



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

Inversão de Matrizes em Bloco Aplicada à Validação Cruzada da Máquina de Vetor de Suporte por Mínimos Quadrados

Ormazabal Lima do Nascimento

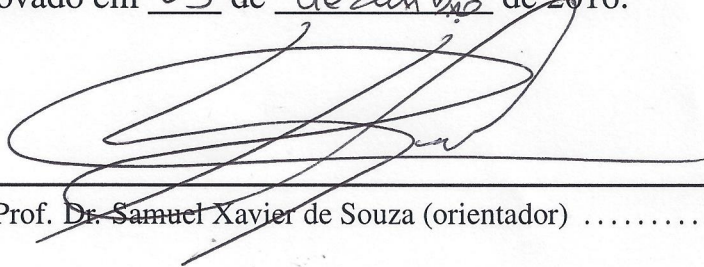
Trabalho de Conclusão de Curso apresentada ao curso de Engenharia da Computação da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do título de Engenheiro da Computação, orientado pelo Prof. Dr. Samuel Xavier de Souza e pela Prof. Me. Iria Caline Saraiva Cosme

Natal - RN
2016

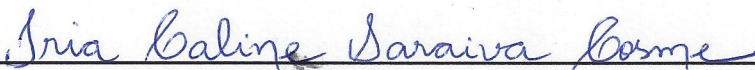
Inversão de Matrizes em Bloco Aplicada à Validação Cruzada da Máquina de Vetor de Suporte por Mínimos Quadrados

Ormazabal Lima do Nascimento

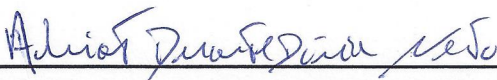
Aprovado em 09 de dezembro de 2016.



Prof. Dr. Samuel Xavier de Souza (orientador) DCA/UFRN



Prof. Me. Iria Caline Saraiva Cosme (co-orientador) IFRN



Prof. Dr. Adrião Duarte Dória Neto DCA/UFRN

Agradecimentos

À minha família, em especial minha mãe Lenita, por todo o apoio e incentivo em todos os momentos da minha vida.

À minha namorada Mariana, por estar sempre comigo me motivando a fazer o meu melhor.

Aos professores Samuel Xavier de Souza e Iria Caline Saraiva Cosme, meu orientador e co-orientadora respectivamente, pela oportunidade e por todo o conhecimento compartilhado.

Aos meus colegas de curso, em especial Mateus Antonio, Yuri Aguiar, Felipe Fernandes, Tiago Fernandes e José Victor, pela troca de ideias, pelas risadas e pelas críticas construtivas.

A todos que de alguma forma fizeram parte da minha formação.

Resumo

Máquina de Vetor de Suporte por Mínimos Quadrados (do inglês, *Least Squares Support Vector Machines* ou LS-SVM) é um classificador de margem ótima que necessita de um conjunto de parâmetros de entrada que, se corretamente selecionados, farão a máquina alcançar bons resultados em suas predições. A Validação Cruzada (do inglês, *Cross-Validation* ou CV) é uma técnica que auxilia nessa seleção, calculando a taxa de erro de cada conjunto de parâmetros testado. Apesar de sua importância, sua aplicação requer um alto custo computacional, inviabilizando sua execução em *hardwares* mais modestos.

A CV requer a inversão de grandes matrizes de dados, denominadas matrizes de *kernel*. Como solução desse problema, este trabalho propõe a técnica da Inversão de Matrizes em Blocos. Essa técnica auxiliará na otimização da CV, alocando menos memória para esse cálculo. Serão apresentados todos os passos necessários para adaptação do algoritmo original, assim como testes em consumo de tempo e memória com o objetivo de validar a proposta.

Palavras-chave: Aprendizado de máquina, LS-SVM, Validação Cruzada, Inversão de matrizes em bloco.

Abstract

Least Square Support Vector Machine (LS-SVM) is an optimal margin classifier that requires a set of input parameters that, if correctly selected, will cause the machine to achieve good results in its predictions. Cross-Validation (CV) is a technique that assists in this selection, calculating the performance of each set of parameters tested. In spite of its importance, its application requires a high computational cost, making it impossible to execute in more modest hardware.

The CV requires the inversion of large data matrices, called kernel matrix. As a solution to this problem, this work will propose the technique of Inversion of Matrices in Blocks. This technique will aid in the optimization of CV, allocating less memory for this calculation, and obtaining similar results. All necessary steps will be presented to adapt the original algorithm, as well as tests in time and memory consumption with the purpose of validating the proposal.

Keywords: Machine learning, LS-SVM, Cross-Validation, Inversion of matrices in block.

Sumário

Sumário	i
Lista de Figuras	iii
Lista de Tabelas	iv
Lista de Símbolos e Abreviaturas	v
1 Introdução	1
1.1 Motivação	1
1.2 Objetivos	2
1.3 Estado da Arte	2
1.4 Organização do Trabalho	3
2 Fundamentação Teórica	4
2.1 Aprendizagem de Máquina	4
2.2 Máquina de Vetor de Suporte	4
SVMs Lineares com Margens Rígidas	4
SVMs Lineares com Margens Suaves	6
SVMs Não Lineares e Funções Kernel	7
2.3 Máquina de Vetor de Suporte por Mínimos Quadrados	9
2.4 Validação Cruzada	10
2.5 Inversão de Matrizes em Blocos	11
Procedimento Recursivo Para Frente	12
Procedimento Recursivo Para Trás	14
3 Metodologia	15
3.1 Ferramentas	15
LS-SVMLab	15
Armadillo	17
Mex	17
3.2 Implementação	18
4 Resultados e Discussão	20
5 Conclusão	25

Lista de Figuras

2.1	Distância entre os hiperplanos H_1 e H_2	5
2.2	Interpretação geométrica das variáveis de folga ξ	7
2.3	Mapeamento de um conjunto de dados não-linearmente separável	8
2.4	Interpretação geométrica das variáveis de erro e	10
2.5	Interpretação gráfica do Procedimento Recursivo Para Frente	13
2.6	Interpretação gráfica do Procedimento Recursivo Para Trás	14
3.1	Organização do fluxo de informações no LS-SVMlab	16
4.1	Comparação em tempo e memória total gastos com 3 folds entre o CV padrão e o BRI	22
4.2	Comparação em tempo e memória total gastos com 4 folds entre o CV padrão e o BRI	22
4.3	Comparação em tempo e memória total gastos com 5 folds entre o CV padrão e o BRI	23
4.4	Comparação em tempo e memória total gastos com 6 folds entre o CV padrão e o BRI	23

Lista de Tabelas

2.1	Funções <i>kernel</i>	9
4.1	Comparativo de Tempo	20
4.2	Comparativo de Memória	21

Lista de Símbolos e Abreviaturas

API:	<i>Application Programming Interface</i>
BRI:	<i>Block Recursive Inverse</i>
BRP:	<i>Backward Recursive Procedure</i>
CSA:	<i>Coupled Simulated Annealing</i>
CV:	<i>Cross-Validation</i>
FRP:	<i>Forward Recursive Procedure</i>
LS-SVM:	<i>Least Squares Support Vector Machine</i>
SV:	<i>Support Vector</i>
SVM:	<i>Support Vector Machine</i>

Capítulo 1

Introdução

1.1 Motivação

Nunca se produziu tanta informação quanto nos últimos anos. A quantidade de dados gerados tem crescido numa escala exponencial. Originados de dispositivos móveis, GPS, redes sociais, entre outras fontes, estima-se que a cada dia são produzidos 2,5 quintilhões de bytes de dados (DEVAKUNCHARI, 2014). Esse fenômeno é denominado *big data*. Toda essa informação traz consigo diversos desafios, como armazenamento e organização desses dados, mas também oportunidades gigantescas. Extrair conhecimento dessas enormes bases de dados é essencial para o avanço da ciência e da indústria, e técnicas de aprendizado de máquina são excelentes ferramentas para esse fim.

Máquina de Vetor de Suporte (do inglês, *Support Vector Machine* ou SVM) é uma máquina de aprendizado estatístico que tem em sua característica principal a obtenção do hiperplano ótimo de separação entre os dados analisados (VAPNIK, 1995). Caso o problema não seja linearmente separável, são utilizadas funções *kernel* que mapeiam esses dados do espaço de entrada para um espaço de alta dimensionalidade, denominado espaço de características, no qual existe alta probabilidade desses dados serem linearmente separáveis.

Devido à dificuldade de sua implementação, como também o uso de programação quadrática (do inglês, *Quadratic Programming* ou QP) em sua formulação, foi desenvolvida a Máquina de Vetor de Suporte por Mínimos Quadrados (do inglês, *Least Squares Support Vector Machine* ou LS-SVM). Essa modificação da SVM proporciona um menor custo computacional sem perda na qualidade das soluções. Ao invés da QP, nela se deve solucionar um conjunto de equações lineares (SUYKENS; VANDEWALLE, 1999).

A generalização do modelo é uma etapa essencial no aprendizado de máquina. Quanto mais preciso ela for, melhores serão as previsões da máquina. Para avaliar sua capacidade de generalização utiliza-se a Validação Cruzada (do inglês, *Cross-Validation* ou CV).

Na LS-SVM, a CV está diretamente relacionada à seleção de parâmetros da máquina, e sua execução depende de um alto consumo de memória. Dado um conjunto de entrada de n pontos, a CV requer $O(n^2)$ de memória (EDWARDS *et al.*, 2013). Essa memória é alocada no armazenamento da matriz de *kernel*, gerada da função *kernel* aplicada aos dados de entrada. Apesar de alocar espaço para toda a matriz, apenas as submatrizes pertencentes a sua diagonal principal são usadas a cada iteração do algoritmo.

