + b$
$\mathcal{L}$	Lagrange function	α_n	Hard Lagrange multiplier for $\mathbf{x}_n$
$\mathcal{V}$	Set of indices of support vectors	C	Regularization hyper-parameter
ξ_n	Slack variable	β_n	Soft Lagrange multiplier for $\mathbf{x}_n$
σ	RBF kernel spread	ε	With of insensitive loss function
ξ_n, ξ_n^*	Upper,lower slack variables	α_n, α_n^*	Upper, lower Lagrange multipliers
P	No. training and test patterns		

Table 1.1: Symbols used in the text and meaning of each one.

Let us consider a linearly separable classification problem, so that a linear classification function is able to discriminate correctly the samples (or training patterns) of both classes without errors. In this case, the SVC is named "hard-margin classifier". A linear classifier is defined by a hyperplane, named as $\mathcal{H}$, whose equation is $H(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$, where $\mathbf{w}$ is a column vector perpendicular to $\mathcal{H}$,while $\mathbf{w}^T$ is the transposed of $\mathbf{w}$ (row vector), $\mathbf{x}$ is the column vector where H is calculated and b is the hyperplane offset, so that $b = H(\mathbf{0})$. The output $y(\mathbf{x})$ of the binary SVC, that is a linear classifier, for a pattern $\mathbf{x}$ is given by:

$$y(\mathbf{x}) = \text{sign}[H(\mathbf{x})] = \text{sign}(\mathbf{w}^T\mathbf{x} + b) \tag{1.1}$$

where $\text{sign}(t) = 1$ when $t \geq 0$ and $\text{sign}(t) = -1$ otherwise, so the classifier output is $+1$ on one side of $\mathcal{H}$ and output -1 on the other side. Note that $\mathbf{w}$ and b are the trainable parameters of the SVC. The distance ϕ_n between $\mathcal{H}$ and a training pattern $\mathbf{x}_n$ is given by:

$$\phi_n = \frac{|H_n|}{|\mathbf{w}|} = \frac{|\mathbf{w}^T\mathbf{x}_n + b|}{|\mathbf{w}|} \tag{1.2}$$

where $H_n = H(\mathbf{x}_n)$, while $|H_n|$ is the value absolute of H_n and $|\mathbf{w}|$ is the Euclidean norm of vector $\mathbf{w}$. According to the SVM theory, the SVC is defined as the linear classifier defined by the hyperplane $\mathcal{H}^*$ that maximizes its minimum distance to the training patterns. This minimum distance will be henceforth named as the "margin" of the hyperplane, being denoted as $\rho(\mathbf{w},b)$. Since the SVC maximizes this margin, the optimal values $\{\mathbf{w}^*,b^*\}$ for the trainable parameters of the SVC can be calculated as:

$$\mathbf{w}^*,b^* = \arg\max_{\mathbf{w},b} \{\rho(\mathbf{w},b)\}, \quad \rho(\mathbf{w},b) = \min_{n=1,\ldots,N} \left\{ \frac{|H_n|}{|\mathbf{w}|} \right\} \tag{1.3}$$

The equation $H(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$ of $\mathcal{H}$ is not unique because multiplying it by any scalar keeps valid as the equation of the same hyperplane. Since the dataset is linearly separable, it is always possible to select H, or equivalently $\mathbf{w}$ and b, in such a way that its value H_r for the training pattern $\mathbf{x}_r$ nearest to $\mathcal{H}$ is $H_r = \pm 1$ (the sign depends on the side of the hyperplane where $\mathbf{x}_r$ is located). Besides, $|H_n| \geq 1$ for all $\mathbf{x}_n$. The distance ϕ_r between $\mathcal{H}$ and $\mathbf{x}_r$ is given by:

$$\phi_r = \frac{|H_r|}{|\mathbf{w}|} = \frac{1}{|\mathbf{w}|} \tag{1.4}$$

In order to maximize this distance, $|\mathbf{w}|$ must be minimized while guaranteeing that all the training patterns are correctly classified with distance to the hyperplane greater than 1. Thus, it must be $H_n \geq 1$ when $y_n = +1$ and $H_n \leq -1$ for $y_n = -1$, so that $y_n H_n \geq 1$ for $n = 1,\ldots,N$. In other words, it is not enough that all the training patterns are correctly classified, so that $y_n H_n > 0$. On the contrary, since all the training patterns must be outside the margin in order to be far from being misclassified, they must verify $y_n H_n \geq 1$. The objective is to select the farthest hyperplane to the training patterns of both classes, among the different hyperplanes that discriminate correctly between both, with the hope that this hyperplane will classify optimally new patterns of both classes that have not been included in the training set. The maximization of the distance between hyperplane and training patterns (margin) pursues to minimize the so called "structural risk", a concept from the statistical learning theory [46] that refers to the risk of classifier overtraining. The hyperplane with the highest margin has the lowest expected risk of overtraining, i.e., misclassification of new, unseen test patterns. Alternatively, this hyperplane has the highest ability to generalize the classification correctly for new patterns not used during training. Thus, the SVM-based classifiers are often referred

as "margin maximization" methods. The optimization problem for the SVC in the hard margin case can be stated as:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} |\mathbf{w}| \quad \text{s.t.} \quad \{y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1 \geq 0\}_{n=1}^N \tag{1.5}$$

where s.t. means "subject to". This optimization problem with linear inequality constraints can be performed using the Lagrange multipliers method, where the Lagrange function $\mathcal{L}$ to be minimized is defined as:

$$\mathcal{L}(\mathbf{w}, b, \vec{\alpha}) = \frac{|\mathbf{w}|^2}{2} - \sum_{n=1}^{N} \alpha_n [y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1] \tag{1.6}$$

Here, $\{\alpha_n\}_{n=1}^N$ are the Lagrange multipliers and $\vec{\alpha} = (\alpha_1, \ldots, \alpha_N)$. Note that minimize $|\mathbf{w}|$ is equivalent to minimize $|\mathbf{w}|^2/2$. This is the so called "primal" optimization problem. Equaling to zero the derivatives of $\mathcal{L}$ with respect to $\mathbf{w}$ and b, we achieve:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n = \mathbf{0} \rightarrow \mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n \tag{1.7}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{n=1}^{N} \alpha_n y_n = 0 \tag{1.8}$$

The optimization theory [30] states that the solution of this problem is a saddle point of the objective function defined by the Karush-Kuhn-Tucker (KKT) conditions:

$$\alpha_n [y_n(\mathbf{w}^T\mathbf{x}_n + b) - 1] = 0, \quad n = 1, \ldots, N \tag{1.9}$$

Note that vector $\mathbf{w}$ defining hyperplane $\mathcal{H}$ is a linear combination of the training patterns defined by the multipliers $\{\alpha_n\}_{n=1}^N$, that verify eqs. 1.8 and 1.9. Substituting $\mathbf{w}$ from eq. 1.7 in the definition of $\mathcal{L}$, we achieve:

$$\mathcal{L}(\vec{\alpha}) = \frac{1}{2}\left(\sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_m\right)^T \left(\sum_{m=1}^{N} \alpha_n y_n \mathbf{x}_n\right) - \sum_{n=1}^{N} \alpha_n y_n \left(\sum_{m=1}^{N} \alpha_m y_m \mathbf{x}_m\right)^T \mathbf{x}_n - b\sum_{n=1}^{N} \alpha_n y_n +$$
$$+ \sum_{n=1}^{N} \alpha_n = \frac{1}{2}\sum_{mn=1}^{N} \alpha_m \alpha_n y_m y_n \mathbf{x}_m^T \mathbf{x}_n - \sum_{mn=1}^{N} \alpha_m \alpha_n y_m y_n \mathbf{x}_m^T \mathbf{x}_n - b\sum_{n=1}^{N} \alpha_n y_n + \sum_{n=1}^{N} \alpha_n$$

where $\sum_{mn=1}^N$ means sum over m and n running from 1 to N. Using eq. 1.8, the third term in the second line of the previous equation vanishes and we achieve the so called "dual" optimization problem:

$$\begin{aligned} \text{maximize} \quad \mathcal{L}(\vec{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{mn=1}^{N} \alpha_m \alpha_n y_m y_n \mathbf{x}_m^T \mathbf{x}_n = \vec{\alpha}^T \mathbf{1} - \frac{\vec{\alpha}^T \mathbf{K} \vec{\alpha}}{2} \\ \text{s.t.} \quad \sum_{n=1}^{N} \alpha_n y_n = 0, \quad \alpha_n \geq 0, n = 1, \ldots, N \end{aligned} \tag{1.10}$$

The matrix $\mathbf{K}$ is the kernel matrix, in this case a linear kernel, of order $N \times N$, defined as $\mathbf{K} = (k_{nm})_{nm=1}^{N}$ with $k_{nm} = y_n y_m \mathbf{x}_n^T \mathbf{x}_m$. Vector $\mathbf{1}$ is the N-dimensional column vector with ones. The dual optimization problem is solved using quadratic interative programing methods that calculate $\{\alpha_n\}_{n=1}^{N}$, a sparse solution because $\alpha_n \neq 0$ only for $N' \leq N$ training patterns, named support vectors. Once $\vec{\alpha}$ is calculated, the weight vector $\mathbf{w}$ is given by:

$$\mathbf{w} = \sum_{n \in \mathcal{V}} \alpha_n y_n \mathbf{x}_n \tag{1.11}$$

being $\mathcal{V}$ the set of N' support vectors. In order to calculate the offset b, note that any support vector $\mathbf{x}_n$ verifies:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1 \tag{1.12}$$

This equation means that when $y_n = +1$, it must be $\mathbf{w}^T \mathbf{x}_n + b = 1$, and when $y_n = -1$, it must be $\mathbf{w}^T \mathbf{x}_n + b = -1$. Therefore, it must be $\mathbf{w}^T \mathbf{x}_n + b = y_n$, and the offset b can be calculated as:

$$b = y_n - \mathbf{w}^T \mathbf{x}_n \tag{1.13}$$

being $\mathbf{x}_n$ any support vector. Finally, the SVC output for a test pattern $\mathbf{x}$ can be written as:

$$y(\mathbf{x}) = \text{sign}\left(\sum_{n \in \mathcal{V}} \alpha_n y_n \mathbf{x}_n^T \mathbf{x} + b \right) \tag{1.14}$$

Note that this output can be re-written as:

$$y(\mathbf{x}) = \text{sign}\left[\left(\sum_{n \in \mathcal{V}} \alpha_n y_n \mathbf{x}_n \right)^T \mathbf{x} + b \right] = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \tag{1.15}$$

so the SVC output is very efficient to calculate, because only $\mathbf{w} = \sum_{n \in \mathcal{V}} \alpha_n y_n \mathbf{x}_n$ and $b = y_n - \mathbf{w}^T \mathbf{x}_n$ for any $n \in \mathcal{V}$ must be calculated just once, and not for each test vector $\mathbf{x}$. This stands only when linear kernels are used, as we will see in section 1.3.

1.2 Soft-margin case

When the dataset is not linearly separable, we achieve the "soft-margin SVC". In this case, the slack variable ξ_n is defined in order to measure, using the so called hinge loss function, the error of the SVC on the training pattern $\mathbf{x}_n$:

$$\xi_n = \max[0, 1 - y_n H_n] = \max[0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)] \tag{1.16}$$

The slack ξ_n is zero or $1 - y_n H_n$, if this value is positive. Note that ξ_n depends on y_n, $\mathbf{w}$, $\mathbf{x}_n$ and b. When $\mathbf{x}_n$ is correctly classified with $y_n H_n \geq 1$, i.e., outside the margin, we have $1 - y_n H_n \leq 0$ and $\xi_n = 0$. When $\mathbf{x}_n$ is correctly classified but $0 \leq y_n H_n < 1$, i.e., it is inside the margin and near to be misclassified, we have: 1) since $y_n H_n < 1$, it follows that $1 - y_n H_n > 0$, so that by eq. 1.16 it is $\xi_n > 0$; and 2) since $0 \leq y_n H_n$, we achieve $-y_n H_n < 0$ and $\xi_i = 1 - y_n H_n \leq 1$. Thus, merging both inequalities we achieve $0 < \xi_n \leq 1$. Finally, when $\mathbf{x}_n$ is missclassified, $y_n H_n < 0 \rightarrow -y_n H_n > 0 \rightarrow 1 - y_n H_n > 1$ and by eq. 1.16 we have $\xi_n = 1 - y_n H_n > 1$. The three cases are compiled in Table 1.2.

ξ_n	Meaning
0	$\mathbf{x}_n$ well classified outside margin (OK)
$0 < \xi_n \leq 1$	$\mathbf{x}_n$ well classified within margin (OK, but near to be wrong)
$1 < \xi_n$	$\mathbf{x}_n$ misclassified (wrong)

Table 1.2: Values of ξ_n and meaning.

Since we are in the soft-margin case, the dataset is not linearly separable, so the goal of the optimization problem is again to minimize $|\mathbf{w}|$ in order to maximize the margin, but in this case the sum of slack variables $\{\xi_n\}_{n=1}^N$, that evaluates the classification error over the training set (empirical risk), must also be minimized. In order to weight relatively the training error and the weight norm, the so called "regularization parameter", often denoted as C, is introduced as weight of the second term. Besides, the linear inequality constraints must be modified in order to take into account the slack variables, and it must be $y_n H_n - 1 \geq -\xi_n$. This inequality can be written as $\xi_n \geq 1 - y_n H_n$, that is the exact meaning of the definition of ξ_n in eq. 1.16. Therefore, we have the following optimization problem:

$$\text{minimize} \quad \frac{|\mathbf{w}|^2}{2} + C \sum_{n=1}^{N} \xi_n \quad \text{s.t.} \quad \{y_n(\mathbf{w}^T \mathbf{x}_n + b) - 1 \geq -\xi_n, \xi_n \geq 0\}_{n=1}^N \tag{1.17}$$

The Lagrange function is given by:

$$\mathcal{L}(\mathbf{w},b,\vec{\alpha},\vec{\xi},\vec{\beta}) = \frac{|\mathbf{w}|^2}{2} + C\sum_{n=1}^{N}\xi_n - \sum_{n=1}^{N}\alpha_n\{y_n(\mathbf{w}^T\mathbf{x}_n+b)-1+\xi_n\} - \sum_{n=1}^{N}\beta_n\xi_n \quad (1.18)$$

where α_n and β_n are the Lagrange multipliers associated to the constraints in eq. 1.17. Deriving $\mathcal{L}$ with respect to $\mathbf{w}$ and b:

$$\frac{\partial\mathcal{L}}{\partial\mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N}\alpha_n y_n\mathbf{x}_n = \mathbf{0} \rightarrow \mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n\mathbf{x}_n \quad (1.19)$$

The derivative of $\mathcal{L}$ with respect to b is:

$$\frac{\partial\mathcal{L}}{\partial b} = -\sum_{n=1}^{N}\alpha_n y_n = 0 \quad (1.20)$$

The derivative of $\mathcal{L}$ with respect to ξ_n is:

$$\frac{\partial\mathcal{L}}{\partial\xi_n} = C - \alpha_n - \beta_n = 0, \quad n = 1,\ldots,N \quad (1.21)$$

Replacing $\mathbf{w}$ in $\mathcal{L}$, we achieve:

$$\mathcal{L}(\vec{\alpha},\vec{\beta}) = \frac{1}{2}\left(\sum_{n=1}^{N}\alpha_n y_n\mathbf{x}_m\right)^T\left(\sum_{m=1}^{N}\alpha_n y_n\mathbf{x}_n\right) + C\sum_{n=1}^{N}\xi_n -$$
$$-\sum_{n=1}^{N}\alpha_n y_n\left(\sum_{m=1}^{N}\alpha_m y_m\mathbf{x}_m\right)^T\mathbf{x}_n - b\sum_{n=1}^{N}\alpha_n y_n - \sum_{n=1}^{N}\alpha_n(\xi_n-1) - \sum_{n=1}^{N}\beta_n\xi_n \quad (1.22)$$

Using eq. 1.20 and 1.21 in the previous equation, several terms vanish and we achieve:

$$\mathcal{L}(\vec{\alpha},\vec{\beta}) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{mn=1}^{N}\alpha_m\alpha_n y_m y_n\mathbf{x}_m^T\mathbf{x}_n = \vec{\alpha}^T\mathbf{1} - \frac{\vec{\alpha}^T\mathbf{K}\vec{\alpha}}{2} \quad (1.23)$$

that is the same as in the linearly separable case, with the same kernel matrix $\mathbf{K}$, but now the constraints are:

$$\sum_{n=1}^{N}\alpha_n y_n = 0, \quad 0 \leq \alpha_n \leq C, \quad n = 1,\ldots,N \quad (1.24)$$

and the KKT conditions are the following:

$$\alpha_n[y_n(\mathbf{w}^T\mathbf{x}_n+b)-1+\xi_n] = 0, \quad \beta_n\xi_n = 0, \quad n = 1,\ldots,N \quad (1.25)$$

Combining eqs. 1.25 (right) and 1.21, we achieve $\xi_n = 0$ when $\alpha_n < C$. Again, the values of α_n and β_n are calculated by quadratic optimization following a numerical iterative procedure. Once calculated $\{\alpha_n\}_{n\in\mathcal{V}}$, where $\mathcal{V}$ is the set of indices of the support vectors with $\alpha_n > 0$, the weight vector $\mathbf{w}$ and the offset b can be calculated similarly to the hard-margin case (eqs. 1.11 and 1.13, respectively).

1.3 Non-linear kernel

Since the SVC is a linear classifier, its ability is limited for highly non-linear classification problems, that are the real-life cases. The way to increase this ability is to use a non-linear kernel in order to map or project a dataset into a space with a higher dimensionality, where by the Cover's theorem [6] the probability of being linearly separable is higher than in the original low-dimensional space. A kernel function $K(\mathbf{x},\mathbf{y})$ can be written as $K(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})^T\Phi(\mathbf{y})$, where $\Phi(\mathbf{x})$ is a projection or kernel mapping from the input space of dimension I to a high dimensionality space $\mathbb{R}^d$ with $d > I$. The kernel function $K(\mathbf{x},\mathbf{y})$ can be considered as a generalized scalar product, or a generalized similarity measurement, in the high-dimensional space, named "hidden" or "feature' space. There are several kernel functions, such as the Gaussian or radial basis function (RBF, eq. 1.26), polynomial (eq. 1.27), linear (eq. 1.28) and hyperbolic tangent (eq. 1.29).

$$K(\mathbf{x},\mathbf{y}) = \exp\left(-\frac{|\mathbf{x}-\mathbf{y}|^2}{2\sigma^2}\right) \tag{1.26}$$

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}^T\mathbf{y} + a)^b \tag{1.27}$$

$$K(\mathbf{x},\mathbf{y}) = \mathbf{x}^T\mathbf{y} \tag{1.28}$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(\mathbf{x}^T\mathbf{y} + a) \tag{1.29}$$

where σ is the RBF kernel spread, while a and b are the hyper-parameters of the remaining kernels, excepting the linear kernel that has no hyper-parameter. The use of kernels does not change very much the formulation of the soft-margin SVC, excepting that the dot products $\mathbf{x}_m^T\mathbf{x}_n$ are now calculated on the hidden space, so they are replaced by the kernel generalized dot product $\Phi(\mathbf{x}_m)^T\Phi(\mathbf{x}_n) = K(\mathbf{x}_m,\mathbf{x}_n)$. This allows to use the kernels implicitely, because no explicit mapping from the input to the feature space is required and no weight vector $\mathbf{w}$ must be calculated in the feature space. This is often called the "kernel trick" in the SVM literature. The Lagrange function in the hidden space is:

$$\mathcal{L}(\mathbf{w},b,\vec{\xi},\vec{\alpha},\vec{\beta}) = \frac{|\mathbf{w}|^2}{2} + C\sum_{n=1}^{N}\xi_n - \sum_{n=1}^{N}\beta_n\xi_n - \sum_{n=1}^{N}\alpha_n\left[y_n(\mathbf{w}^T\Phi(\mathbf{x}_n)+b) - 1 + \xi_n\right] \quad (1.30)$$

Derivating $\mathcal{L}$ with respect to $\mathbf{w}$ and equaling to zero gives:

$$\mathbf{w} = \sum_{n=1}^{N}\alpha_n y_n \Phi(\mathbf{x}_n) \quad (1.31)$$

Derivating $\mathcal{L}$ with respect to $b, \vec{\xi}, \vec{\alpha}$ and $\vec{\beta}$, we achieve the dual optimization problem, i.e., find $\{\vec{\alpha}, b\}$:

$$\begin{aligned} &\text{maximize} \quad \mathcal{L}(\vec{\alpha}) = \vec{\alpha}^T\mathbf{1} - \frac{\vec{\alpha}^T\mathbf{K}\vec{\alpha}}{2}, \quad \text{s.t.} \quad \vec{\alpha}^T\mathbf{y} = 0, \quad 0 \le \alpha_n \le C \\ &\beta_n\xi_n = 0, \quad \alpha_n\left\{y_n\left[\sum_{m=1}^{N}\alpha_m K(\mathbf{x}_n,\mathbf{x}_m) + b\right] - 1 + \xi_n\right\} = 0, \quad n = 1,\ldots,N \end{aligned} \quad (1.32)$$

Now, the kernel matrix $\mathbf{K}$, of size $N \times N$ as in the previous sections, has elements defined as $K_{nm} = K(\mathbf{x}_n,\mathbf{x}_m)$. Using a non-linear kernel, the direct calculation of the weight vector $\mathbf{w}$ is not possible because the space where $\mathbf{w}$ resides is hidden or implicit. Therefore, just the Lagrange multipliers α_n must be calculated using an optimization method. As in the hard and soft margin case, $\alpha_n = 0$ except when $\mathbf{x}_n$ is a support vector. Once $\vec{\alpha}$ is calculated, $\mathbf{w}$ is given by eq. 1.31 and the SVC output is:

$$\begin{aligned} y(\mathbf{x}) = \text{sign}\left[\mathbf{w}^T\Phi(\mathbf{x}) + b\right] &= \text{sign}\left[\sum_{n\in\mathcal{V}}\alpha_n y_n \Phi(\mathbf{x}_n)^T\Phi(\mathbf{x}) + b\right] = \\ &= \text{sign}\left[\sum_{n\in\mathcal{V}}\alpha_n y_n K(\mathbf{x}_n,\mathbf{x}) + b\right] \end{aligned} \quad (1.33)$$

where the kernel trick $\Phi(\mathbf{x}_n)^T\Phi(\mathbf{x}) = K(\mathbf{x}_n,\mathbf{x})$ was used again. The calculation of the weight vector $\mathbf{w}$ in the feature space is replaced by the calculation of the kernel function K, given by some of the expressions in eqs. 1.26-1.29, for the support vectors $\{\mathbf{x}_n\}_{n\in\mathcal{V}}$ and the test pattern $\mathbf{x}$.

