International Master in Computer Vision

Fundamentals of machine learning for computer vision

Eva Cernadas

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

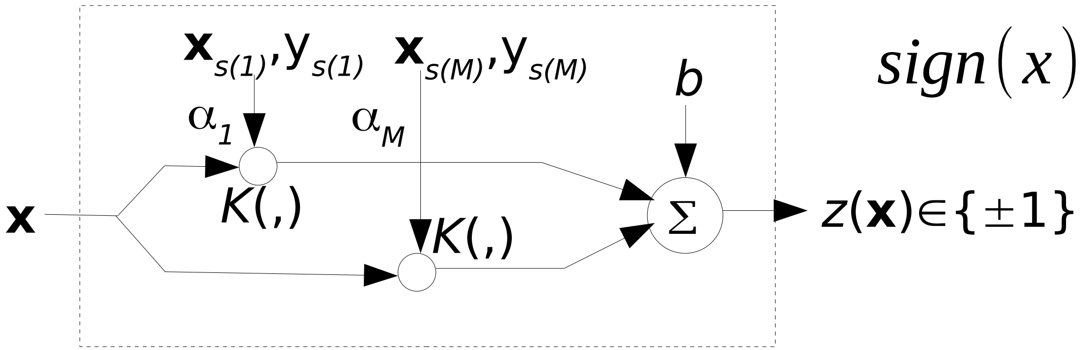
Escola Técnica Superior de Enxeñaría

Contents

- **Machine learning theory (Dr. Jaime Cardoso)**
- **Linear regression and optimization (Dr. Jaime Cardoso)**
- **Model selection and evaluation**
- **Classical classification models**
- **Artificial neural networks (ANN)**
- **Support vector machines (SVM)**
- **Ensembles: bagging, boosting and random forest**
- **Clustering**

Support Vector Machine (SVM)

- **SVM for classification (SVC) :** binary classification (2 classes)



$$sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

$$z(\mathbf{x}) = sign\left(\sum_{i=1}^{M} \alpha_i y_{s(i)} K[\mathbf{x}_{s(i)}, \mathbf{x}] + b \right)$$

Step function

- Linear classifier $z(\mathbf{x}) = sgn(\mathbf{w}^T \mathbf{x} + b)$ in the hidden space. The output $z(\mathbf{x})$ and the true outputs y_i are -1 for one class and +1 for the another; $\mathbf{x}_{s(1)} \ldots \mathbf{x}_{s(M)}$ are the $M<N$ support vectors

Support Vector Machine (SVM)

- Trainable parameters: $\{\alpha_i\}_{i=1}^{M}$, b
- *Tunable hyper-parameters:* regularization (λ) and hyper-parameters of the kernel K
- The SVM combines two ingredients:

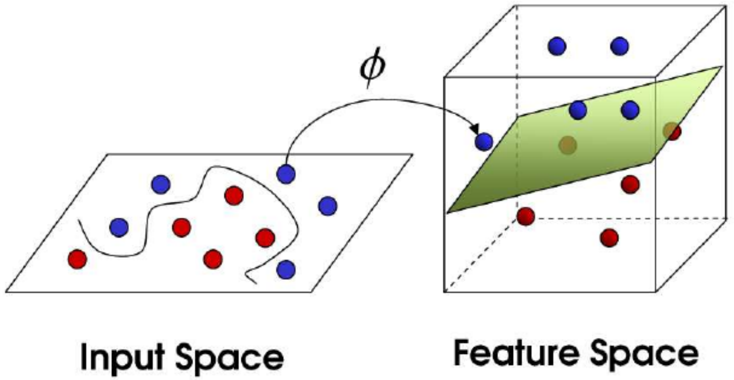
1) A kernel K to project into a hidden space were linear separability is increased.
2) Selection of the linear classifier in the hidden space that minimizes overfitting, maximazing the margin.

SVM: kernel (I)

- Cover theorem (1965): the probability that N patterns in a n-dimensional space will be linearly separable is 1 if $n>N$, and $\frac{1}{2^N}\sum_{i=1}^{n}\binom{N-1}{i}$ if $n<N-1$ (this function increases with n).

- Increasing the dimensionality n of data increases their probability to be linearly separable

- Kernel function: maps the input space $\mathbb{R}^n$ to $\mathbb{R}^m$ with $m>n$: $\mathbf{x} \rightarrow \Phi(\mathbf{x})$: Φ is a vector function



SVM: kernel (II)

- The linear separability is more probable in the $\mathbb{R}^m$ *space (hidden or feature space)* than in the original space.
- The kernel Φ verifies the condition $\Phi(\mathbf{x})^T\Phi(\mathbf{y})=K(\mathbf{x},\mathbf{y})$: Φ is a generalized scalar product).
- $K(\mathbf{x},\mathbf{y})$ is a generalized similarity measure between **x** and **y** in the hidden space.
- The SVM is a linear classifier defined by vector **w** in the hidden space created by the kernel mapping $\Phi(\mathbf{x})$.