O enfoque desse trabalho está na melhoria da CV, propondo que é preciso armazenar apenas a submatriz necessária a iteração em questão. Para tanto, a técnica da inversão de matrizes, proposta em COSME *et al.* (2016), será aplicada como solução para o problema. Esse algoritmo trabalha de forma recursiva, particionando a matriz em *frames* cada vez menores, até que todos eles resultem em *frames* de dimensão 2×2 . Nesse momento é calculado o complemento de Schur de cada *frame*, e eles vão sendo fundidos até que se tenha calculado a inversa dessa matriz (COSME *et al.*, 2016).

Assim, será proposto que os dados de entrada sejam aplicados a esse algoritmo para se calcular a inversa da matriz de *kernel*, em tempo de execução, sem que seja necessário o armazenamento desta matriz inteira na memória. Do contrário, apenas os blocos referentes a cada iteração é que estarão na memória.

1.2 Objetivos

O objetivo deste trabalho é aprimorar o uso da memória no algoritmo de CV da LS-SVM através da técnica da inversão de matrizes em blocos, proposta em COSME *et al.* (2016). A economia no consumo de memória resultará na expansão do volume de dados que poderão ser analisados, sem alterações de *hardware* nas máquinas. Serão realizados testes comparativos em consumo de tempo e memória para validar a proposta desse trabalho.

1.3 Estado da Arte

A CV é uma técnica de extrema importância no aprendizado de máquina. Vários são os trabalhos que sugerem algum tipo de melhoria nesse algoritmo.

Na implementação força bruta do algoritmo *l-fold*, os dados são divididos em l conjuntos, cada um contendo $(l - 1)$ blocos para treinamento e 1 bloco para ser usado na obtenção dos erros de predição. Foi mostrado em AN *et al.* (2007) que, apesar de apenas os blocos usados nos cálculos dos erros serem importantes, é necessário o armazenamento de todos os blocos, inclusive os que não serão usados, resultando em um custo de $O(n^2)$ de memória. A proposta deste trabalho está na modificação desse algoritmo, visando a economia de memória. Para isso, apenas os blocos necessários ao cálculo serão armazenados, resultando numa complexidade espacial de $O(\frac{n^2}{l^2})$. Desse modo, o consumo de memória varia não somente com o volume de dados de entrada n , mas também com o número de *folds* l .

Em EDWARDS *et al.* (2013) a matriz de *kernel* é obtida através de uma aproximação usando matrizes multi-nível de bloco circulante. Matrizes multi-nível são formadas a partir de partições da matriz original $m \times n$ em p regiões $m_i \times n_i$ denominadas níveis. Em uma matriz de bloco circulante, cada linha l é formada a partir de l permutações dos elementos da primeira linha. Isso significa que uma matriz multi-nível de bloco circulante pode ser completamente representada através da primeira linha e do índice multi-nível. Desse modo, o consumo de memória reduz de $O(n^2)$ para $O(n)$.

A proposta de EDWARDS *et al.* (2013) garante uma maior economia de memória, resultando num custo $O(n)$. A abordagem utilizada por ele se justifica, pois a matriz de *kernel* é, aproximadamente, uma matriz de bloco circulante. Ainda assim, muitos valores obtidos dessa maneira seriam diferentes dos valores exatos que deveriam ser calculados na CV. Por isso, neste trabalho, optou-se por não fazer esse tipo de aproximação.

Além desses, outros trabalhos também propõem melhorias na CV. Em DUAN *et al.* (2003), a proposta se concentra no algoritmo *l-fold* da SVM. Nele são testadas diversas medidas de desempenho que não requerem operações custosas, ou seja, não envolvem a utilização da matriz de *kernel*. Seu resultado aponta quais dessas medidas são adequadas à seleção de parâmetros na SVM. Em CAWLEY e TALBOT (2004) é apresentada uma implementação para *leave-one-out* que, ao invés de $O(l^2n^2)$ operações, computa com $O(ln^2)$ operações.

1.4 Organização do Trabalho

Este trabalho está estruturado da seguinte forma:

No Capítulo 2, será apresentada a teoria necessária para esse trabalho. Serão abordados tópicos envolvendo aprendizado de máquina, SVM e LS-SVM; Assim como será dado ênfase as técnicas de CV e inversão de matrizes em blocos.

No Capítulo 3, serão abordadas as ferramentas utilizadas, as modificações e integrações feitas para realização dos testes.

No Capítulo 4, os resultados obtidos serão explicitados, assim como uma discussão deles.

No Capítulo 5, serão apresentadas as considerações finais deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste capítulo será apresentada a teoria necessária para o entendimento deste trabalho. Serão detalhados os fundamentos da SVM e da LS-SVM, assim como os algoritmos de CV e a técnica de inversão de matrizes proposta em COSME *et al.* (2016).

2.1 Aprendizagem de Máquina

A aprendizagem de máquina é o estudo de algoritmos capazes de melhorar seu desempenho em uma determinada tarefa aprendendo através da sua experiência anterior (MJOLNESS; DECOSTE, 2001). Ela está dividida em dois tipos de aprendizagem: supervisionado e não-supervisionado.

No aprendizado supervisionado, existe a figura do professor, o qual fornece à máquina as entradas e as saídas associadas à elas. A máquina extrai seu conhecimento, a partir desses dados, com o objetivo de gerar as saídas corretas para novas entradas, até então desconhecidas por ela. No aprendizado não-supervisionado não há um professor. São apresentados dados não-rotulados à máquina e ela tenta identificar padrões nesses dados.

Caso as saídas sejam discretas, têm-se um problema de classificação. Se elas forem contínuas, têm-se um problema de regressão.

2.2 Máquina de Vetor de Suporte

A SVM é uma máquina de aprendizado estatístico originalmente proposta para problemas de classificação binária. Ela é denominada “classificador de margem ótima”, pois para problemas linearmente separáveis ela constrói um hiperplano que maximiza a margem de separação entre as classes, também chamado hiperplano ótimo. Seu processo de aprendizagem é do tipo supervisionado.

SVMs Lineares com Margens Rígidas

Dado um conjunto de treinamento linearmente separável $\{\mathbf{x}_i, y_i\}_{i=1}^N$ com saída binária $y = \{-1, +1\}$, a equação de um hiperplano de separação para esse problema é dado por:

$$\text{sign}[\mathbf{w}^T \mathbf{x} + b] = 0 \quad (2.1)$$

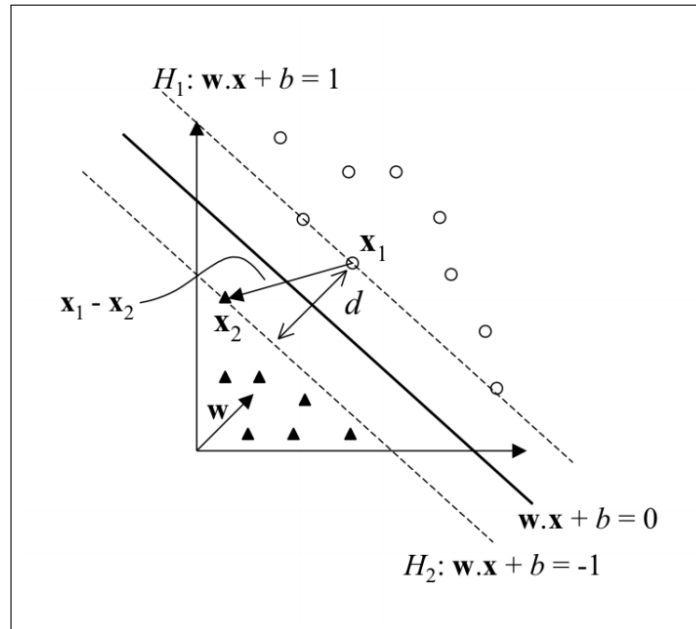
sendo \mathbf{w} um vetor de pesos ajustáveis, \mathbf{x} o vetor de entrada dos dados e b o bias. A classificação de cada dado de treinamento é feita de acordo com sua proximidade em relação às margens da sua classe, assim:

$$\begin{aligned} \text{sign}[\mathbf{w}^T \mathbf{x} + b] &\geq +1, \quad \text{para } y_i = +1 \\ \text{sign}[\mathbf{w}^T \mathbf{x} + b] &\leq -1, \quad \text{para } y_i = -1 \end{aligned}$$

que pode ser expressa de forma mais compacta para cada dado i como:

$$y_i(\mathbf{w}^T \mathbf{x} + b) \geq 1 \quad (2.3)$$

Figura 2.1 – Distância entre os hiperplanos H_1 e H_2



Fonte: (HEARST *et al.*, 1998)

Na Figura 2.1, observa-se que a distância d entre os hiperplanos tem comprimento $\frac{2}{|\mathbf{w}|}$. Disso verifica-se que a maximização da margem de separação pode ser obtida pela minimização de $|\mathbf{w}|$, portanto temos o seguinte problema de otimização:

$$\phi(\mathbf{w}) = \min \frac{1}{2} |\mathbf{w}|^2 \quad (2.4)$$

sujeito à restrição $\{ y_i(\mathbf{w}^T \mathbf{x} + b) - 1 \geq 0, \forall i = 1, \dots, n$

As restrições são impostas de maneira a assegurar que não haja dados de treinamento entre as margens de separação das classes. Por esse motivo, a máquina apresentada é denominada SVM com margens rígidas (LORENA; CARVALHO, 2007). Quando há uma

função de custo convexa, quadrática em \mathbf{w} , e com restrições lineares em \mathbf{w} , dá-se o nome de Programação Quadrática (QP) (LUENBERGER, 1973). Esse tipo de problema possui um único mínimo global, isso significa que existe um conjunto de valores de \mathbf{w} que torna $\phi(\mathbf{w})$ a mínima possível. Na literatura ele é referenciado como problema primal.

Partindo desse problema, pode-se construir um outro, chamado problema dual, que tem o mesmo valor ótimo do problema primal, mas com multiplicadores de Lagrange fornecendo a solução ótima. É interessante notar que apenas os dados de treinamento e seus rótulos são utilizados na forma dual.