1.4 Alternative SVM solvers

Some other methods exist for training the SVC, including sub-gradient descent, that pursues to minimize directly:

$$J(\mathbf{w},b) = \frac{|\mathbf{w}|^2}{2} + C\sum_{n=1}^{N} \max[0, 1 - y_n(\mathbf{w}^T\mathbf{x}_n + b)] \tag{1.34}$$

An advantage of this method is that the number of required iterations does not scale very fast with N. A solver of this type is the primal estimated sub-gradient solver for SVM, named as Pegasos-SVM [39]. An alternative solver method is coordinate descent [19], that maximizes the dual form:

$$J(\vec{\alpha}) = \vec{\alpha}^T\mathbf{1} - \frac{\vec{\alpha}^T\mathbf{K}\vec{\alpha}}{2} \quad \text{s.t.} \quad \vec{\alpha}^T\mathbf{y} = 0, \quad 0 \leq \alpha_n \leq C, \quad n = 1,\ldots,N \tag{1.35}$$

This method adjusts iteratively α_n in the direction of $\frac{\partial J(\vec{\alpha})}{\partial \alpha_n}$ for $n = 1,\ldots,N$, and projects $\vec{\alpha}$ on its nearest $\vec{\alpha}_0$ that verifies the constraints. The process iterates until an optimal $\vec{\alpha}$ is achieved.

1.5 Multi-class classification

When the classification problem has $Q > 2$ classes, it is reduced to several binary classification problems. The most common strategies are "one-vs-one" (OVO) and "one-vs-all" (OVA). The former uses a different binary SVC to discriminate between each pair of classes. This strategy requires $Q(Q-1)/2$ binary SVCs, each trained using only the training patterns of its pair of classes. The predicted class label is decided by voting among the binary SVCs. On the contrary, the OVA approach uses only Q binary SVS, each discriminating between a class and the remaining ones, being trained with the whole training set. The OVO method usually achieves better performance, although for high Q the number of required binary SVCs, that raises with Q^2, may become inefficient compared to OVA, that often achieves lower performance but only requires Q binary SVCs. Alternative approaches include directed acyclic graph SVC [34], that also uses $Q(Q-1)/2$ binary SVCs, error correcting output codes [9] and direct multi-class optimization [7].

1.6 Support vector regression

Although in this thesis we will center on classification, the SVM was also extended for regression problems. This was done by transposing the concept of support vector (a training pattern that is inside the hyperplane margin) to vectors that lie inside a margin around the value of the function (output) to be predicted. The support vector regression (SVR) only considers those

training patterns for what the SVR output is farther from the true output than a threshold ε. This is implemented by the so-called ε-insensitive cost function, defined as:

$$f(t,\varepsilon) = \begin{cases} t & |t| \leq \varepsilon \\ 0 & |t| > \varepsilon \end{cases}$$

leading to the popular ε-SVR. Considering a kernel mapping $\Phi(\mathbf{x})$, the ε-SVR is a linear regressor in the feature space defined by:

$$y(\mathbf{x}) = \mathbf{w}^T \Phi(\mathbf{x}) + b \tag{1.36}$$

The slack variables ξ_n and ξ_n^* define the upper and lower margins, respectively, with respect to the true output y_n:

$$\xi_n = \max[0, (\mathbf{w}^T \Phi(\mathbf{x}) + b) - y_n - \varepsilon] \tag{1.37}$$

$$\xi_n^* = \max[0, y_n - (\mathbf{w}^T \Phi(\mathbf{x}) + b) - \varepsilon] \tag{1.38}$$

The primal optimizacion problem is the following:

$$\text{minimize} \quad \frac{|\mathbf{w}|^2}{2} + C \sum_{n=1}^{N} (\xi_n + \xi_n^*) \tag{1.39}$$

$$\text{s.t.} \quad \mathbf{w}^T \Phi(\mathbf{x}) + b - y_n \leq \varepsilon + \xi_n \tag{1.40}$$

$$y_n - \mathbf{w}^T \Phi(\mathbf{x}) - b \leq \varepsilon + \xi_n^* \tag{1.41}$$

$$\xi_n, \xi_n^* \geq 0, \quad n = 1, \ldots, N \tag{1.42}$$

Following a process analogous to the soft-margin classifier, we achieve the dual problem, defined by:

$$\text{minimize} \quad \frac{\Delta\vec{\alpha}^T \mathbf{K} \Delta\vec{\alpha}}{2} + \varepsilon \sum_{n=1}^{N} (\alpha_n + \alpha_n^*) + \sum_{n=1}^{N} y_n (\alpha_n - \alpha_n^*) \tag{1.43}$$

$$\text{s.t.} \quad \vec{\varepsilon}^T \Delta\vec{\alpha} = 0, \quad 0 \leq \alpha_n, \alpha_n^* \leq C, \quad n = 1, \ldots, N \tag{1.44}$$

where $\Delta\vec{\alpha} = \vec{\alpha} - \vec{\alpha}^*$ and $\mathbf{K}$ is the kernel matrix with elements $\{K(\mathbf{x}_m, \mathbf{x}_n)\}_{mn=1}^N$. Once $\vec{\alpha}$ and $\vec{\alpha}^*$ are calculated, the output is given by:

$$y(\mathbf{x}) = \sum_{n \in \mathcal{V}} \Delta\alpha_n K(\mathbf{x}_n, \mathbf{x}) + b \tag{1.45}$$

where $\Delta\alpha_n = \alpha_n - \alpha_n^*$ and the offset b is calculated as:

$$b = \begin{cases} y_n - \sum_{m \in \mathcal{V}} \alpha_m y_m K(\mathbf{x}_m, \mathbf{x}_n) - \varepsilon & \alpha_n < C \\ y_n - \sum_{m \in \mathcal{V}} \alpha_m y_m K(\mathbf{x}_m, \mathbf{x}_n) + \varepsilon & \alpha_n^* < C \end{cases}$$

being $\mathbf{x}_n$ any support vector.

1.7 Objectives of this thesis

The current thesis is focused on the support vector machine with radial basis kernel for large-scale classification problems. The objective of the work is to formulate a SVC-based classification method efficient enough to be executed on large datasets, either because the number of patterns, inputs or classes is high. In order to propose this method, several limitations of the classical SVC must be addressed. One of the main drawbacks of the SVC is the complexity of its training stage, derived from the need to solve a quadratic optimization problem whose speed and memory requirements strongly depend on the number of training patterns. This drawback hinders the execution of SVC on datasets larger than several tens of thousand patterns, depending on the number of inputs. Related to this issue is the selection of the support vectors, that is a very slow process when the training set is really large. On the other hand, the need to perform hyper-parameter tuning for the regularization and the kernel parameter(s), namely the spread when RBF kernels are used, also requires to execute many times the cycle of training and test over large datasets, and this strongly slows down the SVC execution. These issues will be studied and solutions will be proposed in the chapters 2 and 3 of the current thesis. Specifically, chapter 2 presents our proposal, named fast support vector classifier (FSVC), that is designed to be of low complexity and scalable with the dataset size. This method has been published in [16]. First, the reasons of the high computational cost of SVC for large datasets are analyzed. Then, the FSVC is described in sections 2.1-2.6, describing the differences with respect to SVC, reporting the FSVC pseudo-code and analyzing its computational complexity. Sections 2.7-2.18 report the results in terms of performance, time and memory requirements, discuss the experimental work and compare to other existing methods, including alternative SVM solvers, core vector machines and evolutionary traning set selection methods. Chapter 3 extends the efficient method proposed for hyper-parameter tuning, named ideal kernel tuning (IKT), for the classical SVC. This work that has been submitted for publication [15], being currently in second revision. Section 3.1 reviews the literature about

model selection in SVC and section 3.2 describes the proposed algorithm. The results and comparison with other model selection methods in the literature are reported in section 3.3. Finally, chapter 4 presents the conclusions of this research.

CHAPTER 2

FAST SUPPORT VECTOR CLASSIFIER

There are several reasons that justify the low efficiency of SVC with RBF kernels on large datasets:

1. The calculation of the Lagrange multipliers $\{\alpha_n\}_{n\in\mathcal{V}}$ and of the offset b requires to solve a constrained quadratic optimization problem which with a complexity of $\mathcal{O}(N^3)$, i.e., of order N^3, or that raises with N^3, where N is the number of training patterns [23]. On the other hand, the required memory scales as $\mathcal{O}(N^2)$. Besides, the SVC training requires to know simultaneously $\{\mathbf{x}_n\}_{n=1}^{N}$, so the whole training set must be stored in memory during training, which may be difficult with large N and I.

2. After training, only the N' support vectors and coefficients $\{\mathbf{x}_n, \alpha_n\}_{n\in\mathcal{V}}$, and the offset b, must be stored in memory because the calculation of $K(\mathbf{x}_n, \mathbf{x})$ for a test pattern $\mathbf{x}$ in eq. 1.33 requires to evaluate $|\mathbf{x}_n - \mathbf{x}|$ for $n \in \mathcal{V}$, and this requires to store $\{\mathbf{x}_n\}_{n\in\mathcal{V}}$. Although the number N' of support vectors verifies $N' \leq N$, usually N' raises with N, so that even the storage of only the N' support vectors becomes a serious limitation for large N and I.

3. The training process does not provide values for C and σ, so their values must be previously selected, becoming hyper-parameters that must be tuned because their values, specially σ, have a strong influence over the SVC learning ability and test performance. Often, this tuning proceeds by selecting the values that provide the highest performance over separated validation sets ("grid-search" or related approaches), that requires to repeat the training-testing cycle of SVC so many times as the number of combinations

of values of C and σ. This process may be expensive for large datasets. Alternative approaches exist [22, 43, 48], but they require either to iterate the train-test cycle or to perform calculations whose computational cost may also be unpractical for large-scale datasets.

4. With a large number I of inputs, the calculation of differences $|\mathbf{x}_n - \mathbf{x}_m|$ for the RBF kernel $K(\mathbf{x}_n, \mathbf{x}_m)$ in eq. 1.32 requires large memory space for data storing and becomes very time-consuming.

5. When the number Q of classes is large, the OVO approach used by the SVC may be slow because $Q(Q-1)2$ binary SVCs are required, which becomes unpractical even for medium Q values. For instance, when $Q = 10$ the number of required binary SVCs is $Q(Q-1)2 = 45$, so the training must be repeated 45 times. Note that usually the hyper-parameter tuning is performed only once and the selected values are used for all the binary SVCs.

This high computational cost has been identified in the literature as one major drawback of the SVC [23]. There exist alternative methods of selection of training patterns [24, 31, 32], e.g. using evolutionary algorithms [35, 47], or clustering [4] in order to reduce the memory consumption or the number of iterations. However, these approaches do not allow to reduce a very large dataset to a size low enough to train a standard SVC on it. On the other hand, these methods have an intrinsic computational cost that limits its application to medium size datasets. They can not be applied to large datasets because they would not finish never. Other alternatives are core vector machines [44] and other coreset based methods, that compress the data providing a fast solution that approximates the optimal SVM on the whole data. These approaches have been applied in the literature to continuous unbounded data streams. Random features methods [21] such as double stochastic gradient [8] replace the RBF kernel by an explicit mapping to a random Fourier feature space, although the number of features required to achieve a competitive performance may reach about a hundred thousands, so these methods may also be costly in terms of time and memory.

The solutions proposed in the current thesis to these drawbacks are described in sections 2.1 to 2.5. The proposed method, named fast support vector classifier (FSVC), for an efficient large-scale support vector classification [16] is compiled by algorithms 1-4 in section 2.6. The experimental work studies the performance of FSVC and the influence of the training algorithm (section 2.8), efficient kernel (2.9), spread tuning (2.10), large dimensionality (2.11) and

high number of classes (2.12). The FSVC has also been compared to other standard methods (section 2.13), other solvers (2.14) including some of the cited above, and evolutionary train set selection methods (2.17). Finally, the elapsed times and memory use are studied in sections 2.16 and 2.18, respectively.

2.1 Efficient training

The high cost of the iterative numerical optimization process in the classical SVC makes it unpractical for large-scale datasets, so an efficient alternative for SVC should avoid it. The reduced SVM [25] and proximal parametrical SVC [49] use a faster closed-form solution for the training, but they are not practical for large datasets because they require the inversion of large matrices with N^2 elements. This section proposes an efficient training method that is inspired in the SVC ideas but is fast, direct (non-iterative) and has a closed-form analytical expression, without any numerical iterative optimization nor matrix inversion.

2.1.1 One pattern per class

Let us consider a binary classification problem with only $N = 2$ patterns $\mathbf{x}_1$ and $\mathbf{x}_2$, with $c_1 = 1$ and $c_2 = 2$. Let us also consider a hyperplane $\mathcal{H}$ with equation $H(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$ separating both classes in the feature space generated by a nonlinear kernel mapping Φ associated to a RBF kernel function as in eq. 1.26, although any other kernel might be used. The output of this linear classifier for a test pattern $\mathbf{x}$ is $y(\mathbf{x}) = \text{sign}(\mathbf{w}^T\Phi(\mathbf{x}) + b)$, where $b = H(\mathbf{0})$ is the hyperplane offset and $\mathbf{w}$ is a vector in the feature space perpendicular to $\mathcal{H}$ that is chosen directed to the side of $\mathcal{H}$ where $\Phi_2 = \Phi(\mathbf{x}_2)$ with $c_2 = 2$ is placed. The training must find $\{\mathbf{w}, b\}$ that maximizes the distance ϕ_n from Φ_n, with $n = 1, 2$, to $\mathcal{H}$. This distance is given by:

$$\phi_n(\mathbf{w}, b) = \frac{\mathbf{w}^T\Phi_n + b}{|\mathbf{w}|}, \quad n = 1, 2 \tag{2.1}$$

The choice of $\mathbf{w}$ implies that $\phi_2 > 0$ and $\phi_1 < 0$. With only two patterns, the distance between patterns and hyperplane can be maximized selecting $\{\mathbf{w}, b\}$ such that $\phi_1 = -\phi_2$ and therefore:

$$\frac{\mathbf{w}^T\Phi_1 + b}{|\mathbf{w}|} = -\frac{\mathbf{w}^T\Phi_2 + b}{|\mathbf{w}|} \rightarrow b = \frac{-\mathbf{w}^T(\Phi_1 + \Phi_2)}{2} \tag{2.2}$$

Replacing b in eq. 2.1, we have:

$$\phi_n = \frac{\mathbf{w}^T}{|\mathbf{w}|}\left(\Phi_n - \frac{\Phi_1+\Phi_2}{2}\right), \quad n=1,2 \tag{2.3}$$

Since $\mathbf{w}$ points to the side of $\mathcal{H}$ where Φ_2 is located, we must maximize ϕ_2 and simultaneously minimize ϕ_1, so $\mathbf{w}$ must be parallel to:

$$\Phi_2 - \frac{\Phi_1+\Phi_2}{2} = \frac{\Phi_2-\Phi_1}{2} \tag{2.4}$$

Excluding norm factors, $\mathbf{w}$ can be selected as $\mathbf{w} = \Phi_2 - \Phi_1$. Replacing $\mathbf{w}$ in eq. 2.2, using that $\mathbf{w}^T\Phi(\mathbf{x}) = K(\mathbf{x}_2,\mathbf{x}) - K(\mathbf{x}_1,\mathbf{x})$ and replacing $\{\mathbf{w},b\}$ in the SVC output, its expresion remains:

$$y(\mathbf{x}) = \text{sign}(K(\mathbf{x}_2,\mathbf{x}) - K(\mathbf{x}_1,\mathbf{x}) + b), \qquad b = \frac{K(\mathbf{x}_1,\mathbf{x}_1) - K(\mathbf{x}_2,\mathbf{x}_2)}{2} \tag{2.5}$$

2.1.2 Several patterns per class

With $N_1, N_2 > 1$ training patterns for classes 1 and 2, the distance $\phi_n(\mathbf{w},b)$ is given by the same eq. 2.1, but for $n = 1,\ldots,N$, with $N = N_1 + N_2$. In order to calculate $\{\mathbf{w},b\}$, the hyperplane $\mathcal{H}$ in the feature space should be placed half-way between both classes and with the optimal orientation to maximize the distance to patterns of both classes. To find such a hyperplane is a hard computational task that requires an iterative optimization process similar to eq. 1.30, and this is unpractical for large-scale datasets. A simpler alternative, whose performance might still be competitive, is to require that the average distance from the patterns of one class to $\mathcal{H}$ should be equal, and of opposite sign, to the same magnitude for the other class:

$$\sum_{n=1}\frac{\phi_n}{N_1} = -\sum_{n=2}\frac{\phi_n}{N_2} \tag{2.6}$$

where the notation "$n=1$" in the summation means "for all the training patterns $\mathbf{x}_n$ with $c_n = 1$". Substituting ϕ_n from eq. 2.1 to eq. 2.6 and multiplying by $|\mathbf{w}|$, we achieve:

$$\sum_{n=1}\frac{\mathbf{w}^T\Phi_n + b}{N_1} = -\sum_{n=2}\frac{\mathbf{w}^T\Phi_n + b}{N_2} \tag{2.7}$$

$$b = -\sum_{n=1}\frac{\mathbf{w}^T\Phi_n}{2N_1} - \sum_{n=2}\frac{\mathbf{w}^T\Phi_n}{2N_2} \tag{2.8}$$

where $\Phi_n = \Phi(\mathbf{x}_n)$. Analogously to the previous section, $\mathbf{w}$ must be calculated to maximize the sum for both classes of the average distance between patterns and hyperplane. These

distances are positives (resp. negatives) for class 2 (resp. class 1), so that $\mathbf{w}$ must be selected in order to maximize the following amount:

$$\sum_{n=2} \frac{\phi_n(\mathbf{w},b)}{N_2} - \sum_{n=1} \frac{\phi_n(\mathbf{w},b)}{N_1} \tag{2.9}$$

Substituting $\phi_n(\mathbf{w},b)$ from eq. 2.1 for $n = 1,\ldots,N$, we achieve:

$$\begin{aligned} \sum_{n=2} \frac{\mathbf{w}^T\Phi_n + b}{N_2|\mathbf{w}|} - \sum_{n=1} \frac{\mathbf{w}^T\Phi_n + b}{N_1|\mathbf{w}|} = \\ = \frac{\mathbf{w}^T}{|\mathbf{w}|}\left(\sum_{n=2}\frac{\Phi_n}{N_2} - \sum_{n=1}\frac{\Phi_n}{N_1}\right) \end{aligned} \tag{2.10}$$

This expression is maximized when $\mathbf{w}$ is parallel to the vector between parenthesis, so $\mathbf{w}$ can be selected as:

$$\mathbf{w} = \sum_{n=2} \frac{\Phi_n}{N_2} - \sum_{n=1} \frac{\Phi_n}{N_1} \tag{2.11}$$

Replacing $\mathbf{w}$ in eq. 2.8, the offset b is given by:

$$b = \sum_{nm=1} \frac{k_{nm}}{2N_1^2} - \sum_{nm=2} \frac{k_{nm}}{2N_2^2} \tag{2.12}$$

where $k_{nm} = K(\mathbf{x}_n,\mathbf{x}_m) = \Phi_n^T\Phi_m$ and the sum over "$nm = 1$" means "for all $\mathbf{x}_n$ and $\mathbf{x}_m$ with $c_n = 1$ and $c_m = 1$". Substituting $\mathbf{w}$ from eq. 2.11 on the classifier output, we achieve:

$$y(\mathbf{x}) = \text{sign}\left(\sum_{n=2} \frac{k_n(\mathbf{x})}{N_2} - \sum_{n=1} \frac{k_n(\mathbf{x})}{N_1} + b\right) \tag{2.13}$$

where $k_n(\mathbf{x}) = K(\mathbf{x}_n,\mathbf{x}) = \Phi_n^T\Phi(\mathbf{x})$ and b is given by eq. 2.12. This closed-form expression allows the direct calculation of the classifier output and should be faster than the iterative training methods. However, it requires to calculate $\{k_{nm}\}_{nm=1}$ and $\{k_{nm}\}_{nm=2}$ in order to achieve the offset b, and $k_n(\mathbf{x})$ for every test pattern $\mathbf{x}$ and for all the training patterns $\{\mathbf{x}_n\}_{n=1}^N$.

2.2 Efficient kernel calculation

Since no selection of support vectors is done in eq. 2.13, the whole training set $\{\mathbf{x}_n\}_{n=1}^N$ should be stored in memory, which is obviously not affordable with large datasets. Remember that

the standard SVC uses $N' \leq N$ training patterns as support vectors, but their selection requires a complex iterative optimization problem that is not practical for large datasets. The literature provides several alternative approaches for the selection of support vectors [31], but they often require a considerable computational effort which makes them not adequate for large datasets. The FSVC tries to solve this drawback by using a set of pattern prototypes $\{\mathbf{p}_{ql}\}_{l=1}^{L_q}$ for each class $q = 1,2$, with $L_q = \min(N_q, L) \leq L$, being N_q the number of patterns of class q and L a low number[1]. Since the prototypes are created when the patterns are read (usually, from one or several disk files), the FSVC only stores in memory the $\sum_{q=1}^{Q} L_q \leq QL$ prototypes (a number that is upper bounded, as we will see) but no training pattern, saving huge amounts of memory. Specifically, the first L training patterns $\mathbf{x}_n$ of class q (with $c_n = q$) are stored in memory as starting prototypes $\{\mathbf{p}_{ql}\}_{l=1}^{L}$. With $N_q < L$ patterns of class q, only N_q prototypes are created, so that $L_q = N_q$ and no updating is done (in this case, the original training patterns are stored because N_q is low). When $N_q \geq L$, the first L training patterns of class q create the L starting prototypes of this class, and each following pattern $\mathbf{x}_n$ updates its nearest prototype $\mathbf{p}_{qr}$ of the class $q = c_n$ to which $\mathbf{x}_n$ belongs, according to:

$$\mathbf{p}'_{qr} = \left(1 - \frac{1}{N_{qr}}\right)\mathbf{p}_{qr} + \frac{\mathbf{x}_n}{N_{qr}}, \quad q = c_n \tag{2.14}$$

$$r = \arg\min_{l=1\ldots,L} \left|\mathbf{p}_{ql} - \mathbf{x}_n\right|, \quad N'_{qr} = N_{qr} + 1 \tag{2.15}$$

where $\mathbf{p}_{qr}$ and $\mathbf{p}'_{qr}$ are the old and new versions, respectively, of the r-th prototype of class q, which is the nearest to $\mathbf{x}_n$ among the prototypes of class q, while N_{qr} and N'_{qr} are the old and new numbers of training patterns used to update $\mathbf{p}_{qr}$. This updating scheme does not require to decide whether a training pattern $\mathbf{x}_n$ creates a new, or updates an existing, prototype. Although initially the first L patterns of a class (which initializate its prototypes) were similar among them, new patterns will update their nearest prototypes and finally the prototype collection $\{\mathbf{p}_{ql}\}_{l=1}^{L}$ of class q will represent its training patterns. A prototype $\mathbf{p}_{qr}$ calculated from n patterns $\{\mathbf{x}_i\}_{i=1}^{n}$ of class q can be efficiently updated with a batch of m new patterns $\{\mathbf{x}_{n+i}\}_{i=1}^{m}$ according to:

$$\mathbf{p}'_{qr} = \frac{n}{n+m}\mathbf{p}_{qr} + \frac{1}{n+m}\sum_{i=1}^{m}\mathbf{x}_{n+i} \tag{2.16}$$

[1] With $Q > 2$ classes, a set of L_q prototypes will be stored for each class $q = 1,\ldots,Q$.

where r is the index of the prototype nearest to $\{\mathbf{x}_{n+i}\}_{i=1}^{m}$, see lines 2-11 in algorithm 1. Thus, $\mathbf{p}'_{qr}$ in eq. 2.16 is the mean of the $n+m$ patterns for which the r-th prototype was the nearest one:

$$\mathbf{p}'_{qr} = \frac{1}{n+m} \sum_{i=1}^{n+m} \mathbf{x}_i \tag{2.17}$$

Although the differences $|\mathbf{p}_{ql} - \mathbf{x}_n|$ must be calculated for each training pattern $\mathbf{x}_n$ of class q (with $c_n = q$), and the L_q ($\leq L$) prototypes of this class q, the number L is low, so its calculation is efficient for large datasets. Low L values also avoid large kernel matrices (see subsection 2.3 below), that are of size U^2, with $U = L_1 + L_2$, or $U = 2L$ when $L_1 = L_2 = L$. Non-exhaustive experiments reported that $L = 100$ prototypes is enough to represent a class in large-scale datasets (where $N_q > L$ and therefore $L_q = L$), while keeping the kernel matrix inside a reasonable size of $U^2 = (2L)^2 = 40,000$ values. However, large I or Q may require lower L values in order to fit matrices in the available memory. Considering the use of prototypes, the calculation of the kernel values $\{k_n(\mathbf{x})\}_{n=1}^{N}$, $\{k_{nm}\}_{nm=1}$ and $\{k_{nm}\}_{nm=2}$ in the previous section is limited to $U \leq 2L$ prototypes, and eqs. 2.13 and 2.12 above can be re-written as:

$$y(\mathbf{x}) = \text{sign}\left(\sum_{l=1}^{L_2} \frac{k_{2l}(\mathbf{x})}{L_2} - \sum_{l=1}^{L_1} \frac{k_{1l}(\mathbf{x})}{L_1} + b \right) \tag{2.18}$$

$$b = \sum_{lm=1}^{L_1} \frac{k_{1lm}}{2L_1^2} - \sum_{lm=1}^{L_2} \frac{k_{2lm}}{2L_2^2} \tag{2.19}$$

where $k_{ql}(\mathbf{x}) = K(\mathbf{p}_{ql}, \mathbf{x})$ and $k_{qlm} = K(\mathbf{p}_{ql}, \mathbf{p}_{qm})$, with $q = 1,2$, as shown in lines 3-4 in algorithms 3 and 4. The calculation of $y(\mathbf{x})$ only requires $L_1^2 + L_2^2$ values $\{k_{qlm}\}_{q,lm=1}^{2,L_q}$, where $l,m = 1,\ldots,L_q$ and $q = 1,2$, plus U values $\{k_{ql}(\mathbf{x})\}_{ql=1}^{2,L_q}$ for each test pattern $\mathbf{x}$.