SVM: kernel (III)

- We will see that $\boldsymbol{w}=\sum_{i=1}^{M}\alpha_i y_{s(i)}\boldsymbol{\Phi}[\boldsymbol{x}_{s(i)}]$, with M support vectors $\{\mathbf{x}_{s(i)}\}_{i=1}^{M}$
- So:

$$z(\boldsymbol{x})=sign(\boldsymbol{w}^T\boldsymbol{\Phi}(\boldsymbol{x})+b)=sign\left(\sum_{i=1}^{M}\alpha_i y_{s(i)}\boldsymbol{\Phi}[\boldsymbol{x}_{s(i)}]^T\boldsymbol{\Phi}(\boldsymbol{x})+b\right)$$

$$z(\boldsymbol{x})=sign\left(\sum_{i=1}^{M}\alpha_i y_{s(i)}K[\boldsymbol{x}_{s(i)},\boldsymbol{x}]+b\right)$$

where $K(\mathbf{x}_{s(i)},\mathbf{x})=\Phi[\mathbf{x}_{s(i)}]^T\Phi(\mathbf{x})$ and $y_i\in\{\pm 1\}$

- There are several kernels with different hyper-parameters, that should be tuned for each problem.

SVM: kernel (IV): types

- **Gaussian or RBF kernel**:

$$K(\boldsymbol{v},\boldsymbol{w})=\exp\left(\frac{-|\boldsymbol{v}-\boldsymbol{w}|^2}{2\sigma^2}\right)$$

  The spread σ is the tunable hyper-parameter, with recommended values $\{2^i\}_{-5}^{10}$. The hidden space has infinite dimension.

- ***Polynomial kernel**:* $K(\mathbf{v},\mathbf{w})=(\mathbf{v}^T\mathbf{w}+a)^b$: tunable parameters a,b: degree b=1,2,3 and offset a with values between $-n$ and $+n$, being n the upper bound of $\mathbf{v}^T\mathbf{w}$ so that $|\mathbf{v}^T\mathbf{w}|<n$. Hidden space of finite (high) dimension.

- No kernel means $\Phi(\mathbf{x})=\mathbf{x}$ or ***lineal kernel*** $K(\mathbf{v},\mathbf{w})=\mathbf{v}^T\mathbf{w}$: the hidden space is the input space, the SVM is a linear classifier.

SVM: kernel (V)

- The ***Gaussian kernel*** is normally the best performing, when the spread σ is tuned
- The SVM performance exhibits a peak for the best σ value, and lower values for σ values low and high
- The SVM performance with linear kernel is lower than Gaussian kernel
- With large values of *n* (high-dimensional data), both kernels have similar performance, because the mapping to high-dimensions is no longer required

Minimizing overfitting (I)

- Statistical learning theory (V. Vapnik). The **Vapnik-Chervonenkis dimension** (*h*) of a binary classifier is defined as: the maximum number of patterns that it can learn without making mistakes, independently of the class label.
- *h* measures the classifier complexity: the higher *h*, the larger overfitting: it must be minimized.
- For a linear classifier in a *n*-dimensional space: $h \leq n+1$.

Minimizing overfitting (II)

- If the patterns **x** satisfy |**x**|<*D* and ρ is the margin (minimum distance between **x** and the classifier hyperplane):

$$h \leq min\left(\left\lceil \frac{D}{\rho^2} \right\rceil, n\right) + 1$$

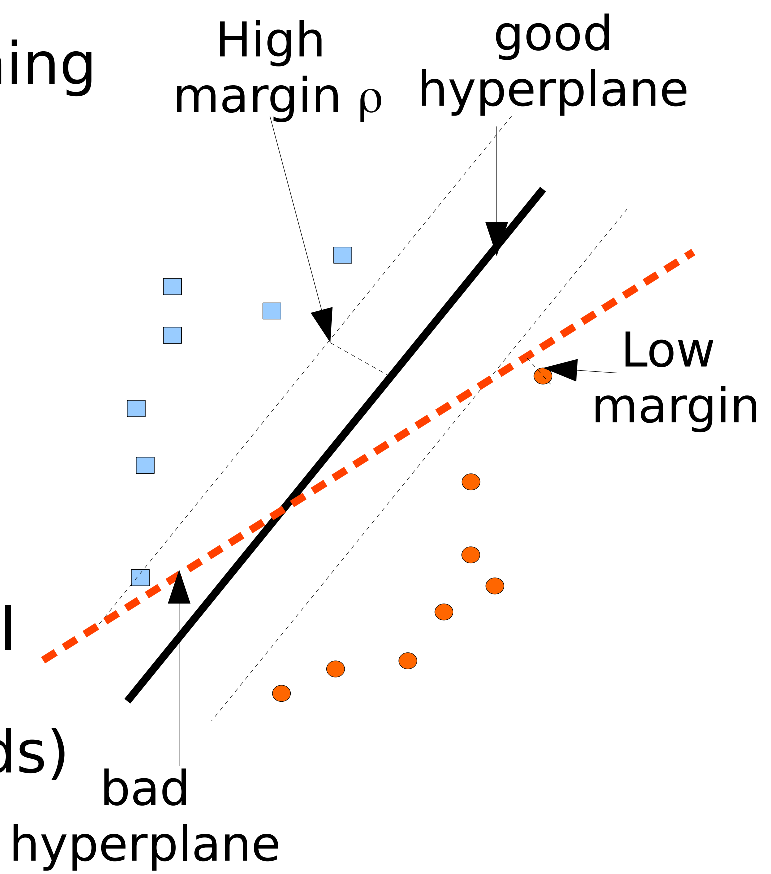
- You must maximize the margin ρ in order to minimize overfitting of the linear classifier (hyperplane) in the hidden space.

