Aprendizagem Automática

Aula Prática

Generalização de Modelos Lineares Aplicados à Classificação:

Discriminantes Logísticos
+
Máquinas de Suporte Vetorial

Ajuste de Parâmetros de Modelos

𝔾. Marques

Discriminantes Logísticos

Discriminantes Logísticos

- Modelo:
  Transformação linear seguida de uma não-linearidade (função de ativação): $\hat{\mathbf{y}} = \varphi\left(\mathbf{W}^\top \mathbf{x}\right)$

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_c \end{bmatrix} = \begin{bmatrix} \varphi(u_1) \\ \varphi(u_2) \\ \vdots \\ \varphi(u_c) \end{bmatrix} = \varphi\left(\underbrace{\begin{bmatrix} w_{01} & w_{11} & w_{21} & \cdots & w_{d1} \\ w_{02} & w_{12} & w_{22} & \cdots & w_{d2} \\ \vdots & & & & \vdots \\ w_{0c} & w_{1c} & w_{2c} & \cdots & w_{dc} \end{bmatrix}}_{\mathbf{W}^\top} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \right)$$

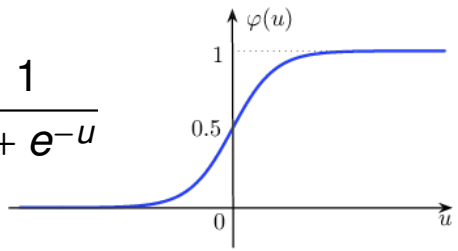
com $u_k = \mathbf{w}_k^\top \mathbf{x} = w_{0k} + w_{1k} x_1 + \ldots + w_{dk} x_d, \quad k = 1, \ldots, c$, em que $\mathbf{w}_k$ é a coluna k da matriz $\mathbf{W}$ ($\mathbf{u}$ vetor de $c \times 1$: $\mathbf{u} = \mathbf{W}^\top \mathbf{x}$).

ATENÇÃO: vetor de dados $\mathbf{x}$ em coordenadas homogéneas.

- Função de ativação $\varphi()$ é uma de duas funções:

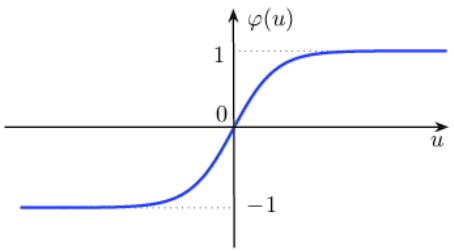
sigmóide

$$\varphi(u) = \frac{1}{1 + e^{-u}}$$



ou tangente hiperbólica

$$\varphi(u) = \tanh(u)$$



Discriminantes Logísticos

- Problema de aprendizagem supervisionada:
  - O objetivo é estimar a matriz de peso **W** baseada num conjunto de N vetores **y** e num conjunto com os correspondentes N vetores **x**.
  - Os vetores **y** são as saídas desejadas, e indicam a classe do vetor **x**. Os vetores **y** são de dimensão $c \times 1$. O valor "+1" na linha k do vetor **y** indica que $\mathbf{x} \in \varpi_k$. Os restantes valores do vetor são "0" ou "−1", dependendo da função de ativação escolhida:

$$\text{para } \mathbf{x} \in \varpi_k, \quad \mathbf{y} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ +1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \longleftarrow \text{linha } k \longrightarrow \begin{bmatrix} -1 \\ \vdots \\ -1 \\ +1 \\ -1 \\ \vdots \\ -1 \end{bmatrix} = \mathbf{y}$$

sigmóide tangente hiperbólica

Discriminantes Logísticos

- Problema de aprendizagem supervisionada:
  - O objetivo é estimar a matriz de peso **W** baseada num conjunto de N vetores **y** e num conjunto com os correspondentes N vetores **x**.
- Estimação dos parâmetros do modelo:
  Os valor da matriz de pesos, **W**, é estimado através da minimização adaptativa do erro quadrático médio.
  - $\mathcal{E}(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\left\|\mathbf{y}[n] - \varphi\left(\mathbf{W}^{\top}\mathbf{x}[n]\right)\right\|^2 = \frac{1}{N}\sum_{n=1}^{N}\left\|\mathbf{y}[n] - \hat{\mathbf{y}}[n]\right\|^2$
  - Adaptação da matriz **W**, feita de modo iterativo. Para a iteração $i+1$, o valor da matriz **W** depende do seu valor prévio (iteração i) menos um termo de gradiente, na direção que minimiza a função do erro.

    $$\mathbf{W}_{i+1} = \mathbf{W}_i - \eta\frac{\partial\mathcal{E}(\mathbf{W}_i)}{\partial\mathbf{W}}$$
  - Existem várias técnicas de otimização que podem ser usadas para estimar os pesos (por exemplo, o método de Newton).
  - A implementação pode ser feita de modo determinístico (*batch*) ou estocástico (*on-line*).