2.3 Efficient tuning of the RBF kernel spread

There are several proposals for the spread tuning [14, 28, 26], but they often have a high computational cost and are unpractical for large-scale datasets. Our proposal is based on the concept of ideal kernel function [33], that for a binary classification problem ($Q = 2$), is defined by $K(\mathbf{x},\mathbf{y}) = 1$ when the patterns $\mathbf{x}$ and $\mathbf{y}$ belong to the same class and $K(\mathbf{x},\mathbf{y}) = 0$ otherwise. In our case, patterns $\mathbf{x}$ and $\mathbf{y}$ are replaced by class prototypes $\{\mathbf{p}_{ql}\}_{ql=1}^{2,L_q}$. Let us considering that prototypes $1,\ldots,L_1$ are of class 1, and prototypes $L_1+1,\ldots,U$ are of class

2. The ideal kernel U-order squared matrix $\mathbf{J}$ has elements J_{lm}, with $l,m = 1,\ldots,U$, defined in the following way. When the l-th and m-th prototypes are of the same class, we have $l,m \in \{1,\ldots,L_1\}$ or $l,m \in \{L_1+1,\ldots,U\}$, and in these cases $J_{lm} = 1$. When the l-th and m-th prototypes are of different classes, we have $l \in \{1,\ldots,L_1\}$ and $m \in \{L_1+1,\ldots,U\}$ or $l \in \{L_1+1,\ldots,U\}$ and $m \in \{1,\ldots,L_1\}$, and in these other cases $J_{lm} = 0$. Let the vectors $\{\mathbf{z}_l\}_{l=1}^{U}$ be defined by:

$$\mathbf{z}_l = \begin{cases} \mathbf{p}_{1l} & l = 1,\ldots,L_1 \\ \mathbf{p}_{2(l-L_1)} & l = L_1+1,\ldots,U \end{cases} \tag{2.20}$$

The elements $\{K_{lm}\}_{lm=1}^{U}$ of the U-order square kernel matrix $\mathbf{K}$ are defined as:

$$K_{lm}(\sigma) = K(\mathbf{z}_l,\mathbf{z}_m) = \exp\left(-\frac{|\mathbf{z}_l - \mathbf{z}_m|^2}{2\sigma^2}\right) \tag{2.21}$$

In order to be a good kernel for the current classification problem, the matrix $\mathbf{K}$ should be so similar as possible to the ideal kernel matrix $\mathbf{J}$. So, the optimal σ should be selected to minimize the difference between $\mathbf{K}(\sigma)$ and $\mathbf{J}$. This difference can be efficiently evaluated as the mean absolute value of $\mathbf{K}-\mathbf{J}$:

$$A(\sigma) = \sum_{lm=1}^{U} \frac{|K_{lm}(\sigma) - J_{lm}|}{U^2} \tag{2.22}$$

The FSVC selects the optimal value σ^* for the kernel spread as:

$$\sigma^* = \arg\min_{\sigma\in\Sigma}\{A(\sigma)\}, \quad \Sigma = \{2^{-\frac{i+1}{2}}\}_{i=-13}^{13} \tag{2.23}$$

The set Σ is defined by the Libsvm guide [3] as a standard collection of 27 spread values which has proven to be adequate for most applications. The selection of σ requires $U^2 \leq (2L)^2$ differences $|\mathbf{p}_{ql} - \mathbf{p}_{rm}|$, where $q,r = 1,2$, while $l = 1,\ldots,L_q$ and $m = 1,\ldots,L_r$.

In multi-class classification problems ($Q > 2$), the FSVC uses the same σ value, that is calculated considering the classes 1 and 2, for all the binary FSVCs both with the OVO and OVA approaches. Although it might seem reasonable to have different spread values, it is common practice in the literature to use the same spread for all the binary SVCs in a multi-class SVC. Our experiments also confirmed that performance did not raise using different spread values, although the elapsed time was much higher due to the repetition of the tuning many times for each pair of binary SVCs, specially using an OVO setting (section 1.5 in chapter 1). Note that the proposed method is not restricted to RBF kernels, but can be applied to any kernel that requires hyper-parameter tuning (see eqs. 1.26-1.29 in chapter 1) such as polynomial or hyperbolic tangent, among others.

2.4 Large input dimensionality

The Cover theorem [6] states that a classification problem is linearly separable with more probability when the input patterns are high-dimensional. In datasets where the original dimensionality is already high, the use of non-linear (e.g. RBF) kernels do not increase the accuracy compared to the original data. In these cases, the linear and RBF kernel tend to achieve similar performances, but the the latter requires a larger computational complexity because it requires to calculate differences between training patterns, whose complexity scales with $\mathcal{O}(N^2)$, to calculate the kernel values for each support vector and test pattern, that scales as $\mathcal{O}(N,T)$, being T the number of test patterns (Table 1.1), and to perform the RBF kernel spread tuning. Using a linear kernel $K(\mathbf{x},\mathbf{y}) = \mathbf{x}^T\mathbf{y}$ the feature space is the same as the input space, so the weight vector $\mathbf{w}$ of the hyperplane $\mathcal{H}$ can be calculated explicitely as in eq. 1.11. In this case, eq. 2.18 simplifies because $k_{ql}(\mathbf{x}) = \mathbf{p}_{ql}^T\mathbf{x}$ and $k_{qlm} = \mathbf{p}_{ql}^T\mathbf{p}_{qm}$, so that:

$$y(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b), \quad \mathbf{w} = \sum_{l=1}^{L_2} \frac{\mathbf{p}_{2l}}{L_2} - \sum_{l=1}^{L_1} \frac{\mathbf{p}_{1l}}{L_1}, \tag{2.24}$$

$$b = \sum_{lm=1}^{L_1} \frac{\mathbf{p}_{1l}^T\mathbf{p}_{1m}}{2L_1^2} - \sum_{lm=1}^{L_2} \frac{\mathbf{p}_{2l}^T\mathbf{p}_{2m}}{2L_2^2} \tag{2.25}$$

Once calculated $\mathbf{w}$ and b from the set of U prototypes, the classification of a test pattern $\mathbf{x}$ only requires to perform a dot product of two I-dimensional vectors ($\mathbf{w}$ and $\mathbf{x}$) and to sum the offset b, being both operations computationally very efficient for datasets of arbitrary large size.

2.5 Large number of classes

When the number Q of classes in a multi-class classification problem is high, the OVO approach may be slow because a large number $Q(Q-1)/2$ of binary SVCs must be trained and tested, so the computational complexity scales as $\mathcal{O}(Q^2)$. In this cases, the OVA approach should be faster because it only uses Q binary SVCs, where the q-th binary SVC discriminates between class q ("positive"), and the remaining ones ("negative"). In FSVC with the OVA approach, the q-th binary FSVC has: 1) the positive class q with L_q prototypes $\{\mathbf{p}_{ql}\}_{l=1}^{L_q}$; and 2) the negative class, with L prototypes of the remaining classes selected by picking up the $W = \max(1, \lceil L/(Q-1) \rceil)$ most populated prototypes of each class (see the block "One-vs-all training/test only" in algorithm 2). When $(Q-1)W > L$, the last $(Q-1)W - L$ prototypes,

Algorithm 1: Multi-class Fast SVC (continued in algorithm 2).

Algorithm: FSVC($\{\mathbf{x}_n, c_n\}_{n=1}^{N}, \mathbf{x}$)

Data: $\mathbf{x}_n \in \mathbb{R}^I$: training pattern; $c_n \in \{1,\dots,Q\}$: class label; $\mathbf{x}$: test pattern
Result: $\mathcal{S}$: trained FSVC; y: predicted class for $\mathbf{x}$

$L \leftarrow 100$; $\{\mathbf{p}_{ql} \leftarrow \mathbf{0}; L_q, N_{ql} \leftarrow 0\}_{ql=1}^{Q,L}$
for $n \leftarrow 1, N$ **do**
  $q \leftarrow c_n$
  if $L_q < L$ **then**
    $l \leftarrow L_q$; $\mathbf{p}_{ql} \leftarrow \mathbf{x}_n$; $L_q \leftarrow L_q + 1$
  else
    $r \leftarrow \arg\min_{l=1\dots,L} \left|\mathbf{p}_{ql} - \mathbf{x}_n\right|$
    $\mathbf{p}_{qr} \leftarrow \left(1 - \frac{1}{N_{qr}}\right)\mathbf{p}_{qr} + \frac{\mathbf{x}_n}{N_{qr}}$; $N_{qr} \leftarrow N_{qr} + 1$
  end
end
Spread tuning, RBF kernel only————————————
$U \leftarrow L_1 + L_2$; $\{J_{lm} \leftarrow 0\}_{lm=1}^{U}$
$\{J_{lm} \leftarrow 1\}_{lm=1}^{L_1}$; $\{J_{lm} \leftarrow 1\}_{lm=L_1+1}^{U}$
$\Sigma \leftarrow \{2^{-\frac{i+1}{2}}\}_{i=-13}^{13}$; $\{\mathbf{z}_l \leftarrow \mathbf{p}_{1l}\}_{l=1}^{L_1}$; $\{\mathbf{z}_{L_1+l} \leftarrow \mathbf{p}_{2l}\}_{l=1}^{L_2}$
$\left\{K_{lm}(\sigma) \leftarrow \exp\left(\frac{-|\mathbf{z}_l - \mathbf{z}_m|^2}{2\sigma^2}\right)\right\}_{lm=1}^{U}$
$A(\sigma) \leftarrow \sum_{lm=1}^{U} \frac{|K_{lm} - J_{lm}|}{U^2}$; $\sigma^* \leftarrow \arg\min_{\sigma\in\Sigma}\{A(\sigma)\}$

which belong to the last class, must be discarded in order to have exactly L prototypes for the negative class. Therefore, only $Z = \max(1, W(2-Q)+L)$ prototypes of the last class are selected. When Q is high, $L < Q-1$ so that W=1, Z=1 and only one prototype of the first L classes is selected for the negative class.

Note that in a OVA setting each prototype can be of a given class (e.g. positive) in a given binary FSVC and of other class (e.g. negative) in other binary FSVC. Thus, prototypes do not need to be repeated for each binary FSVC, that only needs the indices of the prototypes for each class, while the prototypes themselves can be stored in memory just once, strongly reducing the memory requirements of multi-class FSVC.

Algorithm 2: Continuation of algorithm 1.

```
18 One-vs-one (OVO) training/test only————————————
19 {v_q ← 0}_{q=1}^{Q}; m ← 1; w ← Q − 1
20 for q ← 1, w do
21     for r ← q + 1, Q do
22         S_m ←FSVCtrain({p_ql}_{l=1}^{L_q}, {p_rl}_{l=1}^{L_r}, σ*)
23         u ←FSVCtest(S_m, x); m ← m + 1
24         if u ≥ 0 then
25             v_r ← v_r + 1
26         else
27             v_q ← v_q + 1
28 One-vs-all (OVA) training/test only————————————
29 W ← max(1, ⌈L/w⌉); m ← Q
30 for q ← 1, Q do
31     R ← {1, ..., q − 1, q + 1, ..., Q}
32     for r ← 1, w do
33         s ← R_r; u ← argsort({N_sl}_{l=1}^{L_s}, descending)
34         K_r ← {u_l}_{l=1}^{W}
35     Z ← max(1, W(2 − Q) + L); K_w = {K_wl}_{l=1}^{Z}
36     S_q ←FSVCtrain({p_ql}_{l=1}^{L_q}, {p_rl}_{r∈R; l∈K_r}, σ*)
37     v_q ←FSVCtest(S_q, x)
38 FSVC output————————————————————————————-
39 S ← {S_l}_{l=1}^{m}; y ← arg max_{q=1,...,Q} {v_q}
```

2.6 Complexity analysis

Algorithm 1, continued in algorithm 2, describes FSVC for multi-class classification ($Q > 2$) using the OVO and OVA approaches. Notation in line 2 means that $\mathbf{p}_{ql} = \mathbf{0}$ and $L_q, N_{ql} = 0$ for $q = 1, \ldots, Q$ and $l = 1, \ldots, L$. The training and test of binary FSVC are described by algorithms 3 and 4, respectively. The computational complexity of FSVC on large datasets where $L_q = L$ for $q = 1, \ldots, Q$, is as follows:

1. To create the set of QL prototypes $\{\mathbf{p}_{ql}\}_{ql=1}^{Q,L}$ requires to read, from some data files, N training patterns $\mathbf{x}_n$, of dimension I, and to calculate their distances to the L prototypes

Algorithm 3: Two-class Fast SVC training.

1 **Algorithm:** FSVCtrain($\{\mathbf{p}_{1l}\}_{l=1}^{L_1}$,$\{\mathbf{p}_{2l}\}_{l=1}^{L_2}$, σ)

Data: $\mathbf{p}_{ql}$: l-th prototype of class $q = 1,2$; σ: RBF kernel spread
Result: $\mathcal{S}$: binary FSVC

2 RBF kernel only———————————————————

3 $K(\mathbf{p}_{ql},\mathbf{p}_{qm},\sigma) \leftarrow \exp\left(\dfrac{-|\mathbf{p}_{ql}-\mathbf{p}_{qm}|^2}{2\sigma^2}\right)$

4 $\{k_{qlm} \leftarrow K(\mathbf{p}_{ql},\mathbf{p}_{qm},\sigma)\}_{q=1,lm=1}^{2,L_q}$

5 $b \leftarrow \displaystyle\sum_{lm=1}^{L_1} \frac{k_{1lm}}{2L_1^2} - \sum_{lm=1}^{L_2} \frac{k_{2lm}}{2L_2^2}$; $\mathcal{S} \leftarrow \{\{\mathbf{p}_{ql}\}_{ql=1}^{2,L_q}, b, \sigma\}$

6 Linear kernel only———————————————————

7 $\mathbf{w} \leftarrow \displaystyle\sum_{l=1}^{L_2} \frac{\mathbf{p}_{2l}}{L_2} - \sum_{l=1}^{L_1} \frac{\mathbf{p}_{1l}}{L_1}$

8 $b \leftarrow \displaystyle\sum_{lm=1}^{L_1} \frac{\mathbf{p}_{1l}^T\mathbf{p}_{1m}}{2L_1^2} - \sum_{lm=1}^{L_2} \frac{\mathbf{p}_{2l}^T\mathbf{p}_{2m}}{2L_2^2}$; $\mathcal{S} \leftarrow \{\mathbf{w}, b\}$

Algorithm 4: Two-class Fast SVC test.

1 **Algorithm:** FSVCtest($\mathcal{S},\mathbf{x}$)

Data: $\mathcal{S}$: binary FSVC; $\mathbf{x} \in \mathbb{R}^I$: test pattern
Result: $y \in \mathbb{R}$: output of the binary FSVC

2 RBF kernel: $\mathcal{S} = \{\{\mathbf{p}_{ql}\}_{ql=1}^{2,L_q}, b, \sigma\}$———————————

3 $\{k_{ql}(\mathbf{x}) \leftarrow K(\mathbf{p}_{ql},\mathbf{x},\sigma)\}_{ql=1}^{2,L_q}$ // $K(\mathbf{p}_{ql},\mathbf{x},\sigma)$ defined in line 3 of algorithm 3

4 $y(\mathbf{x}) \leftarrow \displaystyle\sum_{l=1}^{L_2} \frac{k_{2l}(\mathbf{x})}{L_2} - \sum_{l=1}^{L_1} \frac{k_{1l}(\mathbf{x})}{L_1} + b$

5 Linear kernel: $\mathcal{S} = \{\mathbf{w}, b\}$———————————————

6 $y \leftarrow \mathbf{w}^T\mathbf{x} + b$

of class c_n (lines 2-11 of algorithm 1). Its complexity scales as $\mathcal{O}(NIL)$, although L is a pre-defined constant that does not scale with the dataset size.

2. The spread tuning (lines 13-17 of algorithm 1) requires to calculate the $(2L)^2$ distances between the L prototypes (of dimension I) of classes 1 and 2, and to calculate $A(\sigma)$ for the 27 values of σ in Σ (eq. 2.23). This process is of complexity $\mathcal{O}(I)$.

3. With multi-class OVO classification, each of the $Q(Q-1)/2$ binary FSVCs requires $(2L)^2$ distances between prototypes to calculate $\{k_{qlm}\}_{q=1,lm=1}^{2,L}$ (eq. 2.19, line 16 in algorithm 1), which is of complexity $\mathcal{O}(IQ^2)$.

4. The test step (lines 20-27 and 29-37 in algorithm 2 for OVO and OVA multi-class approaches, respectively, and algorithm 4 for two classes) requires $2L$ values $\{k_{ql}(\mathbf{x})\}_{ql=1}^{2,L}$ for the T test patterns (see Table 1.1), of dimension I, and for the $Q(Q-1)/2$ binary FSVCs, which is $\mathcal{O}(Q^2TI)$.

Overall, the complexity of FSVC is $\mathcal{O}(NIQ^2T)$, so it is linear in N, I and T and quadratic in the number of classes Q. This complexity depends only on the dataset size (N, I, Q and T), as opposed to the standard SVC, whose speed is known to be different for datasets of the same size depending on their classification complexity (e.g., level of class overlap). The FSVCL, which uses linear kernel (see subsection 2.11 below), has not the item 2 in the previous list, but it keeps the quadratic complexity in Q. However, FSVCA and FSVCLA use the OVA approach (subsection 2.12), so their complexity on items 3-4 is linear in Q (the sorting of line 33 in algorithm 2 does not scale with the dataset size because only a constant number L of values are involved).

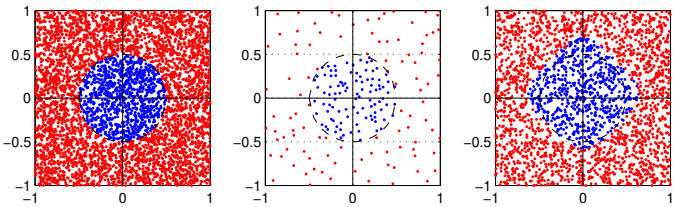

Figure 2.1: Left: circle-in-the-square dataset. Center: prototypes created by FSVC. Right: classification results.

It is interesting to show some illustrative example of the operation of FSVC for a 2D classification problem, mainly in the context of prototype determination (see section 2.2).

Figure 2.1 displays the classical circle-in-the-square classification dataset, with 5,000 patterns randomly generated and classified inside and outside a zero-centered circle of radius 0.5. The left panel shows the whole dataset, each pattern colored by its class label. The center panel shows the L=100 prototypes of each class created by FSVC from the training set (2,500 patterns). The right panel shows the classification developed by FSVC on the test set (the remaining 2,500 patterns), that is fairly accurate although with some errors on the class border.

2.7 Experimental setup

We compared FSVC and two state-of-the-art SVM-based classifiers: Libsvm [3], with RBF kernel (henceforth named SVC), and Liblinear (named LSVC), with linear kernel, that is designed for large-scale high-dimensional problems [11]. We also compared FSVC with the direct kernel perceptron (DKP), an efficient SVC-based classifier [13]. We used a collection of 44 classification datasets selected from the UCI Machine Learning Repository and from the Kaggle repository[2] (datasets `arabic`, `devanagari` and `fruits`). Table 2.1 reports the 22 small datasets (below 15,000 patterns, where the SVC can be executed, upper part of the table) and 22 large datasets (above 15,000 patterns, lower part), sorted by increasing populations, with their numbers of inputs and classes. On the large datasets, with up to 31 millions of patterns (`wesad`), 30,000 inputs and 131 classes (`fruits`), only FSVC and LSVC were executed, because SVC and DKP spend too much time and memory and are not able to finish their execution. Note that there are some differences in N, I and Q with respect to the UCI dataset information, e.g. in `wesad` and `har`. The algorithms were codified on the Octave[3] language v. 4.2.2, and the experiments were run under a computer with 8 Intel© Core^{TM} i7-4790K processors at 4GHz equipped with 32GB RAM and operative system Kubuntu 18.04. The program for FSVC is publicly available[4].

The experimental methodology was 4-fold cross-validation, with 2 training, 1 validation and 1 test folds on the small datasets. The values of the hyper-parameters, C and $\gamma = 1/2\sigma^2$ for SVC and γ for DKP, were selected to achieve the highest average kappa over the validation sets using the classical grid-search methodology. Since FSVC and LSVC have no hyper-parameter tuning and do not require a validation dataset, the 4-fold cross-validation on the large datasets used 3 training and 1 test folds, excepting `hepmass`, `imseg`, `mnist`,

[2] https://www.kaggle.com/datasets (March, 2022).
[3] http://octave.org (March, 2022).
[4] persoal.citius.usc.es/manuel.fernandez.delgado/papers/fsvc (March, 2022).