SVM training (I)

- The margin ρ in the hidden space is:

$$\rho = \min_{i=1\ldots N} \frac{\left| \mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i) + b \right|}{|\mathbf{w}|}$$

- Requiring $|\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i)+b| \geq 1$ for all the training patterns $\mathbf{x}_i$, the margin is $\rho=1/|\mathbf{w}|$



- The training error for $\mathbf{\Phi}(\mathbf{x}_i)$ is $\xi_i=\max[0,1-y_i(\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i)+b)]$

- $\xi_i>0$ when $\mathbf{\Phi}(\mathbf{x}_i)$ is missclassified or well classified but $\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i)+b<1$ (inside bands)

SVM training (II)

- λ=regularization parameter. The hyperplane (**w**,*b*) must minimize:

$$J(\mathbf{w}, b, \vec{\xi}) = \frac{|\mathbf{w}|^2}{2} + \lambda \sum_{i=1}^{N} \xi_i$$

with the conditions:

$$\begin{cases} \mathbf{w}^T \mathbf{x}_i + b \geq -1, y_i = -1 \\ \mathbf{w}^T \mathbf{x}_i + b \geq 1, y_i = +1 \end{cases}$$

$$\xi_i \geq 0, \mathbf{w}^T \mathbf{x}_i + b \geq y_i (1 - \xi_i), i = 1 \ldots N$$

- The Lagrange multipliers $\{\alpha_i, \beta_i\}_{i=1}^{N}$ and function *L are used as optimization method with constrains*:

$$L(\mathbf{w}, b, \vec{\xi}, \vec{\beta}, \vec{\alpha}) = \frac{|\mathbf{w}|^2}{2} + \lambda \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i [y_i (\mathbf{w}^T \vec{\Phi}(\mathbf{x}_i) + b) - 1 + \xi_i] - \sum_{i=1}^{N} \beta_i \xi_i$$

SVM training (III)

- Deriving with respect to **w** and equaling to **0**:

$$\boldsymbol{w}=\sum_{i=1}^{N}\alpha_i y_i \vec{\Phi}(\boldsymbol{x}_i)$$

- The vector **w** is a linear combination of the training patterns. Not scalable to high *N* (many patterns).

- Solution: as we will see $\alpha_i=0$ for many *i* (sparse solution).

- Deriving with respect to b, ξ_i, α_i and β_i, the problem is transformed into finding $\vec{\alpha}=(\alpha_1,\ldots,\alpha_M)$ that maximizes:

$$\vec{\alpha}^T\mathbf{1}-\frac{\vec{\alpha}^T\boldsymbol{K}\vec{\alpha}}{2}$$

1 and α: column vectors

SVM training (IV)

where $\mathbf{K}=(K_{ij})_{i,j=1}^{N}$ and $K_{ij}=K(\mathbf{x}_i^T, \mathbf{x}_j)$, with the conditions:

$$\vec{\alpha}^T \boldsymbol{y}=0, 0\leq\alpha_i\leq\lambda, \beta_i\xi_i=0, \alpha_i\left\{y_i\left[\sum_{j=1}^{N}\alpha_i K\left(\boldsymbol{x}_j^T, \boldsymbol{x}\right)+b\right]\right\}=0, i=1,\ldots N$$

- This optimization problem is solved using iterative numeric procedures.
- The SVM only requires $M<N$ training patterns (support vectors) $\mathbf{x}_{s(1)}\ldots\mathbf{x}_{s(M)}$, for which $0\leq\alpha_i\leq\lambda$, being $\alpha_i=0$ for the remaining patterns.
- The vector **w** in the hidden space is: $\boldsymbol{w}=\sum_{i=1}^{M}\alpha_i y_{s(i)}\vec{\Phi}\left[\boldsymbol{x}_{s(i)}\right]$

SVM training (VI)

- The offset b is: $b = y_j - \sum_{i=1}^{M} \alpha_i y_{s(i)} K[\boldsymbol{x}_j, \boldsymbol{x}_{s(i)}]$

  being $\mathbf{x}_j$ a support vector.

- Substituting $\mathbf{w}$ in $z(\mathbf{x})=sign(\mathbf{w}^T\Phi(\mathbf{x})+b)$ and using that $\Phi(\mathbf{v})^T\Phi(\mathbf{w})=K(\mathbf{v},\mathbf{w})$, we achieve the final expression of the SVM output:

$$z(\boldsymbol{x}) = sign\left(\sum_{i=1}^{M} \alpha_i y_{s(i)} K[\boldsymbol{x}_{s(i)}, \boldsymbol{x}] + b\right)$$

- The SVM suffers less the **curse of dimensionality** (poor performance with high-dimensional input patterns) than other classifiers.

Tunable hyper-parameters

- λ (regularization parameter): values $2^{-5}..2^{15}$: the results are not very sensitive to its value. A default value (when tuning is not possible) would be $\lambda=1$ or $\lambda=100$.
- With Gaussian kernel: σ (kernel spread): values $2^{-5}..2^{10}$: very important in the results: the best value is normally in the median of the σ range. A default value would be $\sigma=1/n$.
- The SVM performance is much higher developing a hyper-parameter tuning.