Discriminantes Logísticos

- Problema de aprendizagem supervisionada:
  - O objetivo é estimar a matriz de peso **W** baseada num conjunto de N vetores **y** e num conjunto com os correspondentes N vetores **x**.
- Regularização:

  Pode-se acrescentar à função do erro, um termo de regularização que penaliza coeficientes da matriz **W** com valores elevados (em termos absolutos).
  - $\mathcal{E}(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\left\|\mathbf{y}[n] - \hat{\mathbf{y}}[n]\right\|^2 + \lambda \sum_{w_{ij}\in\mathbf{W}} |w_{ij}|^{\rho}$
  - λ controla a importância do termo de regularização.
  - Para $\rho = 2$ é uma regularização quadrática (ℓ_2 ou *ridge*)
  - Para $\rho = 1$ é uma regularização ℓ_1 ou *lasso*. Esta regularização tende a pôr a zero vários coeficientes da matriz **W**, o que pode ser importante para descartar dimensões supérfluas dos dados.

Discriminantes Logísticos - `sklearn`

- `LogisticRegression`
  Apesar do nome, este método é um algoritmo de classificação e não de regressão.

```
>>> from sklearn.linear_model import LogisticRegression
>>> Dlog=LogisticRegression().fit(Xtrain,ytrain)
>>> print(Dlog.score(Xtest,ytest))
```

- Vários parâmetros a ter em conta.
  - `tol`: 10^{-4} (`default`). Critério de paragem.
  - `max_iter`: 100 (`default`). Número máximo de iterações.
  - `solver`: `lbgs` (`default`). Método de otimização.
    `liblinear` limitado a classificação binária.
    Para multi-classe, usar `newton-cg, lbfgs, sag, saga`.
  - `multi_class`: `auto` (`default`). Classificação binária ou multi-classe.
    `ovr` One Versus the Rest - problema binário.
    `multinolmial` problema multi-classe.
    `auto` ajusta-se automaticamente.

Discriminantes Logísticos - `sklearn`

- `LogisticRegression`
  Apesar do nome, este método é um algoritmo de classificação e não de regressão.

```
>>> from sklearn.linear_model import LogisticRegression
>>> Dlog=LogisticRegression().fit(Xtrain,ytrain)
>>> print(Dlog.score(Xtest,ytest))
```

- Regularização:

  Por omissão, o discriminante logístico do `sklearn` usa regularização *ridge* ou ℓ_2. O peso dado à regularização é controlado pelo parâmetro `C` e o tipo de regularização, *lasso* ou *ridge*, pelo parâmetro `penalty`.

  - `C`: 1.0 (`default`). Inverso do peso dado ao termo de regularização. Valores pequenos correspondem a uma regularização "forte".
  - `penalty`: `l2` (`default`). Método de regularização.
    lasso: `l1`.
    ridge: `l2`.
  - As otimizações de `newton-cg`, `lbfgs` e `sag` só suportam uma regularização ℓ_2. `liblinear` e `saga` suportam as duas.

Discriminantes Logísticos - `sklearn`

- `LogisticRegression`
  Apesar do nome, este método é um algoritmo de classificação e não de regressão.

```
>>> from sklearn.linear_model import LogisticRegression
>>> Dlog=LogisticRegression().fit(Xtrain,ytrain)
>>> print(Dlog.score(Xtest,ytest))
```

- Atributos de saída.

  Após o treino (chamar a função `.fit()`), pode-se aceder à matriz **W** estimada bem como o número de iteração executadas durante a adaptação.

  - `Dlog.coef_`: coeficientes da matriz **W**, exceto a 1ª linha
  - `Dlog.intercept_`: 1ª linha da matriz **W** (coeficientes $w_{0i},\ i = 1, \ldots, c$)
  - `Dlog.n_iter_`: número de iterações executadas

  ATENÇÃO: Quando só há duas classes, os peso ficam guardados num vetor.