Original name	Dataset	N	I	Q
Fertility	fertility	100	23	2
Breast tissue	tissue	106	9	6
Hepatitis	hepatitis	155	19	2
Wine quality	wine	178	13	3
Sonar, mines vs. rocks	sonar	208	60	2
Seeds	seeds	210	7	3
Heart (Statlog)	heart	270	28	2
Ionosphere	ionosphere	351	33	2
Dermatology	dermatology	366	130	6
Monks	monks	432	17	2
Breast cancer Wisconsin (original)	wdbc	569	30	2
Synthetic control chart time series	synthetic	600	60	6
Australian credit approval	australian	690	43	2
Mammographic mass	mammograph	961	5	2
German credit	german	1,000	65	2
Isolet	isolet	1,559	617	26
Image segmentation	imseg	2,310	18	7
Abalone	abalone	4,177	8	3
Landsat satellite (Statlog)	sat	6,435	36	6
Musk (v.2)	musk	6,598	166	2
Electrical Grid Stability	grid	10,000	13	2
Nursery	nursery	12,958	27	4
Arabic handwritten characters	arabic	16,800	1,024	28
Magic gamma telescope	magic	19,020	10	2
Letter recognition	letter	20,000	16	26
Shuttle (Statlog)	shuttle	58,000	9	7
MNIST	mnist	70,000	784	10
WISDM smartphone and smartwatch activity and biometrics	wisdm	73,803	92	18
Fruits 360	fruits	90,380	30,000	131
Devanagari character	devanagari	92,000	1,024	46
IJCNN 2001 Competition	ijcnn1	141,691	22	2
Covertype	covtype	581,012	54	7
Poker hand	poker	1,025,010	10	2
KDD Cup 1999	kddcup	4,000,000	122	23
SUSY	susy	5,000,000	18	2
Record linkage comparison patterns	record	5,749,132	11	2
Physical unclonable functions	physical	6,000,000	128	2
Detection of IoT botnet attacks	baiot	7,062,606	115	11
HEPMASS	hepmass	10,500,000	28	2
HIGGS	higgs	11,000,000	28	2
Human Activity Recognition from Continuous Ambient Sensor Data	human	13,956,557	36	42
Kitsune network attack	kitsune	21,017,597	115	9
Heterogeneity Activity Recognition	har	29,097,887	14	6
Wearable stress and affect detection	wesad	31,470,603	8	4

Table 2.1: List of the small (top) and large (bottom) classification datasets.

`physical`, `poker` and `fruits`, that used the original separated train and test files without 4-fold cross-validation. The performance measurement was the Cohen kappa score [27], that evaluates the agreement between the true class label and the class predicted by the classifier discarding the agreement by change, e.g. with unbalanced datasets.

Sections 2.8-2.12 below study the influence of the aspects of FSVC described in sections 2.1-2.5, respectively. Section 2.13 compares FSVC to the other approaches in terms of per-

	Training method	Kernel calculation	Tuning method	Kernel type	Multi-class approach
SVC	Libsvm	Support vectors	Grid-search	RBF	One-vs-one
SVC1	**Efficient**	**All patterns**	"	"	"
SVC3	Libsvm	**Prototypes**	"	"	"
SVC2	"	Support vectors	**Efficient**	"	"
FSVC1	Efficient	**All patterns**	**Grid-search**	"	"
FSVC2	"	Prototypes	**Grid-search**	"	"
FSVC	"	Prototypes	Efficient	"	"
FSVCL	"	"	"	**Linear**	"
FSVCA	"	"	"	RBF	**One-vs-all**
FSVCLA	"	"	"	**Linear**	**One-vs-all**

Table 2.2: Versions of SVC and FSVC (in bold those features which are different from the original SVC or FSVC). The symbol " means 'equal as above'.

formance and time. Sections 2.14 and 2.15 compare FSVC with other SVM solvers and other spread tuning methods in the literature. Section 2.16 compares times of SVC with and without the algorithmic modifications proposed on this thesis. Section 2.17 compares FSVC with evolutionary training set selection methods, and section 2.18 discusses the memory requirements of FSVC.

Dataset	SVC	SVC1	SVC2	SVC3	FSVC1	FSVC2
fertility	-1.6	4.1	30.7	-1.6	6.5	15.9
tissue	62.1	59.1	63.9	62.1	49.7	49.7
hepatitis	36.2	48.7	34.7	36.2	52.9	52.9
wine	97.5	95.8	97.5	97.5	95.8	95.8
sonar	63.2	**41.4**	71.8	63.2	50.1	50.1
seeds	89.1	87.2	89.2	89.1	83.5	83.5
heart	35.9	26.4	43.0	37.0	**18.5**	**18.5**
ionosphere	86.8	**56.8**	82.3	84.5	**61.4**	**62.7**
dermatology	95.2	95.6	89.6	95.2	97.0	97.0
monks	100.0	93.5	100.0	100.0	85.2	**81.0**
wdbc	87.6	81.4	89.5	83.9	79.2	80.2
synthetic	97.4	95.0	99.0	97.4	93.2	93.2
australian	70.1	66.7	60.3	71.0	66.3	67.2
mammograph	65.2	57.2	61.9	63.0	58.3	58.7
german	33.6	36.8	36.7	36.1	39.0	35.8
isolet	95.2	86.1	95.4	95.2	83.7	83.7
imseg	89.1	80.8	90.7	89.1	81.1	81.1
abalone	33.8	29.5	33.9	32.4	28.8	29.7
sat	89.8	77.2	89.2	86.7	78.7	78.4
musk	98.8	**78.2**	97.2	**82.0**	**74.9**	**80.8**
grid	99.2	87.9	96.8	94.2	88.6	**81.1**
nursery	100.0	96.7	100.0	**82.5**	95.5	**76.3**
Average	73.8	67.4	75.1	71.7	66.7	66.1
p-value	—	0.1697	0.9625	0.4595	0.1300	0.0803

Table 2.3: Kappa (in %) of SVC, SVC1, SVC2, SVC3, FSVC1 and FSVC2 on the small datasets (in bold values below SVC by more than 15%).

2.8 Influence of efficient training

In order to evaluate the influence of the efficient training (subsection 2.1), we compared the kappa achieved by SVC on the small datasets with a version of SVC, named SVC1 (2nd row in Table 2.2), that uses the proposed efficient training with all the training patterns; and with a version of FSVC, named FSVC1 (5th row in Table 2.2), that uses efficient training but without efficient kernel (it uses all the training patterns instead of class prototypes) and grid-search for spread tuning (selection of the σ with the highest average kappa over the validation sets). Their kappa values are reported by Table 2.3, alongside with the p-value of the Wilcoxon ranksum test, developed using the Matlab function `ranksum`, comparing them to SVC. In most datasets the SVC1 and FSVC1 achieve kappa similar to SVC, with average values (67.4% and 66.7%) only 6-7% below SVC overcoming 15% only in 3 of 22 datasets, so the efficient training is often an accurate approximation to SVC that can be applied on large datasets. The difference is not statistically significant (p=0.1697) but explains the averate loss from SVC (73.8%) to FSVC (67.1%) in Table 2.5 below. Considering times (see Table 2.10 in section 2.16 below), the SVC1 speeds up the SVC about 5 times in average over the small and some large datasets, although the use of the whole training set in SVC1 (that does not use prototypes) reduces this difference for in large datasets.

2.9 Influence of efficient kernel calculation

The method described in subsection 2.2 for the efficient kernel calculation, based on the use of prototypes instead of support vectors, is tested by comparing SVC with SVC3 (3rd row in Table 2.2), a version of SVC trained on these prototypes, and with FSVC2 (6th row in Table 2.2), a version of FSVC which combines efficient training and prototypes with grid-search for spread tuning. The impact of using prototypes instead of all the patterns in SVC3 and FSVC2 (see Table 2.3) is also fairly low. Considering SVC3, the average kappa values is 71.7%, only 2% below SVC, with p-value=0.4595, being below SVC by more than 15% in only 2 of 22 datasets. With respect to FSVC2, the average kappa is 66.1%, with lower p-value=0.080, only 7% below SVC, being below SVC by more than 15% in only 6 22 datasets. The speed up of SVC3 with respect to SVC (see Table 2.10 in section 2.16 below) raises fastly with the dataset size, being 6.11 times faster on average over the small and some large datasets.

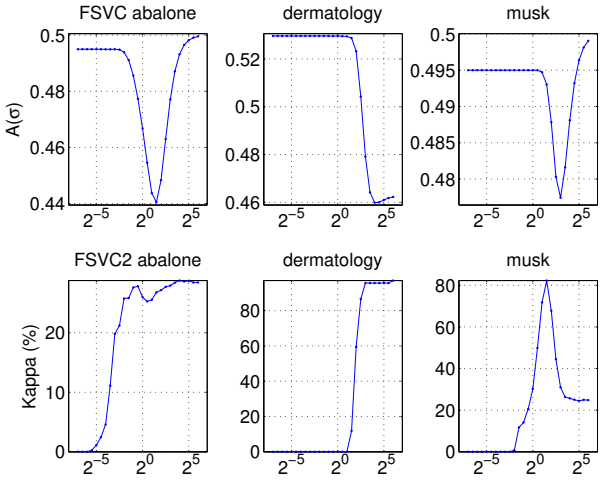

Figure 2.2: Upper panels: examples of function $A(\sigma)$ of eq. 2.22 for datasets `abalone`, `dermatology` and `musk`. Lower panels: profiles of kappa vs. σ achieved by FSVC2 with grid-search tuning on these datasets.

2.10 Influence of efficient RBF spread tuning

The upper panels in Figure 2.2 plot the function $A(\sigma)$ of eq. 2.22 for three datasets. The left example, with a clear minimum on $A(\sigma)$, is the most frequent case, although some datasets (`dermatology`, `fertility` and `hepatitis`) exhibit a sigmoid-like profile (upper center panel) for $A(\sigma)$. The lower panels exhibit the kappa achieved by FSVC2 (see subsection 2.9), which uses grid-search tuning, on these datasets. In the lower left and center panels, there is a region of low kappa values located on the left (low σ values) and kappa increases when σ raises. The majority of the datasets that we studied exhibits this increasing sigmoid profile of kappa vs. σ. Although the σ with the highest kappa in the lower panel does not exactly match the lowest A in the upper panel, the minimum of A is usually located inside the region of high kappa values. Indeed, alongside with the minimum of A, the highest value $\sigma = \max\{2^{-\frac{i+1}{2}}\}_{i=-13}^{13} = 2^6$ also achieves kappa very near to the maximum.

There are also some datasets (`monks` and `musk`) where the kappa profile has a maximum (lower right panel in Figure 2.2) instead of a sigmoid profile, and the highest σ, or the σ that minimizes A, give poor performance, but the highest kappa (low right panel) matches the place where $A(\sigma)$ reaches its left asymptotic value (upper right panel), henceforth named

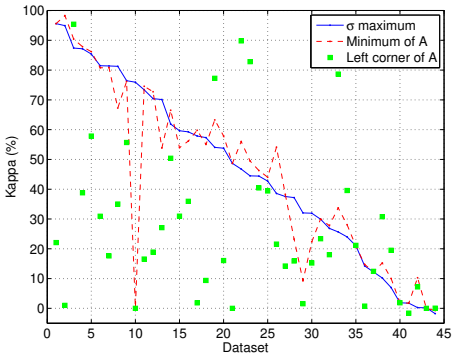

Figure 2.3: Kappa achieved by FSVC for each dataset using $\sigma^* = \sigma_{\max}$, in blue, $\sigma^* = \arg\min\{A\}$, in red, and σ^* on the left corner of A, in green.

"the left corner" of the minimum of $A(\sigma)$. Thus, three tuning methods can be considered: 1) to select σ^* on the maximum value $\sigma_{\max} = 2^6$, which avoids the spread tuning; 2) to select, following eq. 2.23, the value $\sigma^* = \arg\min_\sigma\{A(\sigma)\}$; and 3) to select σ^* on the left corner of $A(\sigma)$. This corner can be found by starting from the lowest value $\sigma_{\min} = 2^{-7}$, and searching the first value of σ where the derivative $A'(\sigma)$ overcomes a threshold value τ, set empirically to $\tau = 0.05 \max\left\{A'(\sigma)\right\}$, i.e., the 5% of the maximum of $A'(\sigma)$.

Figure 2.3 plots the kappa achieved by FSVC using these three tuning methods. The horizontal axis identifies the datasets, sorted by decreasing Kappa of FSVC using $\sigma^* = \sigma_{\max}$ (blue line), in order to enhance the figure visualization. Blue and red lines, which correspond to $\sigma^* = 2^6$ and $\sigma^* = \arg\min\{A(\sigma)\}$, respectively, achieve in general similar results, while the green points, that correspond to σ^* on the left corner of $A(\sigma)$, see the upper right panel of Fig. 2.2, achieves much less kappa excepting some datasets, where it overcomes methods 1 and 2. This figure suggests that method 1, which does not require tuning, works well usually (its performance is very near the maximum in 34 of 44 datasets), in 7 datasets method 3 is required to achieve good performance, and only in 3 datasets method 2 slightly overcomes method 1.

The influence of the efficient spread tuning can be estimated by comparing the performance and time of SVC on the small datasets with RBF kernel spread selected: 1) using the proposed efficient tuning, a version of SVC named SVC2 in Table 2.2; 2) using the kernel density estimation (KDE), proposed in [28]; and 3) using the standard grid-search (SVC). Table 2.3 above and Table 2.9 in section 2.15 report these results: the SVC2 (average kappa

75.1%) outperforms KDE (74.1%) and SVC (73.8%), being in average 27 times faster than SVC and so fast as KDE. Another way to estimate the influence of the efficient spread tuning is to compare FSVC and FSVC2 (6th row in Table 2.2), that uses grid-search, whose average kappa (66.1% in Table 2.3) is only slightly lower than FSVC (67.1% in Table 2.5) with non-significant difference to SVC (p=0.0803). Since the efficient tuning raises the performance of SVC2 compared to SVC, and grid-search reduces the performance of FSVC2 compared to FSVC, our method seems to perform similarly to grid-search but being faster.

2.11 Influence of large dimensionality

Table 2.4 (upper part) reports kappa and time per fold achieved by FSVC and FSVCL, a version of FSVC that uses linear kernel (8th row in Table 2.2), in high-dimensional datasets with more than 100 inputs. The times are in format h:m:s, m:s or s.ss, where h, m and s mean hours, minutes and seconds. Note that 1.02 means 1.02 sec., while 1:03 means 1 minute and 3 sec. or 63 sec. The FSVCL outperforms FSVC on dataset `fruits`, with 30,000 inputs and 131 classes, by almost 10 points being about 5 times faster, which confirms that the former is a good choice for large datasets with many inputs. However, on datasets `kitsune` and `musk` FSVC outperforms FSVCL by 12 and 53 points, respectively. Globally, FSVC achieves higher average kappa than FSVCL (52.0% vs. 46.4%) although the latter is about 2 times faster in average.

2.12 Influence of large number of classes

The middle part of Table 2.4 reports the kappa and time per fold of FSVC (that uses the OVO approach) and FSVCA (that uses the OVA approach, described in 9th row of Table 2.2) on multi-class datasets with more than 5 classes. Note that, for each dataset, $S = Q(Q-1)/2$ is the number of binary SVCs required when the OVO approach is used. On the `fruits` dataset, FSVCA uses L=50, instead of the default value (L=100), due to the large I=30,000 and Q=131, so that W=Z=1 and only one prototype of the first L classes is selected for the negative classes. The FSVCA achieves better kappa than FSVC in all the datasets, with an average of 56.8% vs. 53.6%. Surprisingly, the times of FSVCA are slightly higher than FSVC, what may be caused by: 1) the vectorization of FSVC, that reduces the slowness caused by the quadratic number of binary FSVCs compared to FSVCA; and 2) the time required by FSVCA to manage prototype indices for the negative classes (lines 30-37 in algorithm 2). The lower

				Kappa (%)		Time(s/fold)	
Dataset	N	I		FSVC	FSVCL	FSVC	FSVCL
dermatology	366	130		95.6	95.9	0.10	0.09
isolet	1,559	617		86.2	85.4	1.79	1.41
musk	6,598	166		78.7	25.4	1.84	1.81
arabic	16,800	1,024		21.1	21.6	1:06	27.48
mnist	70,000	784		72.7	71.1	1:46	1:25
fruits	90,380	30,000		32.0	41.6	5:28:9	1:24:17
devanagari	92,000	1,024		57.9	53.1	11:58	2:16
kddcup	4,000,000	122		70.1	69.6	17:23	10:56
physical	6,000,000	128		0.2	0.2	14:14	13:33
baiot	7,062,606	115		38.3	38.8	39:19	24:23
kitsune	21,017,597	115		19.5	7.5	1:32:27	1:13:26
Average				52.0	46.4	46:02	19:10
Dataset	N	Q	S	FSVC	FSVCA	FSVC	FSVCA
tissue	106	6	15	57.3	59.3	0.01	0.06
dermatology	366	6	15	95.6	96.2	0.10	0.17
synthetic	600	6	15	95.4	96.8	0.09	0.18
isolet	1,559	26	325	86.2	85.3	1.79	2.00
sat	6,435	6	15	74.5	77.7	0.52	0.68
arabic	16,800	28	378	21.1	22.1	1:06	58.80
letter	20,000	26	325	77.2	78.3	1.29	1.50
shuttle	58,000	7	21	82.8	87.6	1.35	1.58
mnist	70,000	10	45	72.7	74.9	1:46	2:49
wisdm	73,803	18	153	30.8	30.5	19.47	22.95
fruits	90,380	131	8,515	32.0	32.6	5:28:9	4:55:48
devanagari	92,000	46	1,035	57.9	59.3	11:58	7:23
covtype	581,012	7	21	39.6	40.0	57.45	1:12
kddcup	4,000,000	23	253	70.1	91.9	17:23	52:00
baiot	7,062,606	11	55	38.3	46.4	39:19	38:12
human	13,956,557	42	861	12.4	12.5	50:21	58:39
kitsune	21,017,597	9	36	19.5	27.9	1:32:27	1:40:16
har	29,097,887	6	15	1.9	3.3	24:24	24:44
Average				53.6	56.8	31:34	32:22
Dataset	N	I	Q	FSVC	FSVCLA	FSVC	FSVCLA
arabic	16,800	1,024	28	21.1	22.2	1:06	1:48
fruits	90,380	30,000	131	32.0	41.3	5:28:9	1:10:42
devanagari	92,000	1,024	46	57.9	53.1	11:58	10:01
Average				37.0	38.9	1:53:44	27:30

Table 2.4: Kappa and time per fold of FSVC and FSVCL (linear kernel) in high-dimensional datasets (upper part), by FSVC and FSVCA (one-vs-all) in multi-class datasets (middle part, here $S = Q(Q-1)/2$) and by FSVC and FSVCLA (linear kernel, one-vs-all) in high-dimensional multi-class datasets (lower part).

part of Table 2.4 compares FSVC and FSVCLA (linear kernel and OVA approach, 10th row in Table 2.2) on the datasets with more than 1,000 inputs and 5 classes, simultaneously. The average performance of FSVCLA is better than FSVC (38.9% vs. 37.0%), being also four times faster in average.

	Kappa (%)				Time(s/fold)			
Dataset	FSVC	SVC	LSVC	DKP	FSVC	SVC	LSVC	DKP
fertility	1.8	-1.6	2.3	0.0	0.01	0.09	0.01	0.07
tissue	59.3	62.1	64.4	63.8	0.06	0.10	0.00	0.14
hepatitis	48.7	36.2	32.5	33.0	0.01	0.14	0.01	0.10
wine	98.3	97.5	97.5	95.8	0.03	0.16	0.03	0.14
sonar	54.1	63.2	41.5	64.1	0.04	0.27	0.02	0.17
seeds	87.8	89.1	94.0	85.8	0.01	0.14	0.01	0.17
heart	27.8	35.9	66.8	29.7	0.04	0.38	0.01	0.19
ionosphere	**59.6**	86.8	71.7	87.7	0.03	0.46	0.02	0.27
dermatology	96.2	95.2	96.2	95.6	0.17	1.65	0.05	0.83
monks	89.8	100.0	32.9	94.4	0.05	1.64	0.01	0.29
wdbc	81.3	87.6	90.2	78.5	0.05	0.86	0.03	0.44
synthetic	96.8	97.4	76.0	96.8	0.18	1.82	0.06	1.15
australian	66.5	70.1	69.5	66.1	0.08	1.82	0.02	0.62
mammograph	59.3	65.2	63.4	60.2	0.03	10.66	0.02	0.62
german	37.2	33.6	38.3	25.3	0.14	4.64	0.05	1.31
isolet	86.2	95.2	64.8	86.3	1.79	1:48	15.83	32.24
imseg	81.4	89.1	79.4	83.2	0.11	0.43	0.10	1.01
abalone	30.0	33.8	30.7	30.6	0.13	2:17	0.05	5.41
sat	77.7	89.8	71.8	88.4	0.68	2:02	0.39	27.07
musk	**78.7**	98.8	51.5	86.2	1.84	4:54	1.97	2:10
grid	**81.5**	99.2	75.0	87.8	0.29	2:15	0.20	20.85
nursery	**76.5**	100.0	87.7	94.1	0.47	8:30	0.33	1:08
Average	67.1	73.8	63.6	69.7	0.28	1:01	0.87	13.26
p-value	—	0.142	0.589	0.489	—	4e-05	0.11	2e-04

Table 2.5: Kappa and time/fold of FSVC, SVC, LSVC and DKP on small datasets (in bold the kappa values of FSVC that are below SVC by more than 15 points).

2.13 Comparison of FSVC, SVC, LSVC and DKP

The left parts of Tables 2.5 and 2.6 report the kappa of FSVC, SVC, LSVC and DKP on the small datasets, and by FSVC and LSVC on the large datasets, respectively. The kappa values of FSVC on high-dimensional (resp. multi-class) datasets is the maximum between FSVC and FSVCL (resp. FSVCA), so in general they may differ with respect to Table 2.4. The SVC achieves the best average kappa (73.8%) on the small datasets (Table 2.5), followed by DKP (4.1 points below) and FSVC (difference of 6.7 points, not statistically significant with p=0.142), that is 3.5 points above LSVC (p=0.589). On the large datasets (Table 2.6), the LSVC fails by lack of memory in four datasets: `fruits`, `kitsune`, `har` and `wesad`. The FSVC achieves an average kappa, over datasets where LSVC does not fail, 4.6 points above LSVC, although the difference is not statistically significant (p=0.716). This p-value is higher than for small datasets (p=0.589), where the difference in average kappa is lower, because the number of datasets, discarding the LSVC fails, is lower. Overall, the performance of FSVC is near SVC on the small datasets and slightly better than LSVC. It worths to note that datasets `musk`, `ijcnn1` and `covtype` are also used in the recent paper [32], that uses

SVM alongside with training set selection. This approach spends 43 minutes, 1.5 hours and 28.2 hours (one day and 4 hours), respectively, while FSVC spends 1.8 seconds, 6.57 sec. and 1 minute 12 sec., respectively, that are very large differences in elapsed times.