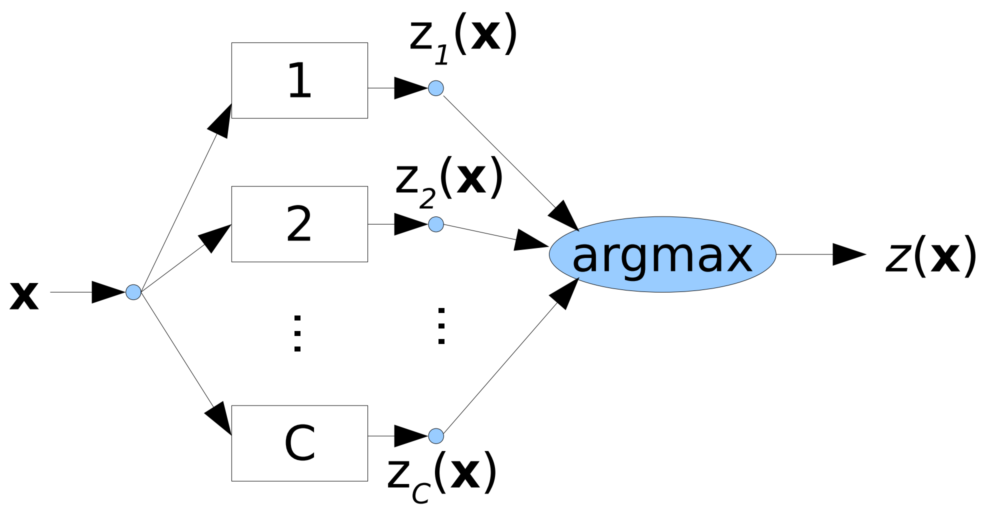
Multi-class SVM classification (I)

- ***One-vs-all* (OVA)** and **one-vs-one (OVO)** approaches
- For high C, use ***one-vs-all* (OVA)** approach: C binary SVMs, where the i-th SVM classifies the patterns between class i and the remaining classes
- The i-th SVM trains with all patterns: y_j=1 for the training patterns $\mathbf{x}_j$ of class i, and y_j=-1 for patterns of the remaining classes
- More efficient because uses only C binary SVMs. Lower performance.

Multi-class classification (II)

- All the SVMs share the λ and σ values.
- $\alpha_{s(i,j)}$, $\mathbf{x}_{s(i,j)}$: j-th coefficient and support vector of i-th binary SVM

$$z_i(\mathbf{x}) = \sum_{j=1}^{M_i} \alpha_{s(i,j)} y_{s(i,j)} K[\mathbf{x}_{s(i,j)}, \mathbf{x}] + b_i$$



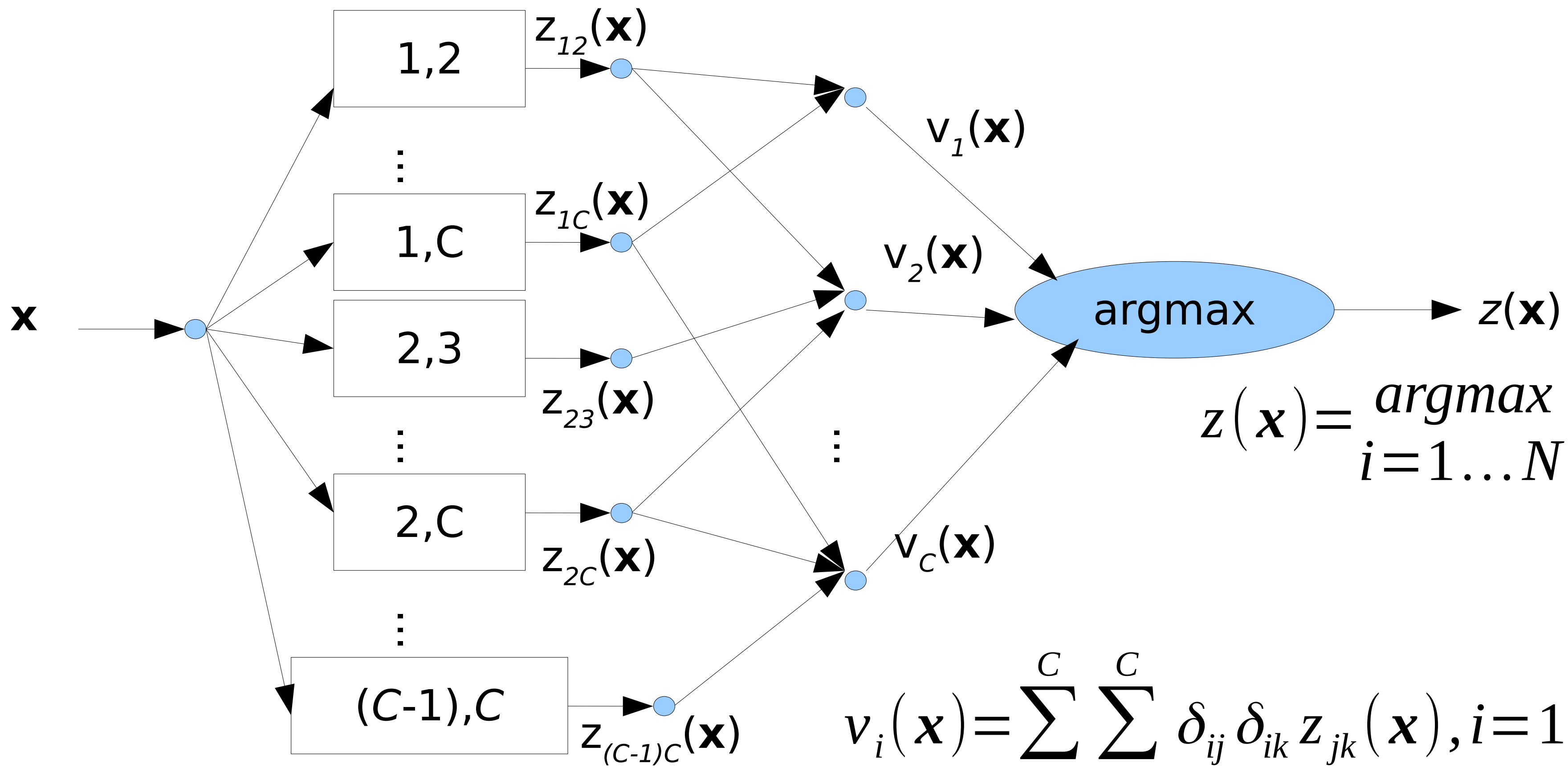
Multi-class classification (III)

- If C>2 is low, use ***one-vs-one* (OVO)** approach. You will need $C(C$-1)/2 binary SVMs. Less efficient because the number of binary SVMs raises with C^2. Better performance.