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \quad (2.5)$$

$$\text{sujeito às restrições} \begin{cases} \alpha_i \geq 0, \forall i = 1, \dots, n \\ \sum_{i=1}^n \alpha_i y_i = 0 \end{cases}$$

Depois de determinados os multiplicadores de Lagrange ótimos α^* , pode-se calcular o vetor peso e o bias ótimos usando as seguintes equações:

$$\mathbf{w} = \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i \quad (2.6)$$

$$b = 1 - \mathbf{w}^T \mathbf{x} \quad (2.7)$$

Somente as amostras de treinamento que se encontram sobre os hiperplanos H_1 e H_2 possuem $\alpha > 0$. Elas são denominadas vetores de suporte (do inglês, *Support Vector* ou SV). São consideradas os dados mais informativos do conjunto de treinamento, pois somente elas influenciam na determinação do hiperplano separador, como visto na Equação 2.8. Todas as outras amostras possuem $\alpha = 0$.

$$y(\mathbf{x}) = \text{sign} \left(\sum_{\mathbf{x}_i \in SV} y_i \alpha_i^* \mathbf{x}_i \cdot \mathbf{x} + b \right) \quad (2.8)$$

SVMs Lineares com Margens Suaves

Ruídos e *outliers*¹ podem estar presentes nesses dados podendo provocar classificações incorretas. SVM's lineares podem ser estendidas para esses casos adicionando-se uma variável de folga ξ que relaxa as restrições já vistas, podendo aceitar como corretos dados situados ligeiramente fora da região de sua classe, impedindo que eles desviem o hiperplano de separação. Esse tipo de máquina é denominada SVM com margens suaves. As formulações que atendem esse tipo de SVM são vistas à seguir:

Problema primal:

$$\phi(\mathbf{w}, \xi) = \min \frac{1}{2} |\mathbf{w}|^2 + C \left(\sum_{i=1}^n \xi_i \right) \quad (2.9)$$

¹Dados inconsistentes, que apresentam um grande afastamento dos demais e por isso não representam a distribuição da classe a que pertencem.

$$\text{sujeito às restrições } \begin{cases} \xi_i \geq 0, \forall i = 1, \dots, n \\ y_i(\mathbf{w}^T \mathbf{x} + b) \geq 1 - \xi_i \end{cases}$$

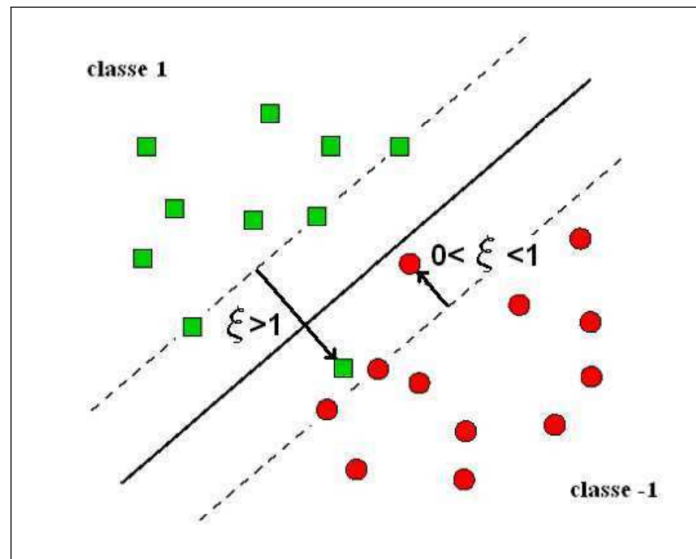
Problema dual:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \quad (2.10)$$

$$\text{sujeito às restrições } \begin{cases} 0 \leq \alpha_i \leq C, \forall i = 1, \dots, n \\ \sum_{i=1}^n \alpha_i y_i = 0 \end{cases}$$

Na formulação, C é um coeficiente de regularização, escolhido pelo usuário, fator que penaliza as variáveis de folga e tenta estabelecer um equilíbrio entre a complexidade do modelo e o erro de treinamento, evitando que todos os exemplos incorretos sejam desconsiderados. Como se observa na Figura 2.2, um erro no treinamento é representado por $\xi > 1$, e quando temos $0 \leq \xi \leq 1$ esses dados encontram-se do lado correto do hiperplano, mas dentro da região de separabilidade.

Figura 2.2 – Interpretação geométrica das variáveis de folga ξ



Fonte: (CARVALHO, 2005)

Na SVM de margens suaves, os SV recebem algumas classificações extras. Caso esteja situado sobre a margem mais próxima da região de sua classe, ele é denominado *SV nonbound* e possui $0 < \alpha < C$. Caso ele se localize entre a margem e a superfície de separação ($0 < \xi < 1$), na própria superfície de separação ($\xi = 1$), ou do outro lado da superfície de separação ($\xi > 1$), é denominado *SV bound* e possui $\alpha = C$.

SVMs Não Lineares e Funções Kernel

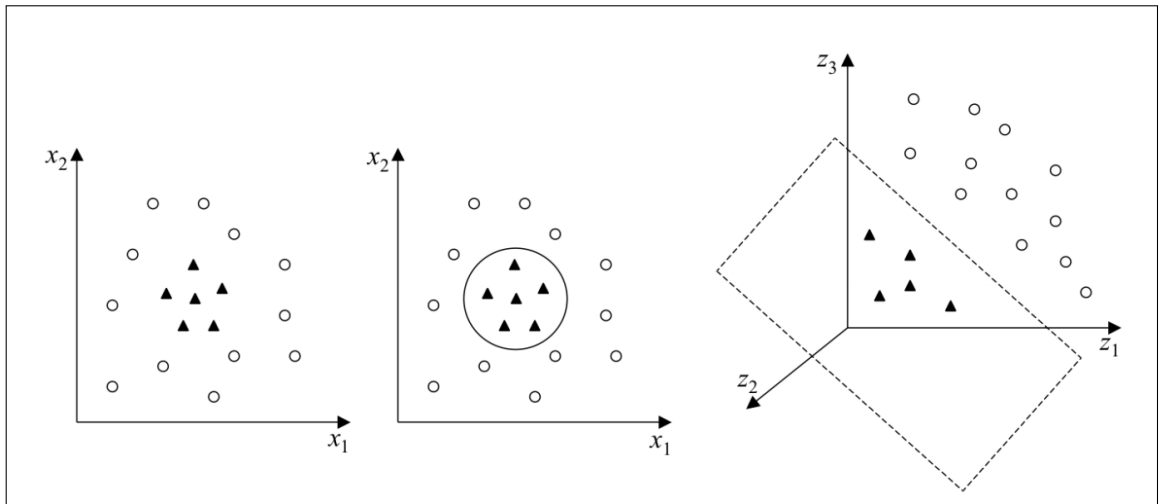
Até o momento, foi visto o uso de SVM na classificação de problemas lineares. Entretanto, problemas não-lineares são bem comuns e, para resolvê-los, as SVMs mapeiam

seu conjunto de treinamento do espaço de entrada para um espaço de alta dimensionalidade, denominado espaço de características. O uso desse procedimento é motivado pelo teorema de Cover (HAYKIN, 1999).

Dado um conjunto de dados não linear no espaço de entradas X , esse teorema afirma que X pode ser transformado em um espaço de características ζ no qual com alta probabilidade os dados são linearmente separáveis. Para isso, duas condições devem ser satisfeitas. A primeira é que a transformação seja não linear, enquanto a segunda é que a dimensão do espaço de características seja suficientemente alta (LORENA; CARVALHO, 2007).

Assim, para um problema não linear, mapeia-se os dados do espaço de entrada para o espaço de características, aplica-se a SVM linear que encontrará o hiperplano de separação, classificando corretamente esses dados. Depois, eles são mapeados de volta para o espaço de entrada, em que nesse espaço a região de separabilidade obtida já não será mais linear. Esse passo-a-passo está exemplificado na Figura 2.3.

Figura 2.3 – Mapeamento de um conjunto de dados não-linearmente separável



Fonte: (LORENA; CARVALHO, 2007)

Para fazer esse mapeamento são utilizadas funções *kernel*. Um *kernel* é uma função que recebe dois pontos do espaço de entradas e calcula o produto escalar desses dados no espaço de características.

$$K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) \quad (2.11)$$

Para garantir a convexidade do problema de otimização e que o *kernel* apresente um mapeamento no qual seja possível o cálculo de produtos escalares, o *kernel* a ser utilizado precisa satisfazer as condições estabelecidas pelo teorema de Mercer (PADILHA, 2013). Um *kernel* que satisfaz as condições de Mercer é caracterizado por dar origem a matrizes positivas semi-definidas \mathbf{K} em que cada elemento K_{ij} é definido por $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ para todo $i, j = 1, \dots, n$ (HERBRICH, 2001). As funções *kernel* mais comumente usadas são:

Tabela 2.1 – Funções *kernel*

Tipo	Função $K(\mathbf{x}_i, \mathbf{x}_j)$	Parâmetros
Polinomial	$(\delta(\mathbf{x}_i \cdot \mathbf{x}_j) + \kappa)^d$	δ, κ, d
Gaussiano	$\exp\left(-\frac{\ \mathbf{x}_i - \mathbf{x}_j\ ^2}{2\sigma^2}\right)$	σ
Sigmoidal	$\tanh(\delta(\mathbf{x}_i \cdot \mathbf{x}_j) + \kappa)$	δ, κ

O problema dual reescrito utilizando funções *kernel*:

$$\max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)) \quad (2.12)$$

2.3 Máquina de Vetor de Suporte por Mínimos Quadrados

A LS-SVM é uma modificação da SVM que garante menor esforço computacional sem perda na qualidade das soluções. Nessa formulação, ao invés de QP, se precisa solucionar um sistema de equações lineares. Para chegar a esse resultado foi utilizado uma função de custo de mínimos quadrados e o uso de restrições de igualdade no problema primal (SUYKENS; VANDEWALLE, 1999).