	Kappa (%)		Time(s/fold)			Time(ms/pat)	
Dataset	FSVC	LSVC	FSVC	LSVC	t_{LSVC}/t_{FSVC}	Train	Test
arabic	22.1	25.4	58.80	2:35:51	159.03	1.61	10.86
magic	46.3	52.5	0.44	0.32	0.74	0.02	0.03
letter	78.3	52.9	1.50	0.81	0.54	0.04	0.14
shuttle	87.6	46.8	1.58	1.01	0.64	0.02	0.03
mnist	74.9	77.0	2:49	41:08	14.61	1.23	3.25
wisdm	30.8	25.0	19.47	24.04	1.23	0.15	0.62
fruits	41.6	—	1:24:17	—	—	52.69	11:50
devanagari	59.3	55.1	7:23	30:44	4.16	1.66	26.25
ijcnn1	31.9	22.0	6.57	4.34	0.66	0.04	0.06
covtype	40.0	52.7	1:12	26.39	0.36	0.07	0.19
poker	10.3	0.1	30.65	21.21	0.69	0.02	0.03
kddcup	91.9	97.8	52:00	1:50:34	2.13	0.16	0.57
susy	44.0	54.3	3:06	2:13	0.72	0.03	0.05
record	75.9	1.8	2:29	2:27	0.99	0.02	0.03
physical	0.2	0.1	14:14	14:36	1.03	0.13	0.18
baiot	46.4	61.2	38:12	5:34:13	8.75	0.22	0.68
hepmass	59.8	67.2	9:59	7:14	0.72	0.05	0.07
higgs	14.8	27.1	10:27	6:51	0.66	0.05	0.07
human	12.5	23.6	58:39	22:49	0.39	0.07	0.64
kitsune	27.9	—	1:40:16	—	—	0.20	0.46
har	3.3	—	24:44	—	—	0.04	0.09
wesad	0.1	—	12:30	—	—	0.02	0.04
Average	45.9	41.3	11:15	40:33	11.00	0.28*	2.11*
p-value	—	0.716	—	0.74			

Table 2.6: Kappa and time per fold achieved by FSVC and LSVC, and ratio t_{LSVC}/t_{FSVC}, on the large datasets. The last two columns report the training and test times per pattern of FSVC. The values with asterisk (*) are calculated discarding dataset `fruits` as an outlier.

The right parts of Tables 2.5 and 2.6 report the times (in the same format as in Table 2.4) on the small and large datasets, respectively. Small datasets report the times of FSVC, SVC, LSVC and DKP, while large datasets only report the times of FSVC and LSVC. Considering small datasets, the average ratio of times t_{SVC}/t_{FSVC} over the small datasets (not shown in Table 2.5) is 162.5, with values ranging from 1.6 to 1,097 among datasets, so FSVC is about two orders of magnitude faster than SVC, difference that is statistically significant (p=4·10^{-5}). The difference in times between FSVC and LSVC on the small datasets is not significant (p=0.11), because the ratio t_{FSVC}/t_{LSVC}=2.96 in average, but the average time[5] of LSVC (0.87 s/fold) is higher than FSVC (0.28). The FSVC is 47 times faster than DKP in average, difference that is significant (p=2·10^{-4}).

[5]This difference is caused by the `isolet` dataset, where LSVC and FSVC spend 15.83 and 1.79 s/fold, respectively.

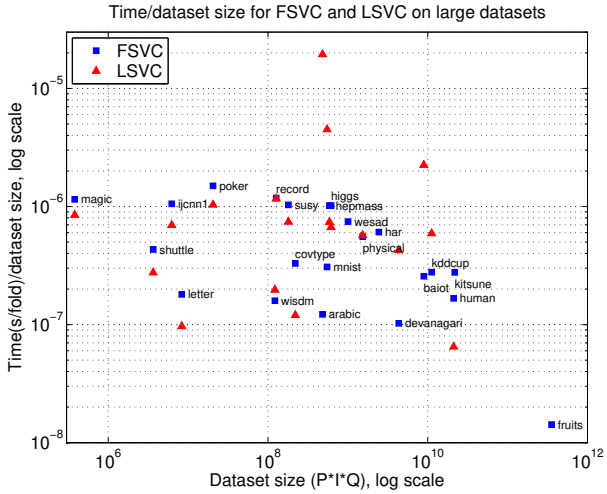

Figure 2.4: Time of FSVC and LSVC per pattern, input and class on the large datasets vs. dataset size.

On the large datasets, the time of FSVC is similar to or lower than LSVC, and the average time of FSVC (11m 15s), over the datasets where LSVC does not fail, is 3.6 times lower than LSVC (40m 33s). This difference, which is not significant (p=0.74) is caused by the high times of LSVC in some datasets (`arabic`, `mnist`, `devanagari`, `kddcup` and `baiot`). Since they are not the largest datasets, this means that the time spent by LSVC, and also by SVC (see e.g. the times of SVC on datasets `mammograph` and `isolet`), depends on the difficulty of the classification problem, and not only on its size, while the time spent by FSVC only depends on its size. The 6th column of Table 2.6, labeled $t_{\text{LSVC}}/t_{\text{FSVC}}$, reports high values on these datasets with an average of 11.0, so the FSVC is about one order of magnitude faster than LSVC. Besides, LSVC is very unstable, being faster than FSVC in some datasets and much slower in others. The FSVC is able to classify datasets as `wesad` (31 millions of patterns) and `har` (29 million patterns) in less than 0.5 hours, `human` (14 million patterns and 42 classes) in less than 1 hour, or `kitsune` (21 million patterns and 115 inputs) and `fruits` (90,000 patterns, 30,000 inputs and 131 classes) in less than 2 hours on a standard computer with 32GB of RAM, while LSVC fails in four of these five datasets by lack of memory.

The train and test times per pattern of FSVC (last two columns of Table 2.6) are very low (average 0.28 and 2.11 ms./pat.), being the test slower than the training excepting the `fruits`

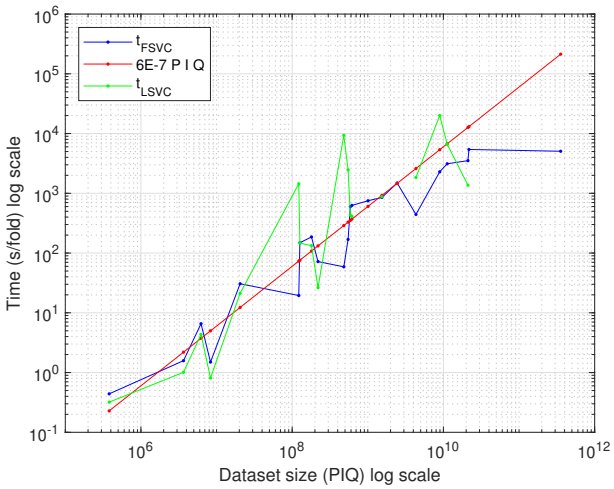

Figure 2.5: Time spent by FSVC, time estimated ($6 \cdot 10^{-7} PIQ$) for FSVC and time of LSVC on the large datasets vs. dataset size.

dataset, due to its high I. The largest datasets (`higgs`, `har`, `human`, `kitsune` and `wesad`) do not achieve the largest times per pattern (achieved with `fruits`, `devanagari`, `arabic` and `mnist`, with high Q and I simultaneously), so the time per pattern of FSVC decreases with the dataset size. Figure 2.4 confirms this by plotting the time of FSVC (blue squares) and LSVC (red triangles), divided by the dataset size (PIQ, where $P = N + T$ is the total number of patterns and T is the number of test patterns), against the dataset size, in a logarithmic scale, for each dataset. The average time of FSVC and LSVC per pattern, input and class are $6{\cdot}10^{-7}$ and $1.9{\cdot}10^{-6}$ sec., respectively. The FVSC is faster than LSVC on the right part of the plot (larger datasets such as `kddcup`, `baiot`, `devanagari`, `mnist` and `arabic`), excepting `human` due to its high Q (the red points of `fruits`, `har`, `wesad` and `kitsune` are missing because LSVC fails), although LSVC is faster in the left and center parts (smaller datasets).

Figure 2.5 plots the true time spent by FSVC, the estimated time ($6 \cdot 10^{-7} \times PIQ$) and the time of LSVC on the large datasets. True and estimated times for FSVC are fairly similar, while LSVC varies much more than FSVC with respect to the dataset size (PIQ). Thus, the time spent by FSVC for any dataset can be estimated with a good accuracy, as opposed to LSVC or SVC, whose times also depend on the dataset difficulty, being very unstable among datasets. Both FSVC and LSVC have lower times on the right part of the plot (larger datasets)

than on the left part (smaller datasets), which confirms that the cost of FSVC per pattern, input and class reduces with the dataset size.

2.14 Comparison with other SVM solvers

We implemented the primal estimated sub-gradient solver for SVM (Pegasos-SVM) in Octave using linear and RBF kernel [39], alongside with the sublinear importance-sampling bi-stochastic algorithm [18], named SVM-SIMBA in the literature and referred henceforth as SIMBA. Both solvers have been proposed for binary classification. For the linear Pegasos, we tuned the regularization parameter λ with values $\{2^i\}$, with i from -5 to 15 with step 2. With RBF Pegasos, λ was tuned equally to linear Pegasos, while the inverse γ of the RBF kernel spread with values $\{2^i\}_{i=-13}^{13}$, corresponding to the values used with SVC. In SIMBA, we used $\nu = 5 \cdot 10^{-5}$, value recommended in figure 1 of [18] for dataset "real vs. simulated". Since the paper does not specify any value for ε and the number T of iterations verifies $T = 10^4 \varepsilon^{-2} \log N$, being N the number of training patterns, we used $\varepsilon = 1$ in order have a moderate T and to avoid very slow executions.

	Kappa (%)				Time (s)			
Dataset	FSVC	LP	RP	SMB	FSVC	LP	RP	SMB
fertility	1.8	-8.1	2.6	-5.8	0.02	0.72	30.22	7e+02
hepatitis	48.7	35.1	11.2	3.3	0.06	0.72	31.93	1e+03
sonar	54.1	44.3	16.1	-4.8	0.17	1.00	39.88	1e+03
heart	27.8	10.8	-3.6	0.0	0.14	0.78	38.48	2e+03
ionosph.	59.6	71.1	30.9	-48.3	0.12	0.85	42.10	2e+03
monks	89.8	38.4	0.0	24.5	0.20	0.83	39.74	3e+03
wdbc	81.3	85.8	43.9	0.4	0.19	1.13	50.22	3e+03
australian	66.5	69.7	31.9	0.0	0.31	2.81	61.79	4e+03
mamm.	59.3	56.3	55.9	32.4	0.12	0.89	49.73	6e+03
german	37.2	34.5	2.8	-0.8	0.58	2.99	91.34	6e+03
musk	78.7	30.5	14.0	-7.5	7.35	10.72	1961.00	5e+04
grid	81.5	71.7	68.5	-34.6	1.17	5.61	642.11	8e+04
magic	46.3	49.7	10.2	—	1.75	8.33	1678.17	—
poker	10.3	0.1	—	—	30.65	20.86	—	—
susy	44.0	38.5	—	—	743.38	744.84	—	—
record	75.9	0.1	—	—	595.86	698.92	—	—
physical	0.2	0.1	—	—	853.99	846.08	—	—
hepmass	59.8	48.8	—	—	598.93	555.29	—	—
higgs	14.8	8.6	—	—	2507.62	2307.08	—	—
Average	49.3	36.1	21.9	-3.4	—	6.05	454.79	2e+04
p-value	—	0.12	0.004	1e-04	—	0.102	2e-05	4e-05

Table 2.7: Comparison of FSVC, linear (LP) and RBF (RP) Pegasos, and SIMBA (SMB) on the binary datasets.

On the `covtype` dataset (class 1 vs. others) used in the subsection 7.1 of [39], FSVC outperforms linear Pegasos (kappa=42.46% against 34.84%), that used the recommended λ

value, being only slightly slower (190.34 s. against 162.9 s. with Pegasos). Using RBF kernel on the `usps` dataset (2 classes, 4-fold), Pegasos spent 41.44 s. and achieved kappa=2%, while FSVC outperforms Pegasos both in time (16.24 s.) and performance (kappa=70.26%). On the `mnist` dataset (2 classes, separated train and test sets), Pegasos spent 1,943.56 seconds (32 minutes, 23.56 seconds) and achieved kappa=-0.38% (accuracy=81.26%), while FSVC spent 96.52 seconds with kappa=66.58% (accuracy=94.08%). Therefore, FSVC outperforms Pegasos both in performance and speed (with high difference) with linear and RBF kernel on these datasets.

Table 2.7 reports the kappa and time spent by linear (LP) and RBF (RP) Pegasos, and by SIMBA (SMB), on the binary datasets of Table 2.1. Large datasets (below `magic`) only have training and test set, so the tuning of the RBF kernel spread can not be developed, as with SVC, and the results for RP are missing. There are also missing values in the SIMBA column caused by excessive memory requirements. The average values on the time columns are the average of the time ratios between the corresponding method (linear or RBF Pegasos, and SIMBA) and FSVC. The average kappa values of linear (36.1%) and RBF Pegasos (21.9%), and of SIMBA (-3.4%), are far from FSVC (49.3%), although linear Pegasos slightly outperforms FSVC on datasets `ionosphere`, `wdbc`, `australian` and `magic`, being very unstable on the remaining datasets. The differences in kappa are statistically significant for RBF Pegasos (p=0.004) and SIMBA ($p = 10^{-4}$). Regarding execution times, the linear Pegasos is 6 times slower than FSVC, while the RBF Pegasos and SIMBA are 454.8 and 21,774 times slower, respectively, time differences that are statistically significant ($p = 2 \cdot 10^{-5}$ and $p = 4 \cdot 10^{-5}$). Overall, FSVC outperforms the three methods both in performance and speed.

We also compared the FSVC to the core vector machine [44] and other coreset methods [17, 45]. Specifically, we used the Matlab implementation of the indefinite core vector machine (ICVM), provided by the authors[6]. Table 2.8 compares the kappa and time of FSVC and ICVM [38] on the small and some large datasets (ICVM could not be run in datasets larger than `shuttle`). The ICVM achieves performance clearly below FSVC (45.5% vs. 64.6%), being about 105 times slower than FSVC in average, and being even much slower in the `isolet` dataset. The ICVM is faster than FSVC in `arabic`, although the kappa is much lower (3.1% and 21.1% using ICVM and FSVC, respectively). Both differences in performance and time are statistically significant (p=0.028 and $p = 4.5 \cdot 10^{-7}$, respectively).

[6] https://www.techfak.uni-bielefeld.de/∼fschleif/software.xhtml (March, 2022).

	Kappa (%)		Time (s)	
Dataset	FSVC	ICVM	FSVC	ICVM
fertility	1.8	3.9	0.02	1.10
tissue	57.3	42.8	0.02	2.73
hepatitis	48.7	37.9	0.06	1.94
wine	98.3	93.2	0.12	2.72
sonar	54.1	33.1	0.17	4.17
seeds	87.8	83.4	0.05	3.83
heart	27.8	69.1	0.14	6.39
ionosphere	59.6	53.2	0.12	9.52
dermatology	95.6	93.9	0.38	24.08
monks	89.8	31.9	0.20	21.16
wdbc	81.3	83.1	0.19	15.53
synthetic	95.4	93.2	0.34	53.68
australian	66.5	65.8	0.31	46.85
mammograph	59.3	56.5	0.12	94.11
german	37.2	19.1	0.58	151.92
isolet	86.2	46.1	7.18	1372.87
imseg	81.4	7.4	0.11	3.28
abalone	30.0	-1.2	0.52	23.70
sat	74.5	22.6	2.07	80.06
musk	78.7	29.0	7.35	34.15
grid	81.5	73.0	1.17	88.00
nursery	76.5	59.2	1.86	170.36
arabic	21.1	3.1	263.67	193.10
magic	46.3	28.5	1.75	115.26
letter	77.2	9.9	5.16	114.48
Average	64.6	45.5	—	105.77
p-value	—	0.028	—	4.5e-07

Table 2.8: Comparison of FSVC and ICVM over the small datasets.

The FSVC was also compared with the double stochastic gradient (DSG) approach [8], an example of random feature based method [21], on the `adult` (named in [8] as `census-income`, with 50,000 patterns) and `mnist8m` datasets. Using 4-fold cross-validation, FSVC and LSVC achieve errors of 24.1% and 16.6%, respectively, while [8] reports an error of 15% for DSG, very near to LSVC and 9% below FSVC, although the experimental methodologies may differ. Concerning times, FSVC spends about 6 seconds for training, similar to LSVC and below DSG (about 8 s.). With respect to the `mnist8m` dataset, we followed the instructions[7] to download the dataset and SGD code, running SGD and FSVC on the PCA transformed training and test data (8,100,000 and 10,000 patterns, respectively, with 100 inputs). The SGD achieves an accuracy of 99.33% spending 21,755 seconds (about 6 hours) and 5.8GB of RAM, while FSVC achieves accuracy of 84.62%, spending 21 minutes 22 seconds and 6.15 MB of RAM. Although the difference in accuracy is about 15%, the FSVC is 17 times faster and requires 965 times less memory than SGD. Overall, SGD seems to achieve

[7] https://github.com/zixu1986/Doubly_Stochastic_Gradients/tree/master/matlab (March, 2022).

higher performance on these datasets, although the gap might be caused by differences in experimental methodologies, being much slower and with much larger memory requirements than FSVC.

	SVC2		KDE		SVC	
Dataset	Kappa(%)	Time(s)	Kappa(%)	Time(s)	Kappa(%)	Time(s)
fertility	30.7	0.25	10.4	0.11	-1.6	0.37
tissue	63.9	0.05	60.6	0.07	62.1	0.41
hepatitis	34.7	0.05	30.8	0.11	36.2	0.55
wine	97.5	0.03	97.5	0.08	97.5	0.65
sonar	71.8	0.51	71.8	0.18	63.2	1.07
seeds	89.2	0.03	90.6	0.08	89.1	0.58
heart	43.0	0.17	43.0	0.27	35.9	1.51
ionosphere	82.3	0.26	82.3	0.36	86.8	1.84
dermatology	89.6	0.34	94.9	0.48	95.2	6.60
monks	100.0	0.31	100.0	0.51	100.0	6.56
wdbc	89.5	0.38	89.5	0.58	87.6	3.44
synthetic	99.0	0.26	99.0	0.34	97.4	7.29
australian	60.3	0.49	60.3	0.91	70.1	7.29
mammograph	61.9	0.83	61.9	1.51	65.2	42.64
german	36.7	1.12	32.9	1.92	33.6	18.56
isolet	95.4	8.85	95.4	9.37	95.2	432.66
imseg	90.7	0.11	89.9	0.14	89.1	0.43
abalone	33.9	5.68	33.9	10.01	33.8	548.48
sat	89.2	5.18	89.2	8.15	89.8	486.63
musk	97.2	35.84	99.0	61.64	98.8	1176.59
grid	96.8	63.90	96.8	115.89	99.2	538.42
nursery	100.0	24.21	100.0	38.42	100.0	2040.47
Average	75.1	—	74.1	1.58	73.8	27.68
p-value	—	—	1.000	0.53	0.963	8.9e-04

Table 2.9: Kappa and time of SVC2, KDE and SVC for RBF kernel spread tuning on the small datasets.

2.15 Comparison with other spread tuning methods

Table 2.9 reports the kappa and time achieved by SVC on the small datasets tuning the RBF kernel spread with: a) the proposed method, version SVC2 on line 4 of Table 2.2; b) the kernel density estimation (KDE) proposed by [28]; and c) with the standard grid-search, version SVC on line 1 of Table 2.2. In the first two approaches, that are designed for binary classification, the optimal spread is estimated using the training patterns of classes 1 and 2. The kappa columns of the row "Average" report the average kappa over the datasets, while the time columns report the average ratio between the times of KDE or SVC and the time of SVC2. The row "*p*-value" reports the *p*-values of the Wilcoxon tests comparing the kappa and times of KDE and SVC to SVC2. The results show that the average performance of SVC2 is higher than KDE and SVC (75.1% vs. 74.1% and 73.8%, respectively), although the differences are

not statistically significant (p=1 and p=0.962, respectively). With respect to times, SVC2 is so fast as KDE and 27 times faster than SVC (significant difference with $p = 9 \cdot 10^{-4}$). However, it should be noted that our approach in FSVC uses a limited number of prototypes per class, so it is more scalable than KDE with the number of patterns.

2.16 Time comparison of SVC, SVC1, SVC2 and SVC3

Table 2.10 reports the time spent by SVC divided by the time spent by SVC1 (that uses efficient training, 2nd row in Table 2.2), SVC2 (that uses efficient tuning of the RBF kernel spread, 4th row in Table 2.2) and SVC3 (that uses efficient kernel, 3rd row in Table 2.2), on the small and some large datasets. The last rows report the average of these ratios and the p-value of the test comparing the times of SVC and each column. The SVC1 is faster than SVC in 17 of 25 datasets (where $t_{\mathrm{SVC}}/t_{\mathrm{SVC1}} > 1$). In the other 8 datasets (e.g., `imseg`) SVC is faster because SVC1, despite using efficient training, needs the whole training set, while SVC is sparse and only needs the support vectors. In average, SVC is 5.20 times slower than SVC1, so the time of SVC1 is about 19% of the time of SVC. Analogously, SVC2 and SVC3 spend about 3% and 16% of the time of SVC, respectively. Note also that this ratios raise fastly with the dataset size. The main speed-up (about 35 times) with respect to SVC is achieved by SVC2 (efficient tuning), with a significant difference (p=0.0023). There are some exceptions (the SVC2 ratio is below 1 in dataset `shuttle`) where the calculation of the kernel matrix $\mathbf{K}$ in line 16 of algorithm 1 is slow due to the high N and to the absence of efficient kernel calculation in SVC2. The efficient training (SVC1) and efficient kernel calculation (SVC3) speed up the SVC about 5-6 times and are not statistically significant (p=0.45 and p=0.56, respectively).

2.17 Comparison with evolutionary training set selection

Table 2.11 reports, for several small and large datasets of our collection, the kappa and time per fold of FSVC, SVC and SGA+SVC, i.e., SVC using the training set selected by steady-state genetic algorithm for instance selection (SGA), an evolutionary algorithm for training set selection [2] implemented by the `Keel` software[8]. The SGA is used to select training patterns, while the validation and test sets are left unchanged. The time of SGA is the sum of the time

[8] https://keel.es (March, 2022).

Dataset	t_{SVC}/t_{SVC1}	t_{SVC}/t_{SVC2}	t_{SVC}/t_{SVC3}
fertility	1.61	1.48	0.90
tissue	0.25	8.20	0.75
hepatitis	1.67	11.00	0.92
wine	0.82	21.67	0.84
sonar	2.14	2.10	0.93
seeds	0.62	19.33	0.84
heart	2.56	8.88	1.02
ionosphere	2.22	7.08	0.96
dermatology	0.94	19.41	1.06
monks	6.76	21.16	1.25
wdbc	2.23	9.05	0.91
synthetic	0.66	28.04	0.96
australian	4.03	14.88	1.18
mammograph	19.83	51.37	3.90
german	5.10	16.57	1.68
isolet	0.51	48.89	1.09
imseg	0.08	3.91	0.98
abalone	19.68	96.56	12.78
sat	2.67	93.94	5.47
musk	5.71	32.83	9.55
grid	7.23	8.43	6.16
nursery	7.16	84.28	12.47
magic	30.00	27.17	43.31
letter	0.49	224.16	5.50
shuttle	5.00	0.73	37.43
Average	5.20	34.44	6.11
p-value	0.45	0.0023	0.56

Table 2.10: Times spent by SVC divided by times of SVC1, SVC2 and SVC3 on the small and some large datasets.

spent by SGA to select the training patterns and by SVC to train (on the reduced training set), validate and test. The last column reports the percentage of reduction of SGA in the training set size. The results show that SGA reduces very much the size of the training set (89-98%) without a large reduction in the performance. However, the training set selection requires much more time than SVC and FSVC training due to its high computational complexity, being much slower than training and becoming the most relevant component of the elapsed time. Besides, the SGA achieves memory errors in datasets `arabic`, `mnist` and larger. In dataset `wisdm` (not included in the Table 2.11), the SGA did not finish in 5 days (120h), so we did not try in datasets larger than `fruits`. Therefore, SGA is able to select the training set while keeping the performance only in small datasets, where SVC can already be run without training selection. Besides, the low speed and high memory requirements of SGA hinders to reduce large datasets, where the selection would be really useful because SVC can not be run on the whole training set.