- The ij-th binary SVM classifies between patterns of class i and j, with i=1..C-1 and j=i+1..C, training only with patterns $\mathbf{x}_k$ of classes i and j (y_k=1 for $\mathbf{x}$ of class i, y=-1 for $\mathbf{x}$ of class j)

$$\mathbf{x} \longrightarrow \boxed{\begin{array}{c} ij\text{-th binary} \\ \text{SVM} \end{array}} \longrightarrow z_{ij}(\mathbf{x}) \in \{\pm 1\}$$

Multi-class classification (IV)



$$z(\boldsymbol{x}) = \underset{i=1\ldots N}{argmax}\{v_i(\boldsymbol{x})\}$$

$$v_i(\boldsymbol{x}) = \sum_{j=1}^{C}\sum_{k=1}^{C}\delta_{ij}\,\delta_{ik}\,z_{jk}(\boldsymbol{x}),\ i=1\ldots C$$

$\delta_{ij}=1$ if $i=j$ and $\delta_{ij}=0$ if $i\neq j$

Complexity

- The SVM training is a quadratic optimization with complexity of $O(N^3)$ and memory requirements of $O(N^2)$
- Efficient implementations: $O(N^p)$ with $1 \leq p \leq 2.3$
- SVM is normally slow for > 10.000-50.000 patterns, depending of the number n of inputs
- With very wide patterns (n high), use linear kernel because it is not necessary to map the data to a high-dimensional space.

$$z(\mathbf{x}) = sign(\mathbf{w}^T \mathbf{x} + b), \mathbf{w} = \sum_{i=1}^{M} \alpha_i y_{s(i)} \mathbf{x}_{s(i)}$$

- In this case, linear and Gaussian kernels achieve similar results.

Implementations

- **LibSVM**: accessible from C++, Octave/Matlab, Python, Weka/Java.
- Function **SVC** in package **scikit-learn** of Python.
- Function **ksvm** in the package **kernlab** of R.

LibSVM in Octave/Matlab

- Functions **svmtrain()** and **svmpredict()**

Examples of options: -s 0 -c 10 -t 1 -g 1 -r 1 -d 3
Classify a binary data with polynomial kernel (u'v+1)^3 and C = 10

```
options:
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC
        1 -- nu-SVC
        2 -- one-class SVM
        3 -- epsilon-SVR
        4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
        0 -- linear: u'*v
        1 -- polynomial: (gamma*u'*v + coef0)^degree
        2 -- radial basis function: exp(-gamma*|u-v|^2)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates: whether to train a SVC or SVR model for probability estimates, 0 or 1 (default 0)
-wi weight: set the parameter C of class i to weight*C, for C-SVC (default 1)

The k in the -g option means the number of attributes in the input data.
```

svm module in Python **scikit-learn**

- **https://scikit-learn.org/stable/modules/svm.html**

As other classifiers, `SVC`, `NuSVC` and `LinearSVC` take as input two arrays: an array `X` of shape `(n_samples, n_features)` holding the training samples, and an array `y` of class labels (strings or integers), of shape `(n_samples)`:

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, y)
SVC()
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
array([1])
```

Kernlab R package

- https://www.rdocumentation.org/packages/kernlab/versions/0.9-29/topics/ksvm

ksvm

From kernlab v0.9-29

Percentile

Support Vector Machines

Support Vector Machines are an excellent tool for classification, novelty detection, and regression. `ksvm` supports the well known C-svc, nu-svc, (classification) one-class-svc (novelty) eps-svr, nu-svr (regression) formulations along with native multi-class classification formulations and the bound-constraint SVM formulations. `ksvm` also supports class-probabilities output and confidence intervals for regression.

Keywords methods, regression, classif, nonlinear, neural

Usage

```
# S4 method for formula
ksvm(x, data = NULL, ..., subset, na.action = na.omit, scaled = TRUE)