$$\phi(\mathbf{w}, \xi) = \min \frac{1}{2} \|\mathbf{w}\|^2 + \gamma \frac{1}{2} \left(\sum_{i=1}^n e_i^2 \right) \quad (2.13)$$

sujeito à restrição $\{ y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) = 1 - e_i, \forall i = 1, \dots, n$

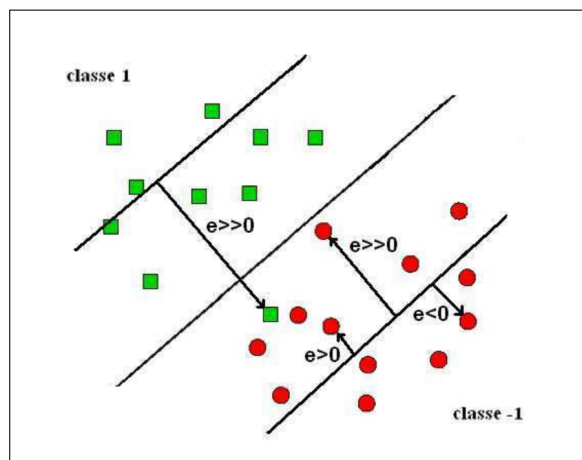
O primeiro termo tem como objetivo maximizar a distância entre os hiperplanos de cada classe. O segundo termo minimiza as variáveis de erro e , sendo γ a constante de regularização, proporcional a C da SVM. As variáveis de erro e funcionam de forma semelhante às variáveis de folga ξ da SVM, entretanto e pode assumir qualquer valor real, diferente de ξ que é sempre positiva.

A variável e_i indica a distância do vetor \mathbf{x}_i à margem correspondente à sua classe. Quanto maior o valor de $|e|$ maior essa distância. Como pode ser visto na Figura 2.4, se $e_i = 0$ ele se encontra exatamente sobre a margem de sua classe. Se $e_i < 0$, ele está aquém da margem da sua classe, portanto classificado corretamente. Se $e_i > 0$, ele está além da margem de sua classe, não havendo como concluir se ele foi ou não classificado corretamente.

A forma dual e a função de classificação são apresentados a seguir:

$$\begin{bmatrix} 0 & 1_n^T \\ 1_n & K + \frac{1}{\gamma} I_n \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix} \quad (2.14)$$

$$y(\mathbf{x}) = \text{sign} \left(\sum_{i=1}^n \alpha_i K(\mathbf{x}_i, \mathbf{x}) + b \right) \quad (2.15)$$

Figura 2.4 – Interpretação geométrica das variáveis de erro e 

Fonte: (CARVALHO, 2005)

Como pode ser observado na Equação 2.15, todos os dados de treinamento são considerados para determinar o hiperplano. Isso se deve ao fato de que os multiplicadores de Lagrange serem proporcionais às variáveis de erro, raramente nulas. Essa é uma desvantagem da LS-SVM quando comparada a SVM, em que só o conjunto de vetores de suporte é considerado nesse cálculo.

2.4 Validação Cruzada

A generalização do modelo é uma etapa essencial no aprendizado de máquina. Na LS-SVM, os hiper-parâmetros são as variáveis que controlam o quão boa é a generalização da máquina. Eles são os parâmetros de *kernel*, que variam de acordo com a função *kernel* escolhida, e o de regularização γ . Selecionar bons parâmetros é uma etapa crucial no aprendizado de máquina, e a CV está intimamente ligada a isso.

Diversos são os métodos e heurísticas que podem ser utilizados como algoritmo de seleção de parâmetros da LS-SVM. Ainda assim, independente de qual deles seja utilizado, a capacidade de generalização de cada parâmetro testado será estimada através da CV.

No algoritmo de CV *l-fold*, um conjunto de n dados é dividido em l subconjuntos de aproximadamente mesmo tamanho. São feitos l treinamentos, cada um deles deixando um subconjunto diferente de fora, o qual será utilizado para computar os erros de classificação. Caso $l = n$, estamos diante de um caso especial do *l-fold*, denominado *leave one out*.

Nessa implementação, treinar o classificador para cada divisão é uma tarefa computacionalmente custosa, principalmente se l é grande. Foi observado, por AN *et al.* (2007), que os conjuntos participantes da etapa de treinamento não são realmente de interesse, apenas os subconjuntos deixados de fora e usados na predição são necessários. Assim, o Algoritmo 1 foi proposto por AN *et al.* (2007) para atender a essas modificações.

Algoritmo 1 Efficient Cross-Validation**Input:** K (matriz *Kernel*), l (Número de *folds*), y (resposta)

$$1: K_\gamma^{-1} \leftarrow \text{inv}(K + \frac{1}{\gamma}I), \quad d \leftarrow 1_n^T K_\gamma^{-1} 1_n$$

$$2: C = K_\gamma^{-1} + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1}$$

$$3: \alpha = K_\gamma^{-1} + \frac{1}{d} K_\gamma^{-1} 1_n 1_n^T K_\gamma^{-1} y$$

$$4: n_k \leftarrow \text{size}(y)/l, \quad y^{(k)} \leftarrow \text{zeros}(l, n_k)$$

5: **for** $k \leftarrow 1, k \leq l$ **do**

$$6: \quad C_{kk} \beta_{(k)} = \alpha_{(k)}$$

$$7: \quad y^{(k)} \leftarrow \text{sign}[y_{(k)} - \beta_{(k)}]$$

$$8: \quad k \leftarrow k + 1$$

$$9: \text{erro} \leftarrow \frac{1}{2} \sum_{k=1}^l \sum_{i=1}^{n_k} |y_i - y^{(k,i)}|$$

Output: *erro*Fonte: (EDWARDS *et al.*, 2013)

Para essa implementação, se faz necessário o cálculo e armazenamento de uma matriz C , gerada a partir da matriz K_γ^{-1} . Na computação das predições, entre as linhas 5 e 8 do Algoritmo 1, apenas as submatrizes C_{kk} , pertencentes à diagonal principal de C , são necessárias. A matriz C consome $O(n^2)$ de memória, enquanto as matrizes C_{kk} consomem, aproximadamente, $O(\frac{n^2}{l^2})$ de memória.

$$C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1l} \\ C_{11}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ C_{1l}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix} \quad (2.16)$$

Outra forma de se calcular a matriz C é através da Equação 2.14. Essa forma será importante no decorrer desse trabalho, por isso será apresentada. Calculando-se a inversa da matriz A , notou-se que a matriz C foi obtida de forma indireta e se encontra na forma de submatriz de A^{-1} . Para se obter C , basta excluir a primeira linha e primeira coluna de A^{-1} .

$$A = \begin{bmatrix} 0 & 1_n^T \\ 1_n & K_\gamma \end{bmatrix} \quad A^{-1} = \begin{bmatrix} C_0 & C_1^T & C_2^T & \cdots & C_l^T \\ C_1 & C_{11} & C_{12} & \cdots & C_{1l} \\ C_2 & C_{11}^T & C_{22} & \cdots & C_{2l} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_l & C_{1l}^T & C_{2l}^T & \cdots & C_{ll} \end{bmatrix} \quad (2.17)$$

2.5 Inversão de Matrizes em Blocos

A inversão de matrizes de ordem extremamente alta tem sido uma tarefa desafiadora devido a limitada capacidade de processamento e memória dos computadores convencionais (COSME *et al.*, 2016). O cálculo da inversão de matrizes em blocos surge como uma

solução para esse problema, permitindo uma elevada economia de memória e com isso expandindo a capacidade do computador em resolver problemas maiores.

No algoritmo denominado *Block Recursive Inverse* (BRI) proposto por COSME *et al.* (2016), a inversa de uma matriz quadrada não-singular M é obtida através do cálculo do complemento de Schur de cada submatriz pertencente a ela. Cada submatriz também deve ser quadrada não-singular de mesma ordem. A seguir, das Equações 2.19 a 2.22 são apresentadas formas equivalentes de se calcular o complemento de Schur. Elas serão aplicadas a sua submatriz correspondente em M .

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad (2.18)$$

$$M/A = D - CA^{-1}B \quad (2.19)$$

$$M/B = C - DB^{-1}A \quad (2.20)$$

$$M/C = B - AC^{-1}D \quad (2.21)$$

$$M/D = A - BD^{-1}C \quad (2.22)$$

Esse algoritmo trabalha recursivamente e sua execução é dividida em duas etapas: Procedimento Recursivo Para Frente (do inglês, *Forward Recursive Procedure* ou FRP), apresentado na Subseção Procedimento Recursivo Para Frente, e o Procedimento Recursivo Para Trás (do inglês, *Backward Recursive Procedure* ou BRP), apresentado na Subseção Procedimento Recursivo Para Trás.

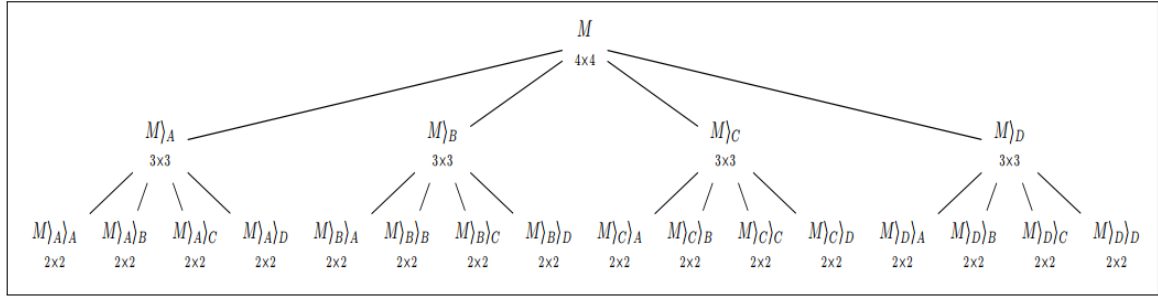
Procedimento Recursivo Para Frente

Enquanto houverem matrizes de dimensão $k > 2$, elas serão particionadas em quatro matrizes quadradas de mesma ordem $k - 1$, chamados *frames*. Esse passo é chamado Procedimento Recursivo Para Frente e está representado graficamente na Figura 2.5. O particionamento em blocos segue algumas regras descritas a seguir.