	Kappa(%)			Time(s)/fold			
Dataset	FSVC	SVC	SGA	FSVC	SVC	SGA	Red.(%)
abalone	30.0	33.8	32.4	0.13	1.82	1:37	96.7
sat	77.7	89.9	87.2	0.68	2:02	9:32	97.8
musk	78.7	98.8	90.1	1.84	4:54	40:30	95.7
grid	81.5	99.2	97.9	0.29	2:15	14:48	92.9
nursery	76.5	100.0	99.9	0.47	8:30	42.58	91.0
arabic	22.1	62.5	—	58.8	29:14:43	—	—
magic	46.3	71.4	69.2	0.44	1:59:22	57:06	92.9
letter	78.3	96.3	94.7	1.50	1:02:46	1:28:09	89.2
shuttle	87.6	99.8	99.6	1.58	6:03:59	9:27:20	93.7
mnist	74.9	96.9	—	2:49	28:19:22	—	—

Table 2.11: Kappa and time per fold of FSVC, SVC and SGA, and % of reduction of SGA on the training set size.

Name	Symbol	Size	Name	Symbol	Size
Train patterns	$\{\mathbf{x}_n\}_{n=1}^{B_N}$	IB_N	Class labels	$\{c_n\}_{n=1}^{B_N}$	B_N
Prototypes	$\{\mathbf{p}_{ql}\}_{ql=1}^{Q,L}$	QLI	Number of prototypes	$\{N_{ql}\}_{ql=1}^{Q,L}$	QL
Ideal kernel	$\{J_{lm}\}_{lm=1}^{2L}$	$(2L)^2$	Distances	$\lvert\mathbf{p}_{1l}-\mathbf{p}_{2m}\rvert$	$(2L)^2$
Train kernel	$\{K_{lm}\}_{lm=1}^{2L}$	$(2L)^2$	Test patterns	$\{\mathbf{x}_n\}_{n=1}^{B_T}$	IB_T
Test kernel	$\{k_q(\mathbf{x}_n)\}_{qn=1}^{Q,B_T}$	QB_T	Weights	$\{z_{ns}\}_{ns=1}^{B_T,S}$	SB_T
Votes	$\{v_{nq}\}_{nq=1}^{B_T,Q}$	QB_T	Output	$\{y_n\}_{n=1}^{B_T}$	B_T

Table 2.12: Memory required by the largest data matrices used by FSVC with RBF kernel and OVO approach, being $S = Q(Q-1)/2$.

2.18 Memory usage

The FSVC reads the training and test patterns from file disks by blocks in order to reduce the number of disk reads. A low block size V means smaller matrices, that may speed up the mathematical operations, but also more disk reads, that may slow the execution down. On the contrary, high V means larger matrices, that may slow down the calculations, but also less disk reads, that may speed up the execution. Besides, it also requires more memory M, limited by the available memory $\mathfrak{M}$. In principle, the weight of each component (low or high V) on the execution speed is unknown. In order to estimate M from V, Table 2.12 reports the sizes of the largest data matrices used by FSVC with RBF kernel and OVO multi-class approach. Each training pattern has I inputs and the class label, so its size is $I+1$. Given V, the default number Y of patterns per block is $Y = \max(1, \lfloor \frac{V}{I+1} \rfloor)$, in such a way that $Y = 1$ when $I+1 > V$, that might happen for high I. The number of training patterns per block is $B_N = \min(N, Y)$, because it must be $B_N = N$ when $N < Y$, i.e., the number B_N of training

patterns per block can not overcome the number N of available training patterns. The train pattern block is of size IB_N (first item in Table 2.12). The class labels is a vector of size B_N.

Considering $N_q > L$ for $q = 1,\ldots,Q$, for large datasets, the QL prototypes, each of length I, have a size of QLI. The number of prototypes for the Q classes is a matrix of size QL. Considering the RBF kernel spread tuning, in multi-class problems σ is the same for all the FSVCs, being selected using prototypes of classes 1 and 2. Thus, the ideal kernel matrix $\mathbf{J}$, the distances $|\mathbf{p}_{1l} - \mathbf{p}_{2m}|$ between prototypes l and m of both classes and the kernel matrix $\mathbf{K}$ (eq. 2.21) have $(2L)^2$ values each one. With T test patterns, the number of test patterns per block is $B_T = \min(T,Y)$, so that $B_T = T$ when $T < Y$. The block of test patterns $\{\mathbf{x}_n\}_{n=1}^{B_T}$ is of size IB_T. The test kernel values k_{qn} (eq. 2.18):

$$k_{qn} = k_q(\mathbf{x}_n) = \sum_{l=1}^{L_q} k_{ql}(\mathbf{x}_n), \quad q = 1,\ldots,Q; n = 1,\ldots,B_T \tag{2.26}$$

are QB_T values. The test weights z_{ns} for classes $q(s)$ and $r(s)$:

$$z_{ns} = \frac{k_{qn}}{L_q} - \frac{k_{rn}}{L_r} + b_s, \quad n = 1,\ldots,B_T; s = 1,\ldots,S \tag{2.27}$$

(see eq. 2.18) require SB_T values, where $S = Q(Q-1)/2$, being q and r the second and first class of the s-th binary FSVC in an OVO approach. The class votes v_{nq} (line 19 in algorithm 2) for the n-th test pattern are given by:

$$v_{nq} = \sum_{s\in\mathcal{A}_q} z_{ns} - \sum_{s\in\mathcal{B}_q} z_{ns}, \quad n = 1,\ldots,B_N; q = 1,\ldots,Q \tag{2.28}$$

where $\mathcal{A}_q$ (resp. $\mathcal{B}_q$) is the set of binary FSVCs where class q is the first (resp. second). Finally, the output y_n of the multi-class FSVC is:

$$y_n = \underset{q=1,\ldots,Q}{\arg\max}\{v_{nq}\}, \quad n = 1,\ldots,B_T \tag{2.29}$$

which requires B_T values. Algorithm 5 describes the procedure to select the block size V given the available memory ($\mathfrak{M}$) and the dataset size, defined by N, I, Q and T. The FSVC sets a starting value V=10^6, then it calculates Y, B_N and B_T. Using Table 2.12, the required memory $M_0 = M(B_N,I,Q,L,B_T)$ can be estimated as:

$$\begin{aligned} M(B_N,I,Q,L,B_T) &= IB_N + B_N + QLI + QL + 3(2L)^2 + IB_T + QB_T + SB_T + QB_T + B_T = \\ &= (I+1)B_N + QL(I+1) + 12L^2 + \frac{Q^2+3Q+2(I+1)}{2}B_T \end{aligned} \tag{2.30}$$

Algorithm 5: Selection of the memory data block size V.

1 **Algorithm:** BlockSize($\mathfrak{M}$,N,I,Q,L,T)

Data: $\mathfrak{M}$: available memory; N: no. train patterns; I: no. inputs; Q: no. classes; L: max. prototypes/class; T: no. test patterns
Result: V: data block size

2 $\varepsilon \leftarrow 0.1$; $\eta \leftarrow 0.25$; $V \leftarrow 10^6$
3 **while true do**
4 $\quad Y \leftarrow \max\left(1, \left\lfloor \frac{V}{I+1} \right\rfloor\right)$; $B_N \leftarrow \min(N,Y)$; $B_T \leftarrow \min(T,Y)$
5 $\quad M_0 \leftarrow M(B_N,I,Q,L,B_T)$ // M= memory required according to eq. 2.30
6 $\quad$ **if** $Y = 1$ **then**
7 $\quad\quad$ **if** $M_0 > \mathfrak{M}$ **then**
8 $\quad\quad\quad$ **error**: Block with just 1 pattern does not fit in memory
9 $\quad\quad$ **end**
10 $\quad\quad$ **break**
11 $\quad$ **end**
12 $\quad$ **if** $M_0 < \eta\mathfrak{M}$ **then**
13 $\quad\quad$ **break**
14 $\quad$ **end**
15 $\quad V \leftarrow (1-\varepsilon)V$
16 **end**

Let us $\eta < 1$ be the percentage of the available memory $\mathfrak{M}$ that can be used. When $M_0 < \eta\mathfrak{M}$, the current V is accepted as valid block size. Otherwise, V is reduced to $(1-\varepsilon)V$, with $\varepsilon = 0.1$. A default value for η=0.25 can be used, that uses 25% of memory, although a higher percentage might also be used. It may happen, specially in high-dimensional datasets, that $V < I+1$, so that $Y = 1$, i.e., only a pattern is read each time because it is very large. In this case, V is accepted even if $M_0 \geq \eta\mathfrak{M}$ in order to guarantee that $B_N, B_T \geq 1$, i.e., the training and test block have at least one pattern (line 10 in algorithm 5). The process continues iteratively, updating V, Y, B_N, B_T and M_0 until either $M_0 < \eta\mathfrak{M}$ (line 13) or $Y = 1$. A very extreme case would be that $Y = 1$ and $M_0 > \mathfrak{M}$ because a data block with only one pattern ($Y = 1$) does not fit in the available memory (line 8). This might only happen with very high I and/or Q, that can not be reduced as B_N or B_T, for the avaliable memory, and in this case the algorithm would fail to select an adequate V.

Table 2.13 (columns labeled as "$\mathfrak{M}$=32GB") reports the time and memory (measured using the Octave `whos` command) spent by FSVC, and the memory required by LSVC on the

large datasets, alongside with their populations (N). In order to ensure that FSVC reduces V in response to the memory restrictions (see the next paragraph) according to algorithm 5, the starting value was raised from $V = 10^6$ (default value) to $V = 10^7$. The LSVC requires more memory than FSVC in all the datasets (columns 4 and 5) excepting `letter`, overcoming 1GB in 6 out of 22 datasets and failing by lack of memory on `fruits`, `har`, `kitsune` and `wesad`. In average (last row), LSVC requires 12.5 times more memory (1.5GB) than FSVC (191MB). The FSVC only overcomes 1GB on `fruits`, due to its high Q=131 and I=30,000, and `human`, due to its high Q=42.

		$\mathfrak{M}$=32GB			$\mathfrak{M}$=2GB		
		FSVC		LSVC	FSVC		LSVC
Dataset	N	Time	Mem.	Mem.	Mem.	Time	Time
arabic	16,800	1:52	73M	132M	73M	1:53	—
magic	19,020	0.54	2M	2M	2M	0.46	0.40
letter	20,000	1.48	17M	3M	17M	1.38	0.74
shuttle	58,000	1.71	6M	6M	6M	1.53	1.01
mnist	70,000	3:01	74M	419M	74M	3:08	—
wisdm	73,803	30.45	43M	52M	43M	29.10	26.23
fruits	90,380	8:33:41	1.5G	—	338M	2:57:33	—
devanagari	92,000	14:13	152M	720M	152M	12:57	—
ijcnn1	141,691	8.26	8M	31M	8M	7.59	4.43
covtype	581,012	1:44	104M	245M	104M	1:44	27.51
poker	1,025,010	48.94	59M	94M	59M	44.17	21.13
kddcup	4,000,000	35:33	225M	3.7G	225M	35:35	—
susy	5,000,000	4:28	70M	734M	70M	4:22	—
record	5,749,132	3:29	109M	537M	109M	3:39	—
physical	6,000,000	23:57	88M	5.8G	88M	23:14	—
baiot	7,062,606	55:35	118M	6.1G	118M	56:10	—
hepmass	10,500,000	13:39	80M	2.3G	80M	14:11	—
higgs	11,000,000	14:03	138M	2.4G	138M	14:09	—
human	13,956,557	1:30:59	2.0G	3.9G	509M	55:10	—
kitsune	21,017,597	2:22:29	192M	—	192M	2:22:46	—
har	29,097,887	25:56	330M	—	330M	25:33	—
wesad	31,470,603	15:58	252M	—	252M	15:49	—
Average		14:40	191M	1.5G	104M	12:39	

Table 2.13: Time(in sec./fold) and memory (in MB or GB) required by FSVC and LSVC on the large datasets with 32GB and 2GB of available memory ($\mathfrak{M}$).

An additional experiment executed FSVC and LSVC on the large datasets using the same computer but with only $\mathfrak{M}$=2GB, instead of $\mathfrak{M}$=32GB, in order to show how FSVC uses algorithm 5 to adjust V until $M_0 < \eta\mathfrak{M}$ or $Y = 1$. In order to perform this, we ran the Linux command `ulimit -Sv 2097152`, in order to set a soft limit of 2,097,152 KB (2 GB) on the virtual memory before running FSVC and LSVC. Last three columns of Table 2.13, under label "$\mathfrak{M}$=2GB", report the time and memory spent by FSVC and the time spent by LSVC, which fails by lack of memory on 15 out of 22 large datasets. The FSVC reduces the average

memory from 191MB to 104MB (columns 4 and 6 in the last row). Indeed, the memory required by FSVC does not change excepting on the `fruits` dataset, where L=10 was used due to the large I=30,000 and to the constraint $\mathfrak{M}$=2GB, reducing from 1.5GB to 338MB, and `human`, where the high Q reduces V from 10^7 to $2.3{\cdot}10^6$, and the required memory from 2GB (still much lower than LSVC, that requires 3.9GB on this dataset) to 509MB.

Using 2GB, the times of FSVC are surprisingly reduced on `fruits` (2.8 times, from 8h 33m 41s to 2h 57m 33s. In this dataset, the high initial time compared to Table 2.6 is caused by the high value $V = 10^7$. The reduction in time is caused by the decreasing from $L = 50$ to $L = 10$. The time is also reduced in `human` (1.6 times), because the lower block size V=$2.3{\cdot}10^6$ allows smaller matrices and faster calculations. The average time of FSVC (columns 3 and 7 of the last row) decreased from 14m 40s using $\mathfrak{M}$=32GB to 12m 39s using $\mathfrak{M}$=2GB, so the reduction in memory does not slow down FSVC necessarily.

CHAPTER 3

IDEAL KERNEL TUNING

The support vector classifier (SVC) is a very popular classification method [5] with state-of-the-art performance, particularly combined with the RBF kernel [12]. Before the training stage, two hyper-parameters must be set: regularization (C) and kernel spread (σ), that measures how large the differences between patterns must be to drive the kernel function towards zero. The optimal σ value depends on the data properties and is, in general, different for each dataset. Sub-optimal values often lead to an important loss in the SVC performance. The spread tuning is usually performed using the "grid-search" strategy, that performs training with several σ values from a pre-specified collection and selects the value with the best average performance on a collection of separated validation sets. The experimental literature has proven that the collection of values for C and σ proposed in [20] is adequate for a vast majority of the applications. This fact highly simplifies the SVC usage because only a repetitive train-test loop must be performed, and no expert knowledge is required to achieve a fine tuning. However, to train and test the SVC many times is very time-consuming for large datasets.

3.1 Related work

The literature about "model selection" for SVC reports several approaches for spread selection alternatives to grid-search. The kernel density estimation (KDE) selects the σ value that maximizes a function of disimilarity between two classes that is based on the RBF kernel [28] and does not require to train the SVC. Genetic algorithms [14] have been used to select C and σ by optimizing the test performance over small datasets (in the paper, up to 768 training

patterns), thus requiring several SVC trainings. Genetic optimization [37] is also used to find the hyper-parameter values that maximize the relative efficiencies of binary SVCs in multi-class problems up to 4,400 training patterns. Evolutionary algorithms are used in [36] to perform surrogate-assisted multi-objective minimization of bias and variance, estimated by training and testing the SVC. The Bat algorithm [43] is a bio-inspired method based on genetic optimization that iteratively searches for C and σ, selecting the values that minimize the classification error of the SVC. Despite of exhibit early convergence compared to other meta-heuristic methods, this method requires to train and test the SVC inside the optimization loop, that reduces its computational efficiency and hinders its scalability for large datasets (only up to 1,000 training patterns and 60 inputs). The same authors combine the social ski driver algorithm and the synthetic minority over-sampling technique (SMOTE) to select σ for imbalanced datasets [42]. A dynamic strategy based on particle swarm optimization is proposed in [22] for the simultaneous selection of C and σ on datasets up to 7,000 training patterns and 256 inputs. Segmented dichotomy is combined with grid-search [41] to select both hyper-parameters, but it also requires to train and test the SVC iteratively, so it was applied to medium size datasets up to 3,000 training patterns. Active testing was used in [29] to select both hyper-parameters from dataset properties such as training set size, number of inputs and classes, mean correlation among inputs, skewness and kurtosis of the inputs, and class entropy, although the SVC execution is also required for each candidate solution. A method based on the Fisher criterion is proposed in [26] to select both hyper-parameters in order to maximize between-class similarity and minimize within-class separability, estimated using cosine similarity in the kernel space.

Similarly to grid-search, most of the previous approaches require to train and test the SVC several times, often under the scope of some optimization strategy that may not scale well with the training set size. The goal of the current paper is to develop a method able to find the optimal σ for a binary SVC directly from the training data (patterns and class labels) without the need to train and test the SVC. Our proposal, named ideal kernel tuning (IKT), is to select the σ value that brings the RBF kernel the closest possible to the ideal kernel [15]. The basic idea of this method, combined to other algorithmic contributions for large-scale classification, was already used in the fast support vector classifier. The current chapter formulates the IKT method alone, without any other elements, for the tuning of the RBF kernel spread in the standard SVC applied to small and medium sized datasets. As well, this study evaluates the performance and speed of IKT by performing exhaustive numerical

experiments on benchmark datasets and performing a comparison with other five state-of-the-art approaches for the model selection.

3.2 Materials and methods

The kernel function $K(\mathbf{x},\mathbf{y})$ is a generalized similarity measure between two patterns $\mathbf{x}$ and $\mathbf{y}$. This similarity can be considered as their dot product after being projected into a high-dimensional feature space by a kernel mapping $\Phi(\mathbf{x})$, so that $K(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})^T\Phi(\mathbf{y})$. The RBF kernel function over two patterns $\mathbf{x}$ and $\mathbf{y}$ is a Gaussian function of spread σ applied over the difference between both patterns:

$$K(\mathbf{x},\mathbf{y}) = \exp\left(-\frac{|\mathbf{x}-\mathbf{y}|^2}{2\sigma^2}\right) \tag{3.1}$$

where the spread σ must be set before the SVC training stage. Let us consider a binary classification problem with Q=2 classes, N training patterns $\mathbf{x}_n$ of dimensionality I and class label $c_n \in \{1,\ldots,C\}$, with $n = 1,\ldots,N$. The RBF kernel matrix $\mathbf{K}$ is a N-order square matrix with elements $\mathcal{K}_{nm} = K(\mathbf{x}_n,\mathbf{x}_m)$, with $n,m = 1,\ldots,N$, being K defined by eq. 3.1. On the other hand, the ideal kernel [33] matrix $\mathbf{J}$, also square of order N, has elements J_{nm} defined by:

$$J_{nm} = \begin{cases} 1 & c_n = c_m \\ 0 & \text{otherwise} \end{cases}, \quad n,m = 1,\ldots,N \tag{3.2}$$

The ideal kernel defines a class-based generalized similarity that discriminates perfectly both classes, but its analytical expression is unknown and different for each dataset. The concept of ideal kernel matrix is useful because it defines how a kernel matrix should be in order to achieve the perfect classification. Selecting the σ that provides the RBF kernel matrix nearest to the ideal kernel matrix is expected to provide the optimal classification performance. This is achieved by minimizing the mean absolute difference between matrices $\mathbf{K}$ and $\mathbf{J}$. The ideal kernel tuning (IKT) algorithm, that is detailed in the following paragraphs and in algorithm 6, is based on this idea.

In principle, this procedure is designed for binary classification ($Q = 2$). However, the multi-class SVC uses several binary SVCs with equal σ. This value can be selected considering only the training patterns of the two most populated classes (denoted as a and b), and it can be used for all the binary SVCs in a multi-class classification problem ($Q > 2$). Considering

Algorithm 6: Ideal kernel tuning.

Algorithm: IKT$(\{\mathbf{x}_n, c_n\}_{n=1}^{N})$

Data: $\{\mathbf{x}_n, c_n\}_{n=1}^{N}$, $\mathbf{x}_n$; $c_n \in \{1,\ldots,Q\}$
Result: σ^*: optimal RBF kernel spread

$\{N_i \leftarrow 0\}_{i=1}^{C}$; $L \leftarrow 100$; $\{\mathbf{p}_{il} \leftarrow \mathbf{0}; L_i, N_{il} \leftarrow 0\}_{il=1}^{C,L}$
for $n \leftarrow 1, N$ **do**
  $i \leftarrow c_n$; $N_i \leftarrow N_i + 1$
  if $L_i < L$ **then**
    $l \leftarrow L_i$; $\mathbf{p}_{il} \leftarrow \mathbf{x}_n$; $L_i \leftarrow L_i + 1$
  else
    $r \leftarrow \arg\min_{l=1\ldots,L} |\mathbf{p}_{il} - \mathbf{x}_n|$; $N_{ir} \leftarrow N_{ir} + 1$
    $\mathbf{p}_{ir} \leftarrow \left(1 - \frac{1}{N_{ir}}\right)\mathbf{p}_{ir} + \frac{\mathbf{x}_n}{N_{ir}}$
  end
end
$a \leftarrow \arg\max_{i=1,\ldots,C}\{N_i\}$; $b \leftarrow \arg\max_{i=1,\ldots,C, i\neq a}\{N_i\}$
$U \leftarrow L_a + L_b$; $\{J_{ls} \leftarrow 0\}_{ls=1}^{U}$
$\{J_{ls} \leftarrow 1\}_{ls=1}^{L_a}$; $\{J_{ls} \leftarrow 1\}_{ls=L_a+1}^{U}$
$\{\mathbf{z}_l \leftarrow \mathbf{p}_{al}\}_{l=1}^{L_a}$; $\{\mathbf{z}_l \leftarrow \mathbf{p}_{b(l-L_a)}\}_{l=L_a+1}^{U}$
$\left\{\mathcal{K}_{ls}(\sigma) \leftarrow \exp\left(\frac{-|\mathbf{z}_l - \mathbf{z}_s|^2}{2\sigma^2}\right)\right\}_{ls=1}^{U}$
$E \leftarrow 19$; $\alpha \leftarrow 2^{15/2}$; $\left\{\sigma_q \leftarrow \alpha 2^{-q/2}\right\}_{q=1}^{E}$
$\left\{D_q \leftarrow \frac{1}{U^2}\sum_{l=1}^{U}\sum_{s=1}^{U}|\mathcal{K}_{ls}(\sigma_q) - J_{ls}|\right\}_{q=1}^{E}$
$j \leftarrow \arg\min_{q=1,\ldots,E}\{D_q\}$; $\delta \leftarrow \frac{|D_E - D_1|}{10}$
if $|D_j - D_1| < \delta$ **or** $|D_E - D_j| < \delta$ **then**
  $F_1 \leftarrow 0$; $\{F_q \leftarrow |D_q - D_{q-1}|\}_{q=2}^{E}$; $j \leftarrow \arg\max_{q=1,\ldots,E}\{F_q\}$
end
$\sigma^* \leftarrow \sigma_j$

only the N_a and N_b training patterns of classes a and b, respectively, both the RBF and ideal kernel matrices $\mathbf{K}$ and $\mathbf{J}$ are of size M^2, with $M = N_a + N_b$. The calculation of this matrix becomes slow and drives out of memory in large datasets with high N_a and N_b. In order to

avoid this drawback, the M training patterns of classes a and b are replaced by a reduced set of class prototypes $\{\mathbf{p}_{il}\}_{l=1}^{L_i}$, with $L_i = \min(N_i, L) \leq L$, where N_i is the number of patterns of class $i \in \{a,b\}$, while L is an upper bound to the number of prototypes per class. The value of L is set low enough to guarantee that $U = L_a + L_b \leq 2L \ll M$, in order to keep the number U of prototypes much less than M when N_a or N_b are high. Thus, both the RBF and ideal kernel matrices $\mathbf{K}$ and $\mathbf{J}$ are of size U^2.