# S4 method for vector
ksvm(x, ...)

# S4 method for matrix
```

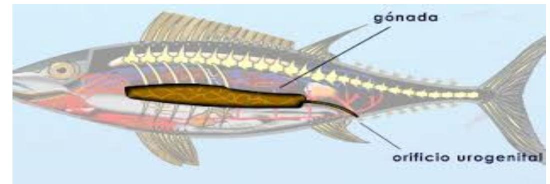
Real application: STERapp

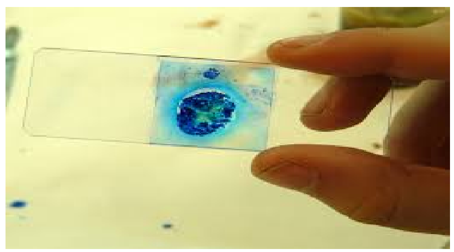
- https://citius.usc.es/transferencia/software/sterapp
- **STERapp** allows the estimation of fish fecundity by an automatic analysis of histological images of fish gonads.
- Specifically, cells are classified into three different development stages and also into cells with/without visible nucleus.
- It uses the Gaussian SVM classifier applied on texture and color features extracted from each cell.
- To calculate fecundity, we need to measure the cells with visible nucleus and to count the cells in each development stage.

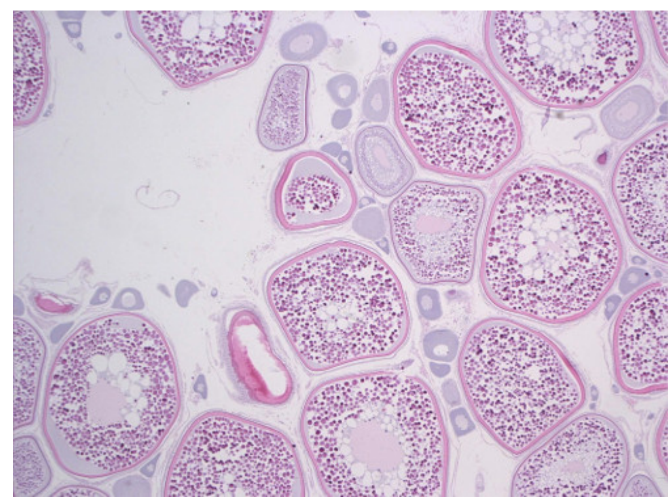
Real application: STERapp

- **Colaborators:**
  - **CiTIUS:** Centro Singular en Tecnoloxías intelixentes da USC.
  - **Universidade de Vigo**.
  - **IIM-CSIC:** Instituto de Investigaciones Marinas de Vigo.
  - **IEO-CSIC:** Instituto español de oceonografía.

Real application: STERapp

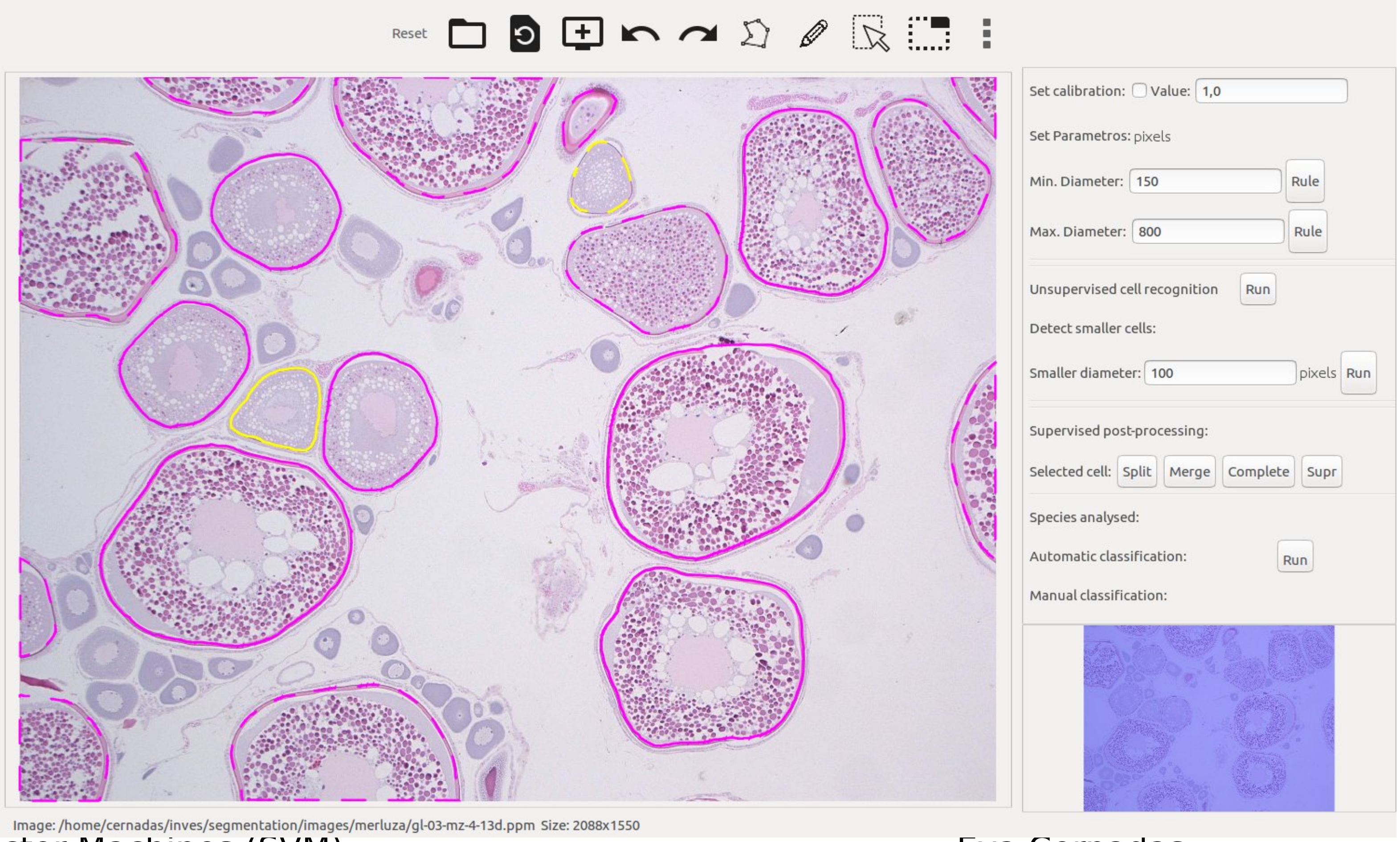






Real application: STERapp

- Three different development stages (colors) and present/absent nucleus (continuous/dashed line).

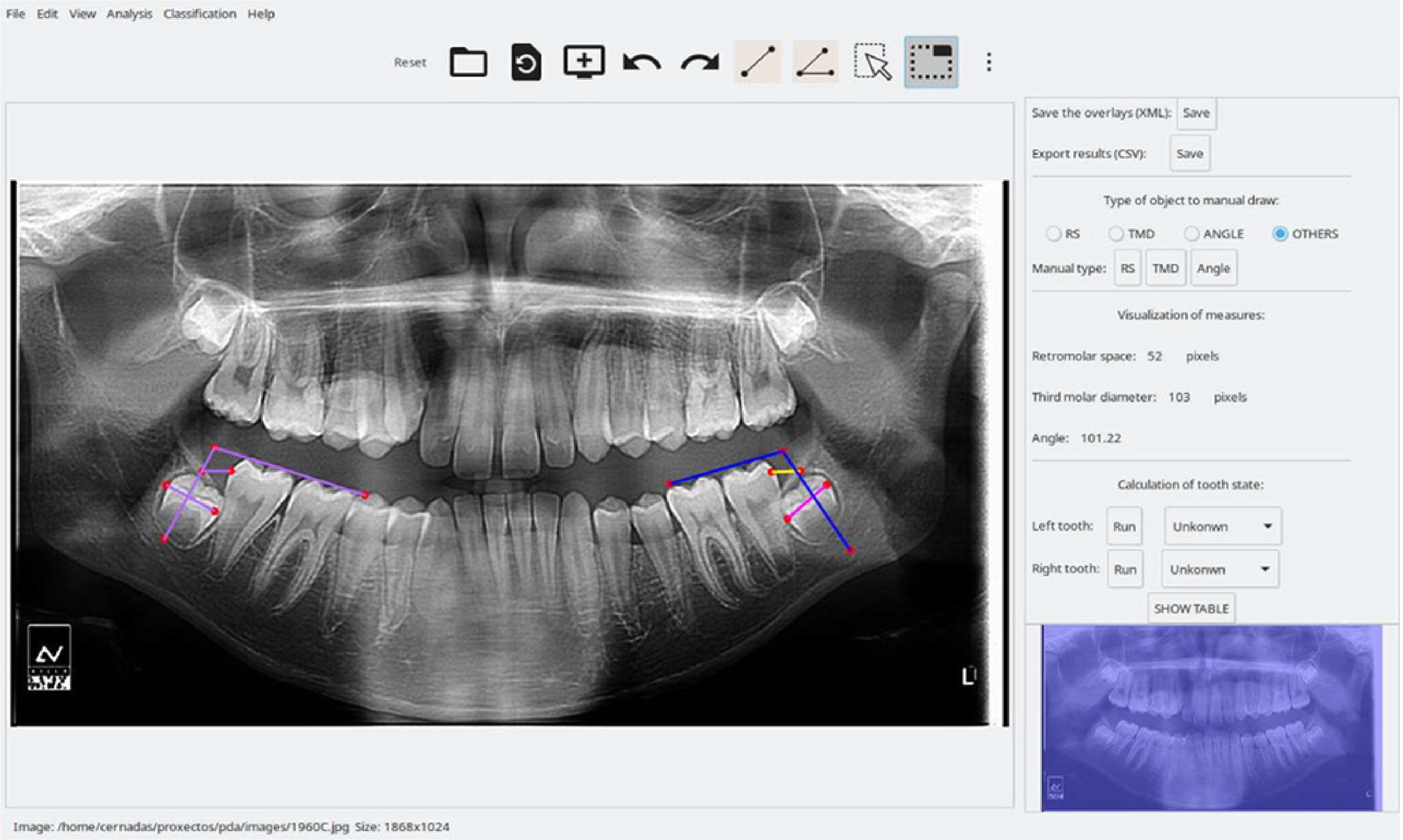


Real application: **PDApp**

- In colaboration with the **Faculty of Medicine and Dentistry** in the USC.
- **PDApp** is a new reliable and easy-to-use software tool to estimate the Third Molar Eruption Potential from the panoramic radiological images of adolescents/teenagers patients.
- Its GUI allows to draw the retromolar space, third molar diameter and angle on the image.
- Use a SVM to predict probability of positive (eruption) and negative (non-eruption) potential.

Real application: **PDApp**

- **https://citius.usc.es/transferencia/software/pdapp**



Fast Support Vector Classifier (FSVC)

- The SVC is unable to train with several thousands of patterns
- Calculation of $\{\alpha_i\}_{i \in SV}$ and b is of complexity $\Theta(N^3)$ and requires RAM memory $\Theta(N^2)$
- The whole training set must be stored in memory during training
- Testing requires to store all the support vectors