Partindo da matriz M , seu particionamento resultará em quatro frames. O frame $M_{\rangle A}$ resulta da remoção da linha mais abaixo e da coluna mais a direita de M ; $M_{\rangle B}$ resulta da remoção da linha mais abaixo e da coluna mais a esquerda de M ; $M_{\rangle C}$ resulta da remoção da linha mais acima e da coluna mais a direita de M ; $M_{\rangle D}$ resulta da remoção da linha mais acima e da coluna mais a esquerda de M .

$$M_{\rangle A} = \begin{bmatrix} M_{11} & \cdots & M_{1(k-1)} \\ \vdots & \ddots & \vdots \\ M_{(k-1)1} & \cdots & M_{(k-1)(k-1)} \end{bmatrix} \quad (2.23)$$

Figura 2.5 – Interpretação gráfica do Procedimento Recursivo Para Frente

Fonte: (COSME *et al.*, 2016)

$$M_{\rangle B} = \begin{bmatrix} M_{12} & \cdots & M_{1k} \\ \vdots & \ddots & \vdots \\ M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix} \quad (2.24)$$

$$M_{\rangle C} = \begin{bmatrix} M_{21} & \cdots & M_{2(k-1)} \\ \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{k(k-1)} \end{bmatrix} \quad (2.25)$$

$$M_{\rangle D} = \begin{bmatrix} M_{22} & \cdots & M_{2k} \\ \vdots & \ddots & \vdots \\ M_{k2} & \cdots & M_{kk} \end{bmatrix} \quad (2.26)$$

Depois linhas e colunas de cada bloco serão trocadas até que o bloco M_{22} seja movido para seu lugar original. Nesse caso, no frame $M_{\rangle A}$ não há modificações; em $M_{\rangle B}$ é necessário trocar as duas colunas mais a esquerda; em $M_{\rangle C}$ é necessário trocar as duas linhas mais acima; em $M_{\rangle D}$ trocam-se as duas colunas mais a esquerda e as duas linhas mais acima.

$$M_{\rangle B} = \begin{bmatrix} M_{13} & M_{12} & \cdots & M_{1k} \\ M_{23} & M_{22} & \cdots & M_{2k} \\ M_{33} & M_{32} & \cdots & M_{3k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(k-1)3} & M_{(k-1)2} & \cdots & M_{(k-1)k} \end{bmatrix} \quad (2.27)$$

$$M_{\rangle C} = \begin{bmatrix} M_{31} & M_{32} & \cdots & M_{3(k-1)} \\ M_{21} & M_{22} & \cdots & M_{2(k-1)} \\ M_{41} & M_{42} & \cdots & M_{4(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k1} & \cdots & M_{k(k-1)} \end{bmatrix} \quad (2.28)$$

$$M\big\rangle_D = \begin{bmatrix} M_{33} & M_{32} & \cdots & M_{3k} \\ M_{23} & M_{22} & \cdots & M_{2k} \\ M_{43} & M_{42} & \cdots & M_{4k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k3} & M_{k2} & \cdots & M_{kk} \end{bmatrix} \quad (2.29)$$

A partir desse ponto, o algoritmo segue numa repetição dos dois passos anteriores até que todos os *frames* tenham dimensão $k = 2$. A partir daqui se inicia o Procedimento Recursivo Para Trás.

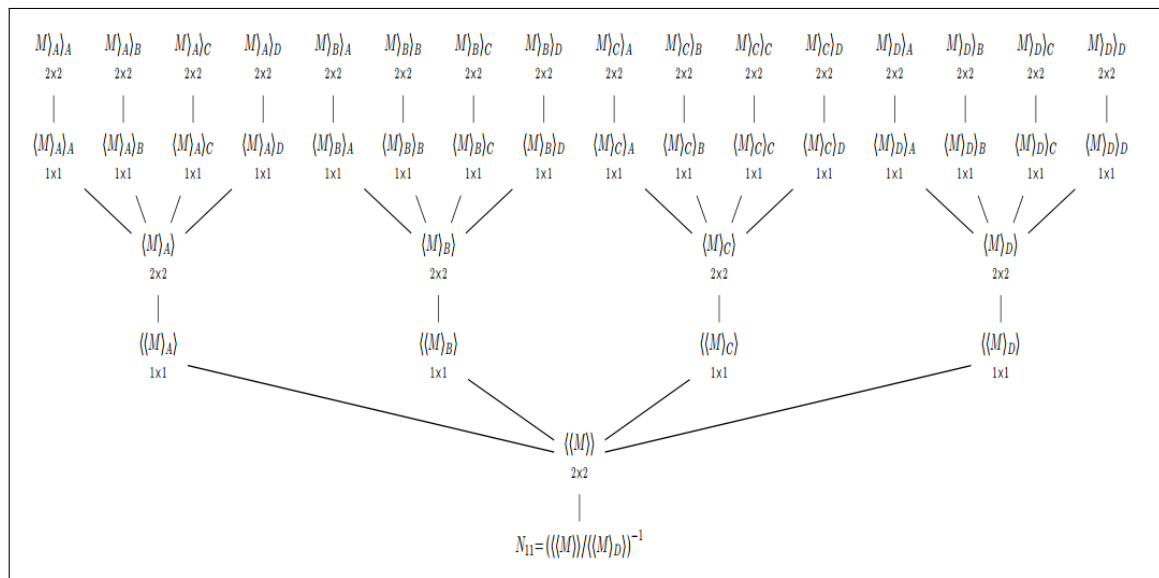
Procedimento Recursivo Para Trás

Nesse passo, partindo-se de uma matriz M_{22} o complemento de Schur é calculado em cada *frame*, aplicando a forma apropriada a posição do *frame*.

Cada *frame* de dimensão $k = 2$, depois de aplicado o complemento de Schur, resultará num *frame* de dimensão $k = 1$. Em seguida, quatro *frames* serão agrupados, formando um *frame* de dimensão $k = 2$ e esse passo é reaplicado. No último passo, apenas a inversa do bloco do canto superior esquerdo é calculado. O BRP está representado graficamente na Figura 2.6.

É importante notar que com as permutações adequadas de linhas e colunas, qualquer bloco pode ser posicionado no canto superior esquerdo de M e assim, ser calculada a inversa desse bloco em M (COSME *et al.*, 2016).

Figura 2.6 – Interpretação gráfica do Procedimento Recursivo Para Trás



Fonte: (COSME *et al.*, 2016)

Capítulo 3

Metodologia

Neste capítulo são apresentados as ferramentas e procedimentos utilizados para realizar os testes, assim como detalhes de suas implementações. Foi utilizado o *software* Matlab R2016b versão *trial* (MATLAB, 2016). O Matlab é um *software* já famoso por suas ferramentas que facilitam o cálculo numérico, algébrico e também de aprendizado de máquina, incluindo o *toolbox* de SVM nesse conjunto de ferramentas. Para o propósito desse artigo, foi utilizado o *toolbox* LS-SVMlab versão 1.8, que é uma biblioteca de LS-SVM para Matlab.

As implementações do BRI foram desenvolvidas em linguagem C++ utilizando a biblioteca especializada em álgebra linear Armadillo. Essa biblioteca provê implementações de alta performance para inversão de matrizes, facilitando o desenvolvimento da ferramenta proposta. Para fazer a integração entre o Matlab e a linguagem C++, foi utilizado o compilador Mex. Todas essas ferramentas e a integração delas será apresentado a seguir.

3.1 Ferramentas

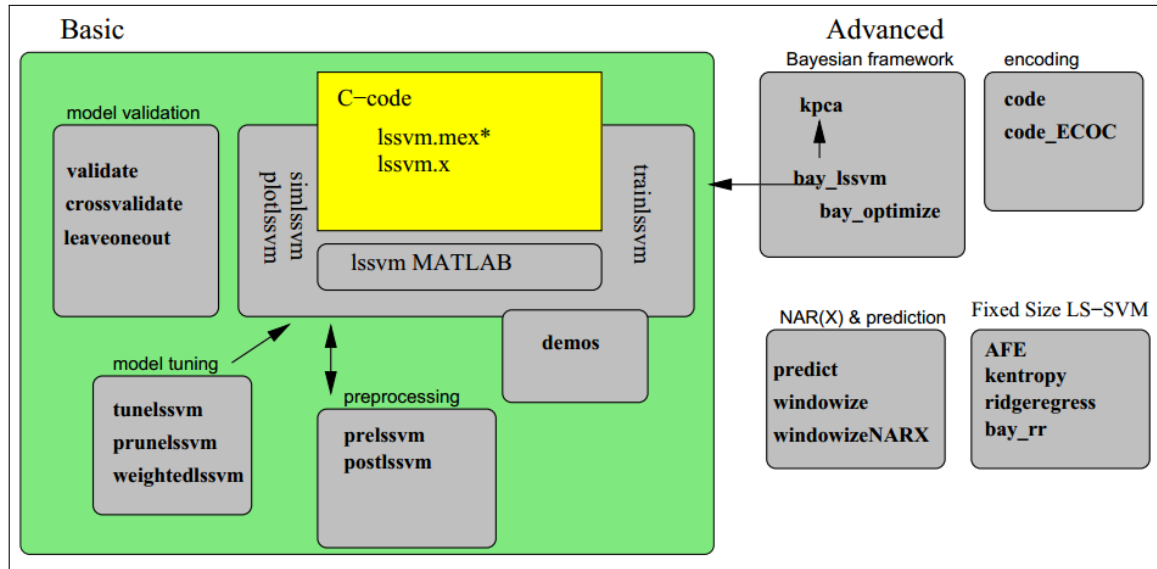
LS-SVMlab

LS-SVMlab é um *toolbox* gratuito e bastante completo que implementa uma LS-SVM usando como *backend* o *software* comercial Matlab. Em desenvolvimento desde meados de 2002, trata-se de uma ferramenta bastante completa que implementa funcionalidades básicas, como treinamento e validação em problemas binários, e também mais avançadas como técnicas de classificação multi-classes e classificação Bayesiana (PELCKMANS *et al.*, 2002). Testado e compilado em Windows e Linux, seu desenvolvimento é acompanhado pelos autores que propuseram a LS-SVM, Suykens e Vandewalle. Um esquemático do fluxo de informações dessa ferramenta pode ser visto na Figura 3.1. À frente será feito uma breve explanação das funções utilizadas nesse trabalho.