To calculate the class prototypes, the first L training patterns $\mathbf{x}_n$ of class i are stored as starting prototypes $\{\mathbf{p}_{il}\}_{l=1}^{L}$. When there are $N_i < L$ patterns of class i, only N_i prototypes (that are the N_i training patterns of class i) are created, so that $L_i = N_i$ and no prototype is updated. When $N_i \geq L$, the first L training patterns of class i initialize the L prototypes of this class. Afterwards, each new training pattern $\mathbf{x}_n$ of class $i = c_n$ updates its nearest prototype $\mathbf{p}_{ir}$ according to:

$$\begin{aligned} \mathbf{p}'_{ir} &= \left(1 - \frac{1}{N_{ir}}\right)\mathbf{p}_{ir} + \frac{\mathbf{x}_n}{N_{ir}}, \quad i = c_n \\ r &= \arg\min_{l=1\ldots,L} |\mathbf{p}_{il} - \mathbf{x}_n|, \quad N'_{ir} = N_{ir} + 1 \end{aligned} \tag{3.3}$$

where $\mathbf{p}_{ir}$ and $\mathbf{p}'_{ir}$ are the old and new versions, respectively, of the r-th prototype of class i, which is the nearest prototype to $\mathbf{x}_n$ of class i, while N_{ir} and N'_{ir} are the old and new numbers of training patterns used to update $\mathbf{p}_{ir}$. Even though the first L training patterns of a class i were similar among them, the following patterns of this class will update their nearest prototypes of class i and, finally, the prototype collection $\{\mathbf{p}_{il}\}_{l=1}^{L}$ of this class will represent its training patterns (those $\mathbf{x}_n$ with $c_n = i$).

Although the differences $|\mathbf{p}_{il} - \mathbf{x}_n|$ must be calculated for each training pattern and the L_i prototypes of class $i = c_n$, the number L_i is upper bounded by L, so the number of calculations keeps low for large datasets. Besides, matrices $\mathbf{K}$ and $\mathbf{J}$ are small because their size U^2 verifies $U^2 \leq (2L)^2$. After some previous experiments, we saw that $L = 100$ prototypes is enough to represent a class in large datasets where $L_a = L_b = L$ and $U = 2L$, keeping the kernel matrix size below $(2L)^2 = 40{,}000$. However, when the number of inputs I or classes Q is large, a lower L should be set to avoid slowness and excesive memory requirements. The elements $\{\mathcal{K}_{ls}\}_{ls=1}^{U}$ of the kernel matrix $\mathbf{K}$ are calculated as:

$$\mathcal{K}_{ls}(\sigma) = K(\mathbf{z}_l, \mathbf{z}_s) = \exp\left(-\frac{|\mathbf{z}_l - \mathbf{z}_s|^2}{2\sigma^2}\right) \tag{3.4}$$

where the vectors $\{\mathbf{z}_l\}_{l=1}^{U}$ are defined by:

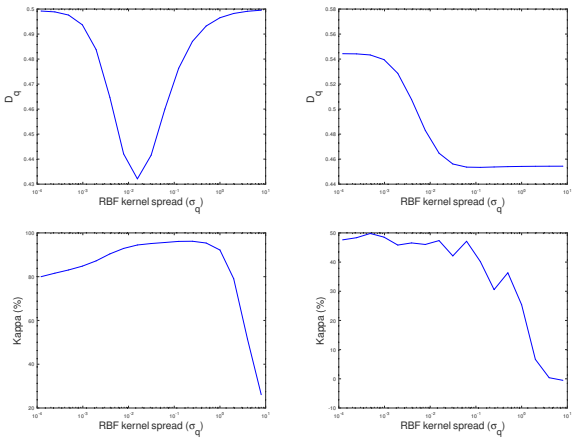

Figure 3.1: Difference D_q (upper panels) and performance (lower panels) vs. σ_q for datasets `letter` (left panels) and `pima` (right panels).

$$\mathbf{z}_l = \begin{cases} \mathbf{p}_{al} & l = 1,\ldots,L_a \\ \mathbf{p}_{b(l-L_a)} & l = L_a+1,\ldots,U \end{cases} \tag{3.5}$$

while $\{\mathbf{p}_{al}\}_{l=1}^{L_a}$ and $\{\mathbf{p}_{bl}\}_{l=1}^{L_b}$ are the prototypes of classes a and b, and $U = L_a + L_b$. The ideal kernel matrix $\mathbf{J}$, also of size U^2, has elements defined by $J_{ls} = 1$ for $l,s \in \{1,\ldots,L_a\}$ or $l,s \in \{L_a+1,\ldots,U\}$, i.e. both prototypes l and s are of the same class, and $J_{ls} = 0$ otherwise, i.e. both prototypes are of different classes. The IKT selects σ to minimize the mean absolute difference between $\mathbf{K}$ and $\mathbf{J}$. In order to avoid a computationally intensive optimization, the search is limited to the $E = 19$ values defined by the Libsvm library guide [20] as $\sigma_q = \alpha 2^{-q/2}$, with $\alpha = 2^{15/2}$ and $q = 1,\ldots,E$. This collection of σ values is widely used in the literature because it covers a wide range of σ values. Thus, it is expected for any arbitrary dataset that the SVC with RBF kernel using some spread value from this collection will achieve a performance very near to the best attainable by this classifier. Note also that α and E are just two convenience notations for $2^{15/2}$ and for the number (19) of values in the Libsvm collection of spread values, respectively. The difference D_q between matrices $\mathbf{K}$ and $\mathbf{J}$ for σ_q is calculated as:

$$D_q = \frac{1}{U^2}\sum_{l=1}^{U}\sum_{s=1}^{U} |\mathcal{K}_{ls}(\sigma_q) - J_{ls}|, \quad q = 1,\ldots,E \tag{3.6}$$

and the optimal spread value σ^* is set to σ_j, being j the index that minimizes $\{D_q\}_{q=1}^{E}$:

$$\sigma^* = \sigma_j, \qquad j = \underset{q=1,\ldots,Q}{\arg\min}\{D_q\} \tag{3.7}$$

Table 3.1: Classification problems: total (P) and training (N) patterns, inputs (I) and classes (Q).

No.	Dataset	Train+Test P	Train N	Inputs I	Classes Q
1	tissue	106	58	9	6
2	hepatitis	155	78	24	2
3	wine	178	90	13	3
4	sonar	208	106	60	2
5	seeds	210	108	7	3
6	heart	270	136	20	2
7	voting	342	218	32	2
8	ionosphere	350	178	33	2
9	dermatology	366	186	98	6
10	german	366	500	44	2
11	monks	432	216	17	2
12	wdbc	569	286	30	2
13	synthetic	600	300	60	6
14	australian	690	346	35	2
15	energy	768	386	8	3
16	pima	768	384	8	2
17	vehicle	846	426	18	4
18	annealing	886	450	52	5
19	tic	958	480	18	2
20	mammograph	961	482	5	2
21	isolet	1,559	780	617	26
22	imseg	2,086	140	18	7
23	abalone	4,177	2,090	9	3
24	sat	6,435	3,222	36	6
25	musk	6,598	3,302	166	2
26	usps	9,298	4,650	256	2
27	grid	10,000	5,000	13	2
28	nursery	12,958	6,480	27	4
29	magic	19,020	9,510	10	2
30	letter	20,000	10,018	16	26
31	chess	28,056	14,044	40	18
32	adult	48,842	24,422	105	2
33	shuttle	57,977	43,483	9	5
34	mnist	70,000	60,000	784	10
35	wisdm	73,803	55,380	92	18
36	ijcnn1	141,691	70,848	22	2
37	poker	1,025,010	25,010	10	2

Note that the calculation of $\{D_q\}_{q=1}^{E}$ and the selection of j in eq. 3.7 is much faster than a direct calculation of σ in order to mimimize $D(\sigma)$, that may strongly depend on the dataset properties, and not only on the dataset size. Figure 3.1 plots $\{D_q\}_{q=1}^{E}$ in the upper panel and the classification performance, measured by the Cohen kappa statistic [27], in the lower panel for two datasets (`letter` and `pima`) included in the experimental work (section 3.3). The left profile is the most common, with a minimum of D_q (upper left) in the values of σ_q where the highest performance (lower left) is attained. However, in some datasets D_q looks like the

upper right plot, where the minimum D_q is very near to D_1 or D_E, and the highest performance (lower right) is achieved in the σ_q with the highest slope of D_q in absolute value (upper right). In these cases, $|D_j - D_1| < \delta$, when the minimum is located near D_1, or $|D_E - D_j| < \delta$, when the minimum is located near D_E, being j calculated in eq. 3.7 and $\delta = |D_E - D_1|/10$ a threshold (lines 20-22 of algorithm 6). When this happens, the absolute value of the derivative of $\{D_q\}_{q=1}^{E}$, denoted as $\{F_q\}_{q=1}^{E}$, is calculated as $F_1 = 0$ and $F_q = |D_q - D_{q-1}|$ for $q = 2, \ldots, E$, and the optimal spread σ^* is set to σ_j, being j the index that maximizes $\{F_q\}_{q=1}^{E}$:

$$\sigma^* = \sigma_j, \qquad j = \underset{q=1,\ldots,E}{\arg\max}\{F_q\} \tag{3.8}$$

Since IKT is based on the kernel matrix for two-class classification problems, it is also appliable to kernel types other than RBF, such as polinomial kernels, and to other kernel-based methods, such as Kernel PCA [50]. Also, IKT can only be applied to select the values of the kernel hyper-parameters, such as the RBF spread or the offset and degree of a polinomial kernel, but not to select other SVC hyper-parameters, such as the regularization C, that are not related to kernel.

The size L of the kernel matrix $\mathbf{K}$ in IKT does not depend on the dimensionality I of the training patterns, so that IKT scales well with I, although of course the calculation of $\mathcal{K}_{nm} = K(\mathbf{x}_n, \mathbf{x}_m)$ is slower when I raises. In large-scale datasets with high N, the RBF spread can also be selected using IKT due to the upper limit $U \leq 2L$ (see line 16 of algorithm 6) for the size of the kernel matrix $\mathbf{K}$, caused by the limited number of prototypes (eq. 3.3). However, the SVC training is very slow when N is high and becomes not practical despite of the efficiency of IKT to select the spread.

3.3 Results and discussion

The proposed method ideal kernel tuning (IKT) was programmed in the Octave[1] scientific computing language version 5.2. The σ selected by IKT is used to train and test the SVC implemented by the Libsvm library [3] using 4-fold cross validation with 3 folds for training (excepting grid-search, see below) and 1 fold for test in each trial. We compared IKT with five state-of-the-art methods for selecting the RBF kernel spread:

1. Kernel density estimation (KDE), implemented in Octave and applied on the two most populated classes [28].

[1] http://www.octave.org (March, 2022).

Table 3.2: Kappa (in %) achieved by each method and dataset.

Dataset	IKT	KDE	GS	GA	PSO	Bayes
tissue	58.0	64.9	61.1	59.6	61.7	64.8
hepatitis	39.3	35.4	42.4	30.4	30.4	31.5
wine	97.5	95.0	97.5	96.6	96.6	95.8
sonar	76.9	71.8	68.9	77.9	58.5	76.0
seeds	91.3	90.5	90.6	88.5	89.8	89.2
heart	53.5	53.5	55.7	61.8	67.0	66.2
voting	89.8	89.8	88.9	87.3	86.0	89.4
ionosphere	84.1	83.5	81.7	84.7	87.1	83.5
dermatology	95.6	95.6	96.6	95.9	96.2	95.9
german	28.1	29.1	28.5	34.0	38.3	41.3
monks	100.0	100.0	100.0	100.0	100.0	100.0
wdbc	89.1	88.4	86.9	85.1	84.8	88.6
synthetic	99.2	99.4	98.8	99.0	99.0	98.4
australian	63.1	63.1	57.8	64.4	67.9	67.2
energy	90.4	90.4	92.3	91.2	92.9	92.5
pima	30.2	40.4	28.0	38.0	47.3	45.5
vehicle	78.1	78.1	76.5	79.4	75.4	75.6
annealing	98.6	98.3	98.1	97.8	97.2	97.5
tic	97.2	97.7	97.2	97.5	97.0	96.8
mammograph	62.1	64.1	46.4	66.3	63.5	63.9
isolet	95.0	95.1	94.5	79.4	93.6	93.6
imseg	90.7	89.9	91.0	89.3	83.9	89.6
abalone	32.9	32.6	23.6	32.6	32.6	33.0
sat	89.4	89.2	89.0	89.5	89.5	89.1
musk	98.9	99.2	98.7	99.0	99.0	99.1
usps	97.4	97.3	96.8	90.6	95.7	97.3
grid	96.8	97.9	97.6	98.1	98.0	98.7
nursery	100.0	100.0	100.0	100.0	100.0	100.0
magic	71.0	71.0	63.3	71.0	70.9	70.2
letter	96.8	96.8	97.0	97.4	97.4	97.4
chess	89.0	88.6	88.8	89.3	89.4	89.3
adult	52.9	53.6	18.0	–	56.0	56.0
shuttle	99.7	99.7	99.8	–	99.8	99.8
mnist	83.5	83.5	80.7	–	–	80.7
wisdm	89.1	75.0	90.3	–	–	90.3
ijcnn1	95.4	–	81.8	–	–	95.7
poker	13.4	17.6	21.8	–	22.9	21.8
Average	78.8	78.2	76.4	79.7	78.4	80.0

2. Grid-search (GS), also implemented in Octave, that selects the spread σ with the highest performance over a validation set with the 50% of the three training folds, randomly selected respecting the relative class populations.

3. Genetic algorithm (GA), implemented in the R statistical computing language[2] by the `GA` package.

4. Particle-swarm optimization (PSO), implemented by the `pso` package in R.

[2] http://www.r-project.org (March, 2022).

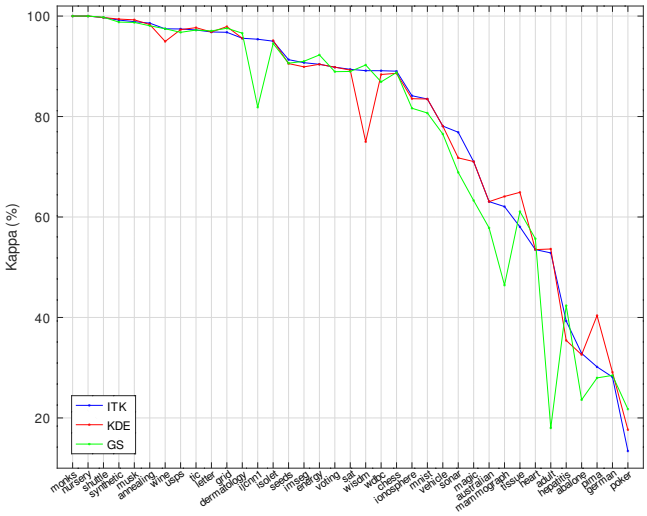

Figure 3.2: Kappa (in %) achieved by IKT, KDE and GS.

5. Bayesian optimizer (Bayes), implemented by the `rBayesianOptimization` package [43], also in R.

The methods KDE and GS optimize the classification performance of the SVC implemented by the Libsvm library in Octave, while GA, PSO and Bayes use the `svm` function of the `kernlab` R package. All the methods use the SVC with the same regularization hyper-parameter $C = 100$, named `cost` in `kernlab`, so its value was not tuned. This choice is the same for all the methods, so it does not condition the fair comparison between them, but avoids possible interactions between tuning of both hyper-parameters. Since IKT, KDE and GS are implemented in Octave, while GA, PSO and Bayes are implemented in R, the comparison of the elapsed times among algorithms might seem somehow conditioned by the use of different programming languages. However, we must think that there is no *a priori* reason to suggest that one language is faster or slower than the other. In fact, it is common in the literature to compare the performance and speed of methods programmed in different languages. On the other hand, translating an algorithm to other language (e.g., the methods in Octave to R, or vice versa) would not be convenient because the new implementation might be sub-optimal in terms of speed or have programming errors, so the speed of the method

or the correctness of the results might be compromised. The best choice is then to compare the original algorithm implementations provided by their authors, whenever available, under the hypothesis that language efficiencies are similar and that algorithms are optimally coded in each language, so the differences in performance and speed are caused exclusively by the algorithms themselves.

Table 3.3: Time (in s.) spent by each method and dataset.

Dataset	IKT	KDE	GS	GA	PSO	Bayes
tissue	0.005	0.010	0.023	2.0	0.9	535
hepatitis	0.010	0.035	0.048	3.2	1.2	319
wine	0.009	0.028	0.038	2.5	1.0	220
sonar	0.032	0.060	0.079	7.3	2.5	152
seeds	0.008	0.039	0.025	2.0	0.9	196
heart	0.021	0.086	0.063	6.7	2.2	128
voting	0.031	0.212	0.075	9.1	2.7	137
ionosphere	0.035	0.143	0.080	7.9	3.1	230
dermatology	0.052	0.085	0.324	29.8	8.8	23.3
german	0.059	0.540	0.573	48.5	11.6	19.8
monks	0.048	0.245	0.350	8.7	4.5	140
wdbc	0.055	0.211	0.157	17.0	3.5	18.9
synthetic	0.067	0.080	0.324	37.3	10.5	65.1
australian	0.091	0.313	0.309	27.7	7.4	20.8
energy	0.042	0.260	0.178	24.4	4.6	88.9
pima	0.041	0.367	0.276	19.3	3.8	30.8
vehicle	0.074	0.128	0.396	37.5	7.8	41.9
annealing	0.064	0.427	0.786	53.5	12.4	180
tic	0.046	0.486	0.925	51.4	18.4	132
mammograph	0.030	0.429	0.517	27.3	6.7	68.9
isolet	1.47	0.720	42.4	2793	1109	313
imseg	0.014	0.020	0.061	3.9	1.7	8.6
abalone	0.100	3.19	8.66	574	161	78.5
sat	0.442	3.69	15.9	2149	439	101
musk	2.06	18.0	123	8872	1244	279
usps	4.50	41.5	1336	36154	3143	1111
grid	0.290	36.6	46.5	1423	225	59.4
nursery	0.459	25.1	140	7257	2226	332
magic	0.470	137	161	13673	2503	218
letter	0.819	1.40	126	26796	4378	361
chess	1.28	28.4	753	70106	14253	1208
adult	10.4	1967	17215	–	85099	17155
shuttle	1.43	2921	18.6	–	469	101
mnist	105	1299	85583	–	–	163685
wisdm	11.4	30.1	4436	–	–	18067
ijcnn1	384	–	12493	–	–	16029
poker	0.775	670	733	–	12686	1017
Average	–	105	163	5549	1760	6420

The six methods were executed on a collection of 37 datasets (Table 3.1) selected from the UCI Machine Learning Repository [10] with up to 70,000 training patterns, 1,000,000 test patterns, 784 inputs and 26 classes. The experiments ran on a desktop computer with the following features: Intel Core i7-9700K processor (8 cores) at 3.6GHz with 64GB RAM

under operating system Kubuntu 20.04. The performance, measured by the Cohen kappa statistic [27]. The time elapsed by the selection of the kernel spread and the memory required by the tuning method were recorded for each method.

Table 3.2 reports the kappa achieved by the 6 methods. The missing values correspond to datasets where the method failed due to out-of-memory errors or did not finish within 6 days (518,400 seconds). The average values of the last line are calculated, for each method, over all the datasets excluding those with fails. The IKT, GS and Bayes never fail, while KDE fails on dataset `ijcnn1`, PSO fails in three datasets (`mnist`, `wisdm` and `ijcnn1`) and GA fails in the 6 largest datasets. The kappa values for each dataset are in general very similar for the different methods, with some exceptions: GS achieves 18% in dataset `adult`, where IKT, KDE, PSO and Bayes achieve 52-56%; and GA achieves 79.4% in `isolet`, while the other methods achieve 94%. The average values are also similar, being Bayes the best (80%), followed by GA (79.7%), IKT (78.8%), PSO (78.4%) and KDE (78.2%), while GS performs slightly worse (76.4%). Figure 3.2 plots the kappa values of IKT, KDE and GS, sorted by decreasing values. The GS is below the other methods in four datasets (`ijcnn1`, `mammograph`, `adult` and `abalone`), while KDE is twice above (datasets `tissue` and `pima`) and once below IKT (dataset `wisdm`).

Table 3.3 reports the time spent by each method to select σ, excluding the SVC training and test. The last row reports, for all the methods excepting IKT and for each dataset, the ratio of the time spent by that method on that dataset divided by the time spent by IKT on the same dataset, averaged over all the datasets excluding fails. This average says how many times the method is slower than IKT. The times spent by IKT are very small and raise very slowly with the dataset size. In some cases the times are higher compared to datasets with similar size: `isolet` (1.47 s. while the previous dataset `mammograph` spends 0.03 and the next dataset `imseg` spends 0.014), because the high I and Q slow down the prototype learning; `musk`, `usps` and `adult`, also due to their high I; `mnist` due to the large $I = 784$ and $Q = 10$. The KDE is also fast, but slower than IKT, in some cases with high difference: in `usps`, `grid` and `nursery`, KDE spends 41, 36 and 25 s., while IKT spends 4.5, 0.29 and 0.45 s.