- Tuning of λ and RBF kernel spread σ requires to repeat training+test many times
- High n (many inputs): calculation of distance $|\mathbf{x}_n - \mathbf{x}_m|$ for kernel is slow
- High C (many classes): it requires to train $\Theta(C^2)$ binary SVCs

FSVC (II)

- We proposed **Fast SVC**: Fast Support Vector Classification for Large-Scale Problems, Z. Akram-Ali-Hammouri, M. Fernández-Delgado, E. Cernadas, S. Barro, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(10), 2022, DOI: 10.1109/TPAMI.2021.3085969
- Five elements that provide efficiency to SVC training+test:

1) **Efficient training**: no iterative optimization to calculate $\{\alpha_i\}$ and *b*. Instead, direct calculation of *b* and *y*(**x**) without training set storage

$$\boxed{y(\boldsymbol{x}) = sign\left(\sum_{n=2} \frac{k_n(\boldsymbol{x})}{N_2} - \sum_{n=1} \frac{k_n(\boldsymbol{x})}{N_1} + b\right)} \qquad k_n(\boldsymbol{x}) = K(\boldsymbol{x}_n, \boldsymbol{x})$$

$$K(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(\frac{-\left|\boldsymbol{x} - \boldsymbol{y}^2\right|}{2\sigma^2}\right) \qquad b = \sum_{nm=1} \frac{k_{nm}}{2N_1^2} - \sum_{nm=2} \frac{k_{nm}}{2N_2^2} \qquad k_{nm} = K(\boldsymbol{x}_n, \boldsymbol{x}_m)$$

FSVC (III)

2)Efficient kernel calculation: prototypes $\mathbf{p}_{ql}$ of classes created using *on-line kmeans clustering*:

$$\boldsymbol{p}_{ql}(t+1)=\left(1-\frac{1}{N_{qr}}\right)\boldsymbol{p}_{ql}(t)+\frac{\boldsymbol{x}_n}{N_{ql}} \quad q=c_n \quad r=\underset{l=1,\ldots L_q}{argmin}\left|\boldsymbol{p}_{ql}-\boldsymbol{x}_n\right| \quad N_{ql}=N_{ql}+1$$

- The previous equations are re-formulated for prototypes $\mathbf{p}_{ql}$ instead of training patterns $\mathbf{x}_n$:

$$\boxed{y(\boldsymbol{x})=sign\left(\sum_{l=1}^{L_2}\frac{k_{2l}(\boldsymbol{x})}{L_2}-\sum_{l=1}^{L_1}\frac{k_{1l}(\boldsymbol{x})}{L_1}+b\right)} \qquad k_{ql}(\boldsymbol{x})=K(\boldsymbol{p}_{ql},\boldsymbol{x})$$

$$K(\boldsymbol{x},\boldsymbol{y})=\exp\left(\frac{-\left|\boldsymbol{x}-\boldsymbol{y}^2\right|}{2\sigma^2}\right) \quad b=\sum_{lm=1}^{L_1}\frac{k_{1lm}}{2L_1^2}-\sum_{lm=1}^{L_2}\frac{k_{2lm}}{2L_2^2} \quad k_{qlm}=K(\boldsymbol{p}_{ql},\boldsymbol{x}_m)$$