No *script* desenvolvido em Matlab, foi utilizada a função *rand()* para se gerar uma distribuição uniforme dos dados de entrada, assim dando origem a um problema de classificação binário. Os dados se encontram no espaço bidimensional para facilitar a visualização gráfica deles e da superfície obtida. Para usar a validação cruzada no *toolbox* é necessário a passagem de algumas informações como tipo do problema (regressão ou

classificação), o tipo de CV (*leave one out* ou *l-fold*), o *kernel* que será utilizado e o algoritmo aplicado na seleção de parâmetros.

Figura 3.1 – Organização do fluxo de informações no LS-SVMLab



Fonte: (PELCKMANS *et al.*, 2002)

Todos esses parâmetros são passados para a função `tunelssvm`. Essa função é a responsável por selecionar os melhores parâmetros de entrada da máquina. Diante da escolha do *kernel* RBF, ela retornará os parâmetros de *kernel* σ e de regularização γ . Como algoritmo de otimização foi utilizado *gridsearch*. Nesse algoritmo, todas as combinações de σ e γ são testados, e é selecionado o conjunto que obteve menor taxa de erro durante os testes. A biblioteca dispõe de outros como *simplex* e o *linesearch*, que é utilizado em *kernel* linear. Apesar do *gridsearch* ser uma implementação força bruta, existe uma alta taxa de confiabilidade associada a esse algoritmo, por isso ele foi escolhido. Para gerar o conjunto de valores de σ e γ que participaram dos testes é utilizada a técnica de otimização *Coupled Simulated Annealing* (CSA). Por padrão, cinco valores são gerados para cada variável.

Para o tipo de CV, foi usada a função `crossvalidatelssvm`. Ela é uma implementação do *l-fold* como proposto no Algoritmo 1. Por padrão, caso o usuário não passe nenhum parâmetro, são utilizados cinco *folds*. Caso o usuário queira usar o método *leave one out*, o *toolbox* tem uma implementação própria para isso, não sendo preciso a passagem de *n folds* como parâmetro para a função `crossvalidatelssvm`. Na CV, a taxa das classificações erradas em cada teste é calculada pela função `misclass`. Esse método é acionado de dentro da função `crossvalidatelssvm`, mas também pode ser chamado pelo *script* do usuário e ser utilizado na fase de teste, logo após o treinamento da máquina.

Usualmente, após o retorno dos parâmetros ótimos pela `crossvalidatelssvm`, é chamada a função `trainlssvm` para treinamento da máquina. Como entrada são passados os dados e os parâmetros selecionados pela CV. Essa função retorna α e b , soluções do pro-

blema *dual* visto na equação 2.14. Para um conjunto de entrada bidimensional, a função *plotlssvm* pode ser chamada para exibir graficamente os dados e a superfície de separação obtida. Após a fase de treinamento, uma fase de testes pode ser programada utilizando a função *simlssvm*. Para ela devem ser passados o conjunto de teste, o modelo e as soluções α e b obtidas no treinamento, sendo retornado por ela o conjunto de predições feitas pela máquina. Como último passo, caso se tenha as saídas esperadas para cada dado do conjunto de teste, pode-se chamar a função *misclass* que calculará a taxa de erro obtida nas predições. Todos os passos descritos encontram-se no Algoritmo 2.

Algoritmo 2 Script de Testes

Input: $[X, Y]$ (conjunto de treinamento), $[X_t, Y_t]$ (conjunto de teste), l (Número de *folds*), K (tipo de Kernel), *type* (tipo de problema)

- 1: $[\gamma, \sigma] \leftarrow \text{tunelssvm}([X, Y], K, \text{type}, \text{gridsearch}, \text{crossvalidatelssvm}, l, \text{misclass})$
- 2: $[\alpha, b] \leftarrow \text{trainlssvm}([X, Y], K, \text{type}, \gamma, \sigma)$
- 3: *plotlssvm*($[X, Y], K, \text{type}, \gamma, \sigma, \alpha, b$)
- 4: $Y_p \leftarrow \text{simlssvm}([X, Y], K, \text{type}, \gamma, \sigma, \alpha, b, X_t)$
- 5: *erro* $\leftarrow \text{misclass}(Y_t, Y_p)$

Output: *erro*

Armadillo

Armadillo é uma biblioteca de álgebra linear em código aberto escrita na linguagem C++ que propõe um bom equilíbrio entre eficiência e facilidade de uso. É uma *Application Programming Interface* (API) que, similar ao Matlab, implementa operações matemáticas usuais expressas para o usuário de forma mais simples e familiar (SANDERSON; CURTIN, 2016).

Para realizar operações matriciais, inclusive inversão de matrizes, a Armadillo usa os pacotes LAPACK e BLAS. BLAS provê o padrão para execução de operações básicas em vetores e matrizes; LAPACK implementa essas rotinas provendo resultados eficientes para soluções de problemas lineares, operações entre matrizes, entre outros. Ambos os pacotes também são utilizados pelo Matlab para realizar operações matriciais. Assim, na inversão de uma matriz A usando Armadillo e/ou Matlab, esperam-se os mesmos resultados.

O BRI se utiliza das funcionalidades da biblioteca Armadillo visando eficiência e rapidez na implementação. Devido a isso, todo o seu código foi desenvolvido em C++, precisando de um intermediário que interligasse o BRI ao LS-SVMLab.

Mex

O Mex é uma interface integrada ao Matlab, que permite que códigos escritos nesse *software* possam chamar rotinas escritas em linguagem C, C++ ou Fortran. A chamada é feita declarando o nome do arquivo fonte, usando a sintaxe de uma chamada de métodos dentro do código Matlab, passando se necessário os parâmetros de entrada e alocando espaço para os parâmetros de retorno.

No código C/C++ é necessário incluir o método *void mexFunction()*, sendo ele o intermediário entre os parâmetros enviados pelo Matlab e os que serão retornados para ele pelo C/C++. Devem ser declarados quatro parâmetros, dois de entrada e dois de saída. Cada conjunto contém um vetor, que guarda os elementos que estão sendo passados por parâmetro, e um inteiro que guarda o número de elementos nesse vetor. Apesar de ser um método por definição sem retorno, o Matlab tem acesso às variáveis de saída, e o C/C++ acesso às variáveis de entrada. Devido a isso, a comunicação é feita sem complicações.

3.2 Implementação

A proposta desse artigo é comparar a eficiência de obtenção da matriz C , apresentada na equação 2.16, usando o algoritmo padrão do *toolbox* LS-SVMlab, que a obtém a partir do processo de inversão da matriz A inteira pelo método LU; ou através de uma inversão parcial, bloco por bloco, usando o BRI. Diante disso, a primeira preocupação é a adequação do BRI aos objetivos desse trabalho.

O BRI é um algoritmo recursivo dividido em dois passos. No passo FRP, enquanto a matriz tiver dimensão $k > 2$, ela será quebrada em quatro blocos de dimensão $(k - 1)$ até que enfim, se obtenham blocos de dimensão $k = 2$. No passo BRP é aplicado o complemento de Schur em cada bloco, que resultará em dimensões $k = 1$. Os blocos resultantes se juntarão quatro em quatro para formar novos blocos de dimensão $k = 2$ e o passo se repete, até que se resulte em apenas um bloco, a partir do qual será calculada a inversa. Esta será a saída do BRI e corresponderá ao bloco superior à esquerda da matriz inversa.

Observa-se que no BRI, apenas o bloco superior a esquerda é obtido. Esse método não atende totalmente aos propósitos desse artigo, visto que devem ser obtidos todos os blocos C_{kk} pertencentes a diagonal principal de C . Mas COSME *et al.* (2016) frisam que através de permutações em linhas e colunas é possível obter qualquer bloco na posição superior a esquerda. Para atender a essa alteração, o código foi modificado e nele foi implementado uma estrutura que guarda a posição de cada bloco dentro da matriz A . Assim, depois de invertida, pode-se facilmente retornar o bloco que a iteração do *l-fold* necessita.

Como parâmetros de entrada para essa modificação do BRI devem ser passados o número de *folds*, o tamanho de cada bloco, as dimensão dos dados de entrada, os parâmetros para o kernel RBF γ e σ , e a iteração atual do *l-fold*. Multiplicando-se o número de *folds* pelo tamanho de cada bloco, obtém-se o número de elementos da matriz A . Essa informação é importante, pois a matriz A não é passada por parâmetro, mas somente a matriz X de entrada. Para gerar a matriz A a partir da matriz X deve-se ter o tipo de *kernel* e os parâmetros γ e σ que estão sendo testados na *crossvalidatelssvm*. Além disso, a iteração atual do *l-fold* indicará qual bloco da matriz C deve ser retornado para o LS-SVMlab, visto que para cada iteração k o bloco da diagonal principal C_{kk} é o bloco requisitado pelo *toolbox*.

Para que o BRI se comunicasse com o LS-SVMlab, foi implementada a interface Mex no código C++. Uma integração entre Armadillo e Mex também foi necessária, visto a necessidade da conversão de tipos do Matlab pro Armadillo. O Armadillo dispõe de um template que facilita essa integração. Após as modificações no BRI, a função

crossvalidatelssvm foi modificada para os testes comparativos.