Larger differences occur in `adult`, where IKT and KDE spend 10.4 and 1,967 s., and `shuttle` (1.43 and 2,921 s.), because the two most populated classes (1 and 4) have 40,856 of 43,483 training patterns, so KDE is much slower than IKT, that uses a reduced number of class prototypes. In fact, KDE fails in dataset `ijcnn1` by lack of memory. Given that GS requires to train and test the SVC many times, it is much slower than IKT and KDE, specially

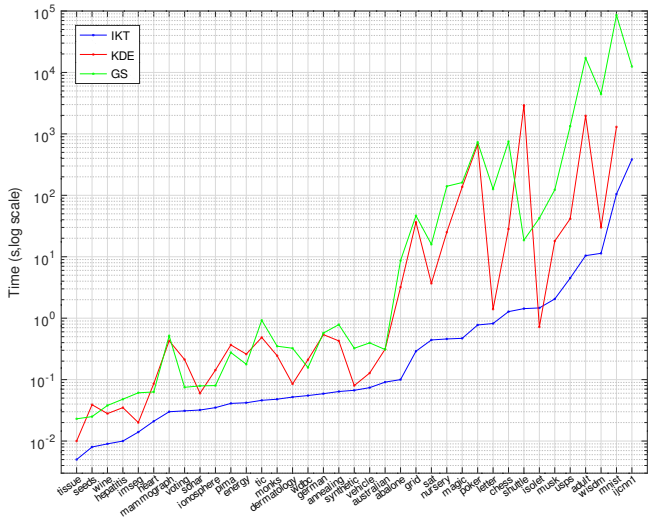

Figure 3.3: Time (in s.) spent by IKT, KDE and GS.

in datasets `adult` (17,215 s. with GS, 10.4 and 1,967 s. with IKT and KDE) and `mnist` (85,583 s. with GS, 105 and 1,299 s. with IKT and KDE, respectively). However, GS is much faster than KDE in dataset `shuttle` (18.6 and 2,921 s., respectively), because the SVC training in the former is relatively fast, while the latter is slow as we explained above. The GA is clearly the slowest method, in fact it does not finish after days of execution in the six largest datasets. The PSO is also slow, but faster than GA in all the datasets, failing only in the three largest datasets. Finally, the Bayes method is slower than GA and PSO in the small datasets, but faster in the largest ones, and it does not fail in any dataset, as opposite to GA and PSO. In average over all datasets, KDE and GS are 105 and 163 times (two orders of magnitude) slower than IKT, while GA, PSO and Bayes are 5,549, 1,760 and 6,420 times (three-four orders of magnitude) slower than IKT. The number corresponding to Bayes is somehow confusing, because it seems to be the slowest tuning method, but in the large datasets where GA and PSO do not fail they are slower than Bayes. Similarly to GS, the large times of GA, PSO and Bayes are expectable because they require to train and test the SVC.

Table 3.4: Memory (in MB) required by each method and dataset.

Dataset	IKT	KDE	GS	GA	PSO	Bayes
tissue	0.04	0.02	0.02	0.04	0.04	0.04
hepatitis	0.38	0.23	0.05	0.05	0.05	0.05
wine	0.26	0.16	0.05	0.05	0.05	0.05
sonar	0.81	0.45	0.24	0.12	0.12	0.12
seeds	0.28	0.18	0.03	0.04	0.04	0.05
heart	0.98	0.67	0.10	0.06	0.06	0.07
voting	1.0	1.7	0.09	0.06	0.05	0.06
ionosphere	1.1	1.1	0.21	0.11	0.10	0.11
dermatology	1.1	0.43	0.52	0.20	0.20	0.20
german	1.2	8.7	0.41	0.15	0.15	0.16
monks	1.0	1.6	0.10	0.06	0.06	0.06
wdbc	1.1	2.9	0.34	0.14	0.14	0.14
synthetic	1.0	0.42	0.71	0.25	0.25	0.26
australian	1.3	4.3	0.35	0.14	0.14	0.14
energy	1.0	3.5	0.14	0.07	0.07	0.07
pima	0.99	5.1	0.13	0.07	0.07	0.07
vehicle	1.1	1.7	0.32	0.13	0.12	0.13
annealing	1.2	5.5	0.53	0.20	0.20	0.20
tic	1.2	8.0	0.32	0.11	0.11	0.12
mammograph	0.98	8.0	0.09	0.06	0.06	0.07
isolet	11.8	0.55	18.6	5.7	5.7	5.7
imseg	0.16	0.06	0.11	0.07	0.06	0.07
abalone	1.2	70.9	0.70	0.24	0.24	0.24
sat	2.6	80.0	4.7	1.4	1.4	1.4
musk	7.9	380	19.9	6.4	6.3	6.4
usps	8.0	756	45.6	13.7	13.7	13.7
grid	1.9	859	2.7	0.81	0.81	0.81
nursery	3.2	634	4.2	1.1	1.1	1.1
magic	2.3	3106	3.9	1.2	1.2	1.2
letter	3.3	22.6	8.9	1.9	1.9	1.9
chess	8.1	659	13.7	3.3	3.3	3.3
adult	8.2	20505	94.3	–	29.5	29.5
shuttle	4.4	25474	3.7	–	3.4	3.4
mnist	9.7	2660	690	–	–	344
wisdm	3.6	641	80.7	–	–	39.1
ijcnn1	4.5	–	49.6	–	–	18.3
poker	3.5	9547	4.3	–	2.1	2.1
Average	—	383.9	3.9	0.313	0.4	1.8

Figure 3.3 plots the time spent by IKT, KDE and GS in all the datasets. The plots of KDE and GS are almost always above IKT, with the only exception of `isolet`, where the high Q slows down the prototype learning of IKT (1.47 s.), that is slower than KDE (0.72 s.). Generally, IKT is between one and two orders of magnitude faster than KDE and GS. To avoid confusion in the figure, the times of GA, PSO and Bayes are not plotted, but they are much higher in all the datasets.

Table 3.4 reports the RAM memory required by each method and dataset, excluding the memory used by the SVC training and prediction, that is much larger. Thanks to the limitation on the number of class prototypes, the memory required by IKT is below 10 MB even in the

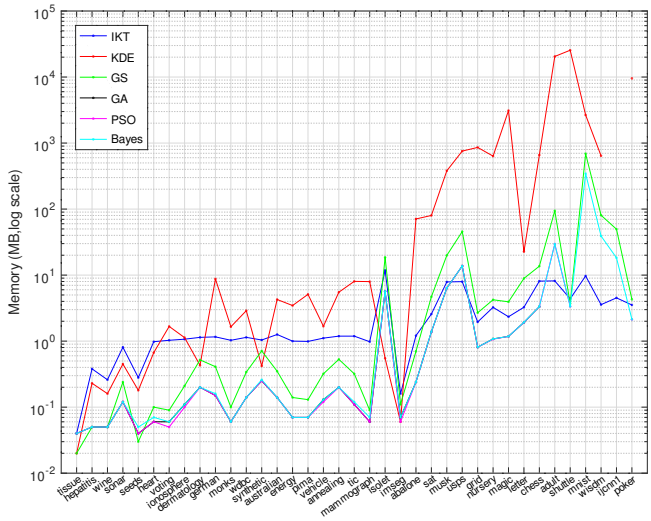

Figure 3.4: Memory consumption (in MB) of each algorithm.

largest datasets, much below the capability of a today's standard desktop computer. On the contrary, the memory requirements of KDE raise in dataset `abalone` (N=2,090, memory of 70.9 MB) reaching large values in datasets `adult` (20.5 GB, that is a huge amount of memory for a medium sized dataset), `shuttle` (25.4 GB), `mnist` (2.6 GB) and `poker` (9.5 GB), due to the high number of training patterns of the two most populated classes: 24,422 in `adult`, 40,856 in `shuttle`, 12,873 in `mnist` and 25,010 in `poker`. The GS uses also more memory than IKT, specially in datasets larger than `musk`. The GA and PSO require small amounts of memory. Bayes requires less memory than IKT in the small datasets, and more memory in the large datasets: `adult` (29.5 MB and 8.2 MB with Bayes and IKT, respectively), `mnist` (344 MB and 9.7 MB), `wisdm` (39.1 MB and 3.6) and `ijcnn1` (18.3 MB and 4.5 MB). The last line reports the ratio of the memory requirements of each method divided by IKT excluding datasets with fails: the KDE, GS and Bayes require 400, 4 and 2 times more memory, respectively, than IKT. The memory of GA and PSO is so low as IKT on the small datasets, being unknown on the largest datasets because they fail. Figure 3.4 plots the memory consumption of the six methods. The memory of IKT raises only very slowly with the dataset size even for the largest datasets. In the small datasets GA, PSO and Bayes

are below IKT, but in the large datasets GS and PSO do not run and Bayes overcomes IKT, while KDE is the most expensive and GS also overcomes IKT.

CHAPTER 4

CONCLUSION

The current thesis proposes methods that allow to execute the radial basis function (RBF) kernel-based support vector classifier (SVC) on large-scale classification problems. It is widely known in the literature that SVC is not able to process datasets beyond several thousands of training patterns, depending on the pattern dimensionality and on the number of classes. Although there are approaches in the literature that try to allow the execution of SVC over large datasets, they are usually focused to training set selection, often by means of complex selection methods that are slow for really large datasets. On the other hand, alternative SVC solvers are not efficient enough so as to be executed on large datasets.

The proposed approach, named fast support vector classifier (FSVC), is inspired on the SVC theory and designed to allow the classification of large datasets with many patterns, inputs and classes. It uses a closed-form expression that directly calculates the output of the binary SVC from the training patterns in an efficient way. The use of a closed-form expression for the calculation of the SVC trainable parameters is very fast, but requires the storage in memory of the whole training set, so it does not scale pretty well with the number of training patterns. In order to avoid storing the whole training set in memory, the FSVC uses a small number of prototypes representing each class, being scalable to highly populated datasets. As well, the size of the trainable parameters, that raises with the dataset size for small datasets, is upper bounded for large sizes, so that FSVC can still be executed when the dataset population raises to tens of millions of training patterns.

One of the most important issues in the SVC performance is the tuning of the RBF kernel spread. This process is very relevant because the performance is very sensitive to the spread

value, but it has a high computational cost and a high impact on the SVC speed. Most existing tuning methods require to train and test the SVC several times, that slows down its execution. The RBF kernel spread is selected in FSVC directly from data by requiring a RBF kernel matrix closest to the ideal kernel matrix. This is a very efficient approach and avoids to repeat the training, that is not factible for large datasets. The linear kernel is used instead RBF kernel with high-dimensional datasets, increasing even more the efficiency of FSVC because only simple dot and sum operations are required. For multi-class classification, the one-vs-one approach, that requires a number of binary SVCs that scales quadratically with the number of classes, is used by default due to its higher performance. For datasets with a high number of classes, the one-vs-all approach must be used for efficiency purposes, because it only requires so many binary SVCs as classes.

On the small datasets and in terms of performance, FSVC outperforms Liblinear, that is the state-of-the-art linear SVM classification for large-scale datasets, and is near to Libsvm, that uses RBF kernel. In terms of speed, FSVC is four times faster as Liblinear and two orders of magnitude faster than Libsvm. On the large datasets (up to 31 million patterns, 30,000 inputs and 131 classes), where Libsvm fails to be executed, FSVC outperforms Liblinear in performance, being about 11 times faster and requiring 12.5 times less memory, while Liblinear fails on the largest 4 datasets by lack of memory. The FSVC is able to process datasets of arbitrarily large sizes either in number of patterns, inputs and classes, on a standard computer with 32GB RAM, in our dataset collection in less than 1 hour 40 minutes. Unlike Libsvm and Liblinear, the time of FSVC does not depend on the difficulty of the classification problem, but only on its size, and it can be accurately estimated for a new dataset as $6{\cdot}10^{-7}$ multiplied by the number of total (training and test) patterns, inputs and classes.

The current thesis compares FSVC with several existing efficient SVM solvers in the literature, such as primal estimated sub-gradient solver for SVM (Pegasos-SVM), sublinear importance-sampling bi-stochastic algorithm (SVM-SIMBA), indefinite core vector machine (ICVM) and doubly stochastic gradient (DSG). The comparison has been also performed with evolutionary-based set selection methods, specifically the steady-state genetic algorithm for instance selection (SGA). Due to the storing of a reduced collection of prototypes, FSVC adapts to the available memory by reducing the data block size to be read from disk files, being able to execute on low power computing devices and classifying even the largest datasets (31 million patterns) with only 2GB of RAM in less than 3 hours.

The proposal of this thesis for RBF kernel spread tuning has also lead to a wider comparison to existing model selection methods, that is a very relevant subject in the SVM literature. This method, named ideal kernel tuning (IKT), is a simple and efficient approach that selects the kernel spread of the SVC directly from data, being scalable for large datasets because no training nor testing is required. First, the ideal kernel matrix, with value 1 only in the items corresponding to patterns of the same class and 0 otherwise, is created for the binary classification problem defined by the two most populated classes. Second, the spread that mimimizes the difference between the RBF and ideal kernel matrices is found. Third, both matrices are calculated for a limited number of prototypes of both classes instead of the original training patterns, in order to avoid unaceptably large kernel matrices. Since their size is bound, IKT scales well with the training set size, so in large datasets nor time nor memory consumption will increase until raising memory failures.

Overall, to select the optimal spread only requires to calculate the kernel matrix for a reduced collection of spread values, already defined in the literature. The size of the RBF and ideal kernel matrices is upper bounded, so it scales well with the dataset population, although with large datasets the SVC can not be trained due to its high computational cost. The mean of the absolute difference between the ideal and the RBF kernel matrices is calculated for each spread value in the collection. The selected spread is the one that achieves the lowest difference. When this minimum is located on the extremes of the spread range, the spread value that maximizes the derivative of this difference is selected instead. This spread value can be used either in binary or multi-class classification using the two most populated classes, where it is common practice to be shared among binary SVCs.

Our experiments compared IKT to five alternative RBF kernel spread tuning methods: kernel density estimation (KDE), grid-search (GS), genetic algorithms (GA), particle swarm optimization (PSO) and Bayesian optimization on a collection of 37 datasets up to 70,000 training patterns, 1,000,000 test patterns, 784 inputs and 26 classes. The performance of IKT is higher than GS, that is the standard tuning method, KDE and PSO, being only slightly below GA and Bayes. However, GA fails in the largest datasets, while both GA and Bayes spend many days to finish in the remaining ones. On the contrary, IKT is the fastest method: 105 and 163 times faster than KDE and GS, and 5,549, 1,760 and 6,420 times faster than GA, PSO and Bayes, respectively. Besides, IKT requires little memory for the capability of today's computers (383 times less than KDE), and this memory raises very little with the dataset size, as opposed to KDE, that is the only alternative tuning method that performs tuning without

SVC training. The conclusion is that IKT provides an interesting alternative for the RBF kernel spread tuning for SVC with medium and large sized datasets.

The future work pursuits to extend FSVC to regression problems; to design approaches more efficient than OVO and OVA for classification problems with many classes; to increase the performance of the proposed efficiency training; and to use the information about the dataset provided by the kernel matrix in order to refine the SVC performance.

Bibliography

[1] H. Bhavsar and M.H. Panchal. A review on support vector machine for data classification. *Int J Adv Res in Comput Engin & Technol*, 1:185–189, 2012.

[2] J.R. Cano, F. Herrera, and M. Lozano. Using evolutionary algorithms as instance selection for data reduction in KDD: an experimental study. *IEEE T Evol Comput*, 7:561–575, 2003.

[3] C.C. Chang and C.J. Lin. LIBSVM: A library for support vector machines. *ACM Trans Intel Systs Technol*, 2:27:1–27:27, 2011.

[4] G. Chen, Y. Cheng, and J. Xu. Cluster reduction support vector machine for large-scale data set classification. In *IEEE Pacific-Asia Workshop on Comput Intel and Ind Appl*, pages 8–12, 2008.

[5] C. Cortes and V. Vapnik. Support-vector networks. *Mach Learn*, 20(3):273–297, 1995.

[6] T.M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Trans Electron*, 14(3):326–334, 1965.

[7] K. Cramer and Y. Singer. On the algorithmic implementation of multiclass kernel-based support vector machines. *J Mach Learn Res*, 2:265–292, 2001.

[8] B. Dai, B. Xie, N. He, Y. Liang, A. Raj, M.F. Balcan, and Le Song. Scalable kernel methods via doubly stochastic gradients, 2015.

[9] T. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *J Artif Intel Res*, 2:263–286, 1995.

[10] D. Dua and C. Graff. UCI machine learning repository, 2017. http://archive.ics.uci.edu/ml.

[11] R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. LIBLINEAR: A library for large linear classification. *J Mach Learn Res*, 9:1871–1874, 2008.

[12] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *J Mach Learn Res*, 15:3133–3181, 2014.

[13] M. Fernández-Delgado, E. Cernadas, S. Barro, J. Ribeiro, and J. Neves. Direct kernel perceptron (DKP): Ultra-fast kernel ELM-based classification with non-iterative closed-form weight calculation. *Neural Netw*, 50:60–71, 2014.

[14] F. Friedrichs and C. Igel. Evolutionary tuning of multiple SVM parameters. *Neurocomputing*, 64:107–117, 2005.

[15] Z.A. Ali Hammouri, M. Fernández-Delgado, A. Albtoush, E. Cernadas, and S. Barro. Ideal kernel tuning: fast and scalable selection of the radial basis kernel spread for support vector classification. *Neurocomputing*, In second revision:1–10, 2022.

[16] Z.A. Ali Hammouri, M. Fernández-Delgado, E. Cernadas, and S. Barro. Fast SVC for large-scale classification problems. *IEEE T Pattern Anal*, pages 1–12, 2021.

[17] S. Har-Peled, D. Roth, and D. Zimak. Maximum margin coresets for active and noise tolerant learning. In *Intl J Conf Artif Intel*, pages 836–841, 2007.

[18] E. Hazan, T. Koren, and N. Srebro. Beating SGD: learning SVMs in sublinear time. In *Advanc Neur Inf Proc Systs*, pages 1233–1241, 2011.

[19] C. Hsieh, K. Chang, C. Lin, S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *Proc Intl Conf Mach Learn*, page 408–415, 2008.

[20] C.W. Hsu, C.C. Chang, and C.J. Lin. A practical guide to support vector classification, 2003.

[21] P.S. Huand, H. Avron, T.N. Sanath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on timit. In *Intl Conf Acoustic, Speech Signal Proc*, pages 205–209, 2014.

[22] M.N. Kapp, R. Sabourin, and P. Maupin. A dynamic model selection strategy for support vector machine classifiers. *Appl Soft Comput*, 12:2550–2565, 2012.

[23] B. Li, Q. Wang, and J. Hu. A fast SVM training method for very large datasets. In *Intl J Conf Neural Netw*, pages 1784–1789, 2009.

[24] X. Lian, Y. Ma, Y. He, L. Yu, R.C. Chen, T. Liu, X. Yang, and T.S. Chen. Fast pruning of superfluous support vectors in SVMs. *Patt Recog Lett*, 34:1203–1209, 2013.

[25] K.M. Lin and C.J. Lin. A study on reduced support vector machines. *IEEE T Neural Netw*, 14:1449–1459, 2003.

[26] Z. Liu and H. Xu. Kernel parameter selection for support vector machine classification. *J Alg Comput Technol*, 8(2):163–177, 2014.

[27] M.L. McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, 22:276–282, 2012.

[28] M. Menezes, L. Torres, and A. Braga. Width optimization of RBF kernels for binary classification of support vector machines: a density estimation-based approach. *Patt Recogn Lett*, 128:1–7, 2019.

[29] P. Miranda and R. Prudencio. Active testing for SVM parameter selection. In *Intl J Conf Neural Netw*, pages 1–8, 2013.

[30] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[31] J. Nalepa and M. Kawulok. Selecting training sets for support vector machines: a review. *Artif Intel Rev*, 52:857–900, 2019.

[32] X. Pan and Y. Xu. A novel and safe two-stage screening method for support vector machine. *IEEE T Neur Net Lear*, 30:2263–2274, 2019.

[33] M. Pérez-Ortiz, P.A. Gutiérrez, J. Sánchez-Monedero, and C. Hervás-Martínez. A study on multi-scale kernel optimisation via centered kernel-target alignment. *Neural Proc Let.*, 44:481–517, 2016.

[34] J. Platt, N. Cristianini, and J. Shawe-Taylor. Large margin DAG for multiclass classification. In *Adv Neur Inf Proc Syst*, page 547–553, 2000.

[35] A. Rosales-Pérez, S. García, J.A. González, C.A. Coello, and F. Herrera. An evolutionary multiobjective model and instance selection for support vector machines with Pareto-based ensembles. *IEEE T Evol Comput*, 21(6):863–877, 2017.

[36] A. Rosales-Pérez, J.A. González, C.A. Coello, H.J. Escalante, and C.A. Reyes-García. Surrogate-assisted multi-objective model selection for support vector machines. *Neurocomputing*, 150:163–172, 2015.

[37] V. Saradhi and K. Girish. Effective parameter tuning of SVMs using radius/margin bound through data envelopment analysis. *Neural Proc Lett*, 41:125–138, 2015.

[38] F.M. Schleif and P. Tino. Indefinite core vector machine. *Patt Recogn*, 71:187–195, 2017.

[39] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: primal estimated sub-gradient solver for SVM. *Math Program Ser B*, 127:3—30, 2011.

[40] J. Shawe-Taylor and S. Sun. A review of optimization methodologies in support vector machines. *Neurocomputing*, 74:3609–3618, 2011.

[41] H. Shi, H. Xiao, J. Zhou, N. Li, and H. Zhou. Radial basis function kernel parameter optimization algorithm in support vector machine based on segmented dichotomy. In *Intl Conf Syst Inform*, pages 383–388, 2018.

[42] A. Tharwat and T. Gabel. Parameters optimization of support vector machines for imbalanced data using social ski driver algorithm. *Neural Comput Appl*, 32:6925–6938, 2020.

[43] A. Tharwat, A.E. Hassanien, and B.E. Elnaghi. A BA-based algorithm for parameter optimization of support vector machine. *Patt Recogn Lett*, 93:13–22, 2017.

[44] I. Tsang, J. Kwok, and P. Cheung. Core vector machines: fast SVM training on very large datasets. *J Mach Learn Res*, 6:363–392, 2005.

[45] M. Tukan, C. Baykal, D. Feldman, and D. Rus. On coresets for support vector machines. In *Intl Conf Theory Appl Models Comput*, pages 287–299, 2020.

[46] V. Vapnik. *Statistical learning theory*. Wiley-Interscience, 1998.

[47] N. Verbiest, J. Derrac, C. Cornelis, S. García, and F. Herrera. Evolutionary wrapper approaches for training set selection as preprocessing mechanism for support vector machines: Experimental evaluation and support vector analysis. *Appl Soft Comput*, 38:10–22, 2016.

[48] W. Wang, Z. Xu, W. Lu, and X. Zhang. Determination of the spread parameter in the Gaussian kernel for classification and regression. *Neurocomputing*, 55:643–663, 2003.

[49] Z. Wang, Y. Shao, and T. Wu. Proximal parametric-margin support vector classifier and its applications. *Neural Comput Appl*, 24:755–764, 2014.

[50] C. Zhang, F. Nie, and S. Xiang. A general kernelization framework for learning algorithms based on kernel PCA. *Neurocomputing*, 73(4):959–967, 2010.

List of Figures

List of Tables

This thesis proposes the fast support vector classifier, an efficient implementation of the radial basis function support vector machine (SVM) for large-scale classification problems. It achieves performance near the state-of-the-art, being much faster than existing approaches over datasets up to 31 million patterns, 30,000 inputs and 131 classes. It also adjusts the memory requirements, allowing to be executed on datasets of almost arbitrary size. The thesis also proposes the ideal kernel tuning, an efficient tuning method for the Gaussian kernel spread of the SVM, that is the fastest compared to other five methods in the literature, with performance very near to the best and reduced memory requirements.