FSVC (IV)

3)Efficient hyper-parameter tuning:

- Efficient training removes λ hyper-parameter
- Spread σ of RBF kernel estimated minimizing difference between kernel matrix **K** and **ideal** kernel matrix **J**
- $K^{(\sigma)}_{lm}=K(\mathbf{p}_l,\mathbf{p}_m,\sigma)$: RBF kernel for $\mathbf{p}_l$ and $\mathbf{p}_m$ with spread σ
- $J_{lm}=1$ when $c_l=c_m$ and $J_{lm}=0$ otherwise
- Difference $A(\sigma)$ between $\mathbf{K}^{(\sigma)}$ and **J**: $A(\sigma)=\sum_{lm=1}^{L_1+L_2}\frac{|K_{lm}^{(\sigma)}-J_{lm}|}{(L_1+L_2)^2}$
- Select σ_0 as: $\sigma_0=\underset{\sigma\in\Sigma}{argmin}\{A(\sigma)\}$ $\quad\Sigma=\{2^{\frac{-(i+1)}{2}}\}_{i=-13}^{13}$
- Avoids repetition of training+test

FSVC (V)

4)Large input dimensionality n: use of linear instead of RBF kernel: $y(\mathbf{x})=\text{sign}(\mathbf{w}^T\mathbf{x}+b)$, with **w** and b:

$$\boldsymbol{w}=\sum_{l=1}^{L_2}\frac{\boldsymbol{p}_{2l}}{L_2}-\sum_{l=1}^{L_1}\frac{\boldsymbol{p}_{1l}}{L_1} \qquad b=\sum_{lm=1}^{L_1}\frac{\boldsymbol{p}_{1l}^T\boldsymbol{p}_{1m}}{2L_1^2}-\sum_{lm=1}^{L_2}\frac{\boldsymbol{p}_{2l}^T\boldsymbol{p}_{2m}}{2L_2^2}$$

Very efficient: n-dimensional dot product and sum

5)Large number C **of classes**: use of *one-vs-all* instead of *one-vs-one*

- **Computational complexity of FSVC**: linear in N (no. training patterns), n (no. inputs) and T (no. test patterns), quadratic only in C (no. classes)
- **Low memory required**: tunable depending on the available memory; less memory → less speed

FSVC (VI)

- Implementation in CodeOcean: DOI:

  https://doi.org/10.24433/CO.8733864.v1

- Code also available from this **link**
- Executed on datasets up to N=31 millions of patterns, n=30.000 inputs and C=131 classes
- Average performance 6% below SVC on small datasets
- The slowest dataset: 21 millions patterns, 115 inputs, 9 classes. FSVC spent 1h 40m per fold (4-fold cross validation)
- Can be run in low-power computers (small memory)
- Faster and more accurate than Pegasos-SVM, SVM-SIMBA and Indefinite Core Vector Machine. Faster than evolutionary training set selection, that is unable to run on most datasets