Na *crossvalidatelssvm*, foram feitas modificações visando a melhoria da execução dos testes. Foram realizados testes comparativos de consumo de tempo e memória entre a CV padrão e o BRI. Como foi visto, o Algoritmo 1 contém todos os passos necessários para a CV, entretanto isso vai além da inversão de matrizes, objetivo desse trabalho. Por isso, o algoritmo foi modificado sendo mantida apenas a inversão usando Matlab e sendo implementada uma chamada ao BRI em cada iteração do *l-fold*, para cálculo de cada matriz C_{kk} . Essas modificações resultaram no Algoritmo 3 visto a seguir.

Algoritmo 3 Efficient Cross-Validation Modificado

Input: A (matriz A), X (dados de entrada), l (Número de *folds*)

```

1:  $C \leftarrow inv(A)$ 
2: for  $k \leftarrow 1, k \leq l$  do
3:    $C_{kk} \leftarrow BRI(X)$ 
4:    $k \leftarrow k + 1$ 

```

O algoritmo proposto não realiza todos os passos da CV, impedindo que ela seja finalizada com sucesso. Isso acontece porque não é foco deste trabalho executar comandos extras aos necessários para os testes comparativos, como cálculos de α e β , sendo mais ágil ao obter as medições de tempo e memória num comparativo entre as linhas 1 (inversa padrão) e 3 (BRI) do algoritmo.

Capítulo 4

Resultados e Discussão

Neste capítulo, os resultados experimentais serão apresentados. Todos os testes foram feitos num *laptop* com processador Intel Core i3 1.7Ghz e 4Gb de memória RAM. Foram feitos testes para o *l-fold* modificado, como mostrado no Algoritmo 3, com 3, 4, 5 e 6 *folds* usando o *kernel* RBF. Como já foi mostrado, dentre as funções *kernel*, a RBF é a função que requer o menor número de parâmetros, e como o *gridsearch* testa todas as combinações possíveis entre os conjuntos de parâmetros, por precisar de apenas dois, a CV usando *kernel* RBF é executada mais rapidamente.

Os testes realizados apresentam um comparativo entre o BRI e a CV padrão, em consumo de memória e tempo gasto. Como pode ser notado, testes usando a técnica *leave-one-out* não foram realizados. Esse fato se deve principalmente às limitações de *hardware* propostas, entretanto esse algoritmo é um caso especial do *l-fold* usando *n-folds* como entrada. Por isso, as conclusões e estudos obtidos aqui também podem ser estendidos para esse algoritmo.

Todos os testes foram executados cinco vezes, em seguida a média das execuções foi realizada e os valores catalogados. As medições foram feitas usando a ferramenta *profiler* do Matlab. Na Tabela 4.1, são apresentados os resultados obtidos no comparativo de tempo. Para a CV padrão, o tempo obtido é referente a uma chamada à função *inv()* do Matlab. Esse procedimento faz sentido, pois na CV padrão, a inversa de A precisa ser calculada apenas uma vez. Depois de obtido C, a cada iteração apenas operações

Tabela 4.1 – Comparativo de Tempo

Qntd. de dados (n)	Padrão	BRI			
		l=3	l=4	l=5	l=6
1000	0.94s	2.06s	6.11s	16.2s	54.33s
2000	7.88s	8.55s	21.43s	63.35s	205.17s
3000	24.04s	26.5s	62.26s	157.46s	495.33s
4000	58.83s	58.95s	136.25s	344.33s	1017.84s
5000	111.18s	110.92s	243.82s	652.65s	1866.21s
6000	202.17s	193.25s	411.64s	1089.6s	3112.96s
7000	340.56s	275.87s	638.36s	1999.1s	4787.66s
8000	501.95s	425.91s	961.04s	2641.1s	6959.88s
9000	899.15s	570.17s	1216.54s	3715.7s	9747.36s

Tabela 4.2 – Comparativo de Memória

Qntd. de dados (n)	Padrão	BRI			
		l=3	l=4	l=5	l=6
1000	23003.34Kb	872.39Kb	489.66Kb	312.64Kb	286.94Kb
2000	91523.9Kb	4048.69Kb	1954.77Kb	1251.15Kb	396.77Kb
3000	209443.21Kb	6205.81Kb	3813.66Kb	2819Kb	1761.59Kb
4000	374902.87Kb	13882.06Kb	6877.48Kb	4979.64Kb	2538.52Kb
5000	585905.65Kb	20282.06Kb	11254.72Kb	7831.75Kb	4972.66Kb
6000	926485.3Kb	28526.56Kb	16682.95Kb	11244.79Kb	7806.03Kb
7000	1148460.61Kb	42523.7Kb	23929.53Kb	15347.76Kb	10631.22Kb
8000	1837536.78Kb	55530.8Kb	31273.47Kb	20063.33Kb	13632.33Kb
9000	3307564.8Kb	70355.75Kb	39574.3Kb	25383.62Kb	17690.23Kb

de leitura são feitas para obter os blocos C_{kk} . Para o BRI, são realizadas l chamadas ao algoritmo, cada uma responsável por obter o bloco C_{kk} referente aquela iteração, sendo listado o tempo total da computação do laço *for* da linha 2, no Algoritmo 3.

Na Tabela 4.2, são apresentados os resultados obtidos no comparativo de consumo de memória. Na CV padrão, apenas uma chamada é computada; já no BRI, embora l chamadas sejam realizadas, o consumo de memória delas permanece constante, visto que o espaço alocado para o bloco C_{kk} não muda. Por isso, nesse teste não precisou ser computado todo o laço *for* para se obter o resultado, mas apenas uma iteração dele. Deve ser lembrado que essa é uma abordagem adotada para os testes, e essas otimizações de código não poderiam ser usadas caso se quisesse obter os resultados da CV em si.

No comparativo com três *folds* verifica-se que os tempos entre as duas técnicas são praticamente idênticos, com a CV padrão obtendo os melhores tempos quando $n < 5000$. Nota-se que à medida que o tamanho do problema aumenta, o BRI melhora a diferença de tempos, conseguindo a partir de $n = 5000$ os melhores tempos. Esses resultados demonstram a tendência de que, para problemas que tenham grandes bases de dados, o BRI conseguiria fazer o trabalho de forma mais ágil que a CV padrão. Mas é no comparativo de memória que o BRI mostrou todo seu potencial. Independente do volume de dados testado, em média ele reduziu a memória alocada em 95% quando comparado a CV padrão.

Analisando a complexidade espacial do BRI, entende-se porque ele provê tamanha economia de memória. Na CV padrão, são gastos $O(n^2)$ de memória para armazenar a matriz C ; no BRI, $O(\frac{n^2}{l^2})$ são gastos em armazenamento do bloco C_{kk} . Conclui-se que, aumentando o número de *folds* l usados no algoritmo, também se amplia a economia de memória obtida no BRI. Nos próximos testes essa tendência é provada. Com seis *folds* foi obtida uma economia de 99% de memória.

Por outro lado, analisando-se os comparativos de tempo, verifica-se que o aumento de l resulta em um aumento do tempo total quando se usa o BRI. Esse resultado já era esperado, pois, com mais iterações no algoritmo, mais chamadas são feitas ao BRI. Comparativos de espaço e tempo geram resultados inversamente proporcionais, em que a medida que se melhora os resultados de um, deve-se lidar com a piora dos resultados do outro.

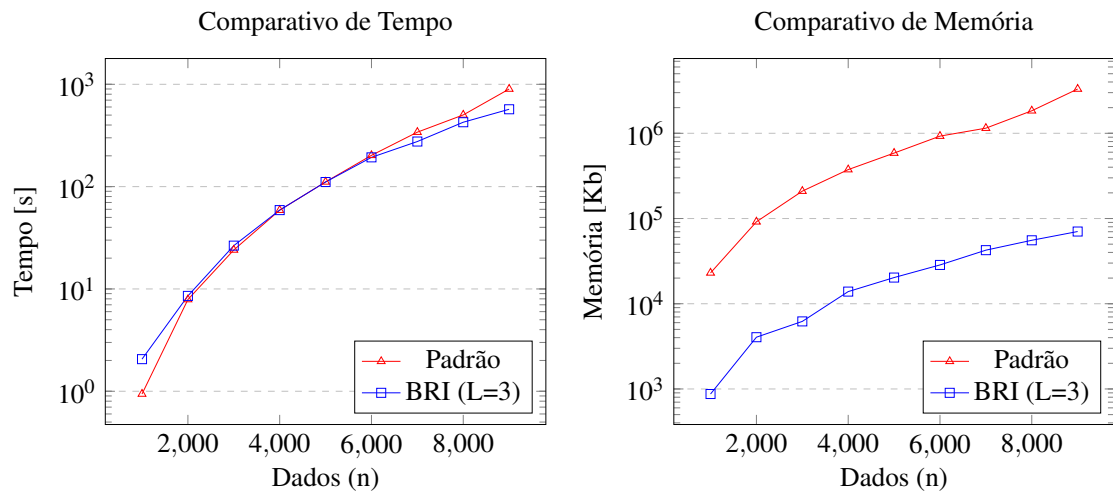


Figura 4.1 – Comparação em tempo e memória total gastos com 3 folds entre o CV padrão e o BRI

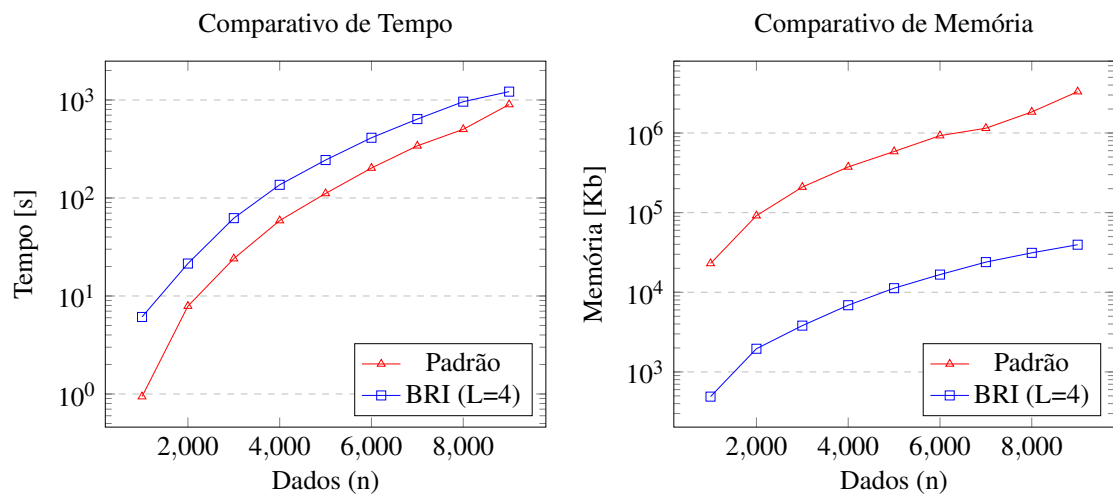


Figura 4.2 – Comparação em tempo e memória total gastos com 4 folds entre o CV padrão e o BRI

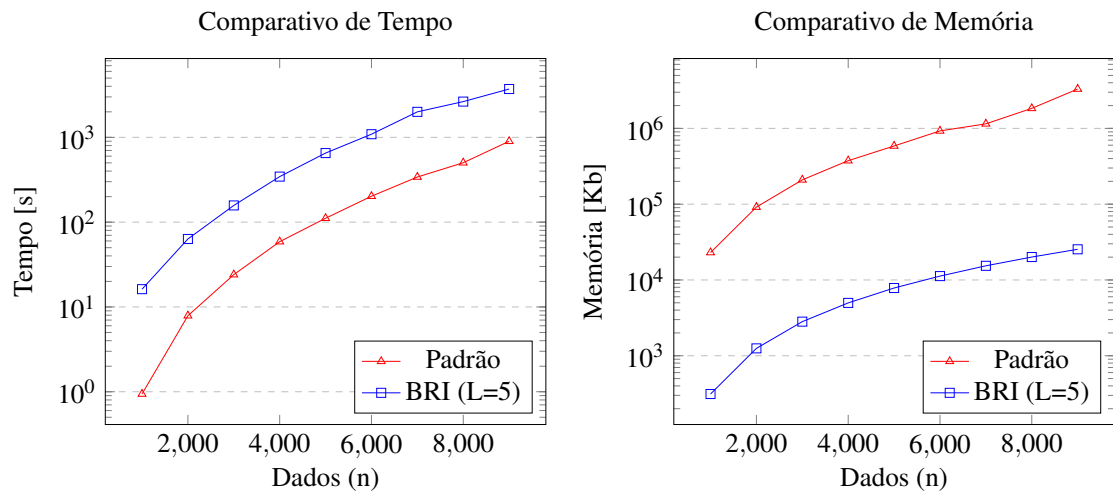


Figura 4.3 – Comparação em tempo e memória total gastos com 5 folds entre o CV padrão e o BRI

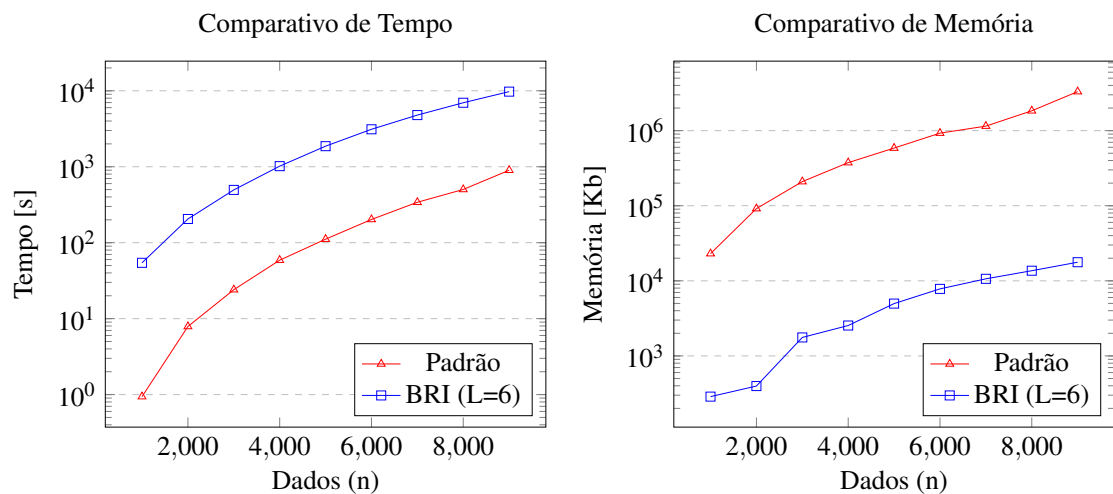


Figura 4.4 – Comparação em tempo e memória total gastos com 6 folds entre o CV padrão e o BRI

Mudanças no número de *folds* não geram alterações na performance da CV padrão.

No *hardware* descrito, a CV padrão apresentou problemas ao tentar computar conjuntos de entrada com mais de 10000 pontos. Esses testes resultaram em falta de memória. Devido à economia de memória alcançada, o BRI não teve esse mesmo problema, conseguindo computar conjuntos de entrada bem maiores que esse. Assim, foi mostrado que o aumento no número de *folds* reduz o gasto de memória, mas provoca o aumento do tempo total de computação. Entretanto, feito o comparativo para grandes volumes de dados, verifica-se que à medida que n aumenta, diminui-se a diferença de tempos entre a CV padrão e o BRI.

Cabe ao pesquisador fazer uma boa análise do tamanho do problema que ele está lidando e do *hardware* disponível para executá-lo. Esse estudo o fará selecionar um bom valor para l . Caso o volume de dados do problema seja grande, provou-se que o BRI consegue uma grande economia de memória com tempos bastante semelhantes ao algoritmo padrão.

Capítulo 5

Conclusão

A Validação Cruzada é uma técnica de grande importância no aprendizado de máquina, mas seu alto custo em tempo e memória limita o número de problemas que podem ser tratados numa máquina comum. Na LS-SVM, esse custo elevado está ligado à inversão da matriz de *kernel* de complexidade espacial $O(n^2)$. Para minimizar o gasto de memória, a inversão de matrizes em bloco foi proposta como solução desse problema.

Nos resultados obtidos, notou-se que, independente do tamanho do problema, obteve-se uma economia de mais de 95% de memória, e essa economia se alarga ainda mais quando se aumenta o número de *folds* usado na CV. Também se verificou que quanto maior o número de dados, mais otimizado o algoritmo se torna, obtendo tempos similares à implementação padrão.

Essa solução se mostra bastante promissora, pois expande a capacidade de resolução de problemas da máquina principalmente em *hardwares* mais limitados. Com um custo $O(\frac{n^2}{l^2})$ de memória, o usuário poderá escolher o número de *folds* que mais se adequa a seu *hardware*, evitando os habituais problemas de falta de memória.

Referências Bibliográficas

AN, S.; LIU, W.; VENKATESH, S. Fast cross-validation algorithms for least squares support vector machine and kernel ridge regression. *Pattern Recognition*, v. 40, p. 2154–2162, 2007.

CARVALHO, B. de. Novas estratégias para detecção automática de vetores de suporte em least squares support vector machines. 2005.

CAWLEY, G. C.; TALBOT, N. L. Fast exact leave-one-out crossvalidation of sparse least-squares support vector machines. *Neural Networks*, v. 17, p. 1467–1475, 2004.

COSME, I. C. S.; FERNANDES, I. F.; CARVALHO, J. L. de; XAVIER-DE-SOUZA, S. Block recursive matrix inverse. 2016. Acesso em 01 Dez 2016. Disponível em: <<https://arxiv.org/abs/1612.00001>>.

DEVAKUNCHARI, R. Analysis on big data over the years. *International Journal of Scientific and Research Publications*, v. 4, jan. 2014.

DUAN, K.; KEERTHI, S.; POO, A. N. Evaluation of simple performance measures for tuning SVM hyperparameters. *Neurocomputing*, v. 51, p. 41–59, 2003.

EDWARDS, R.; ZHANG, H.; PARKER, L.; NEW, J. Approximate l-fold cross-validation with least squares SVM and kernel ridge regression. *12th International Conference on Machine Learning and Applications*, 2013.

HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. [S.l.]: Prentice-Hall, 1999. v. 2.

HEARST, M. A.; SCHÖLKOPF, B.; DUMAIS, S.; OSUNA, E.; PLATT, J. Trends and controversies - support vector machines. *IEEE Intelligent Systems*, p. 18–28, 1998.

HERBRICH, R. *Learning Kernel Classifiers: Theory and Algorithms*. [S.l.]: MIT Press, 2001.

LORENA, A. C.; CARVALHO, A. C. P. L. F. de. Uma introdução às support vector machines. *RITA*, v. 14, 2007.

LUENBERGER, D. G. *Introduction to Linear and Nonlinear Programming*. [S.l.]: Addison-Wesley, 1973.

MATLAB. *R2016b*. Natick, Massachusetts: The MathWorks Inc., 2016.

MJOLSNESS, E.; DECOSTE, D. Machine learning for science: State of the art and future prospects. *Science*, v. 293, 2001.

PADILHA, C. A. A. Algoritmos genéticos aplicados a um comitê de LS-SVM em problemas de classificação. 2013.

PELCKMANS, K.; SUYKENS, J.; GESTEL, T. V.; BRABANTER, J. D.; LUKAS, L.; HAMERS, B.; MOOR, B. D.; J.VANDEWALLE. LS-SVMlab: a MATLAB/C toolbox for least squares support vector machines. 2002.

SANDERSON, C.; CURTIN, R. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, v. 1, 2016.

SUYKENS, J.; VANDEWALLE, J. Least squares support vector machine classifiers. *Neural Process*, v. 9, p. 293–300, 1999.

VAPNIK, V. *The Nature of Statistical Learning Theory*. New York: Springer, 1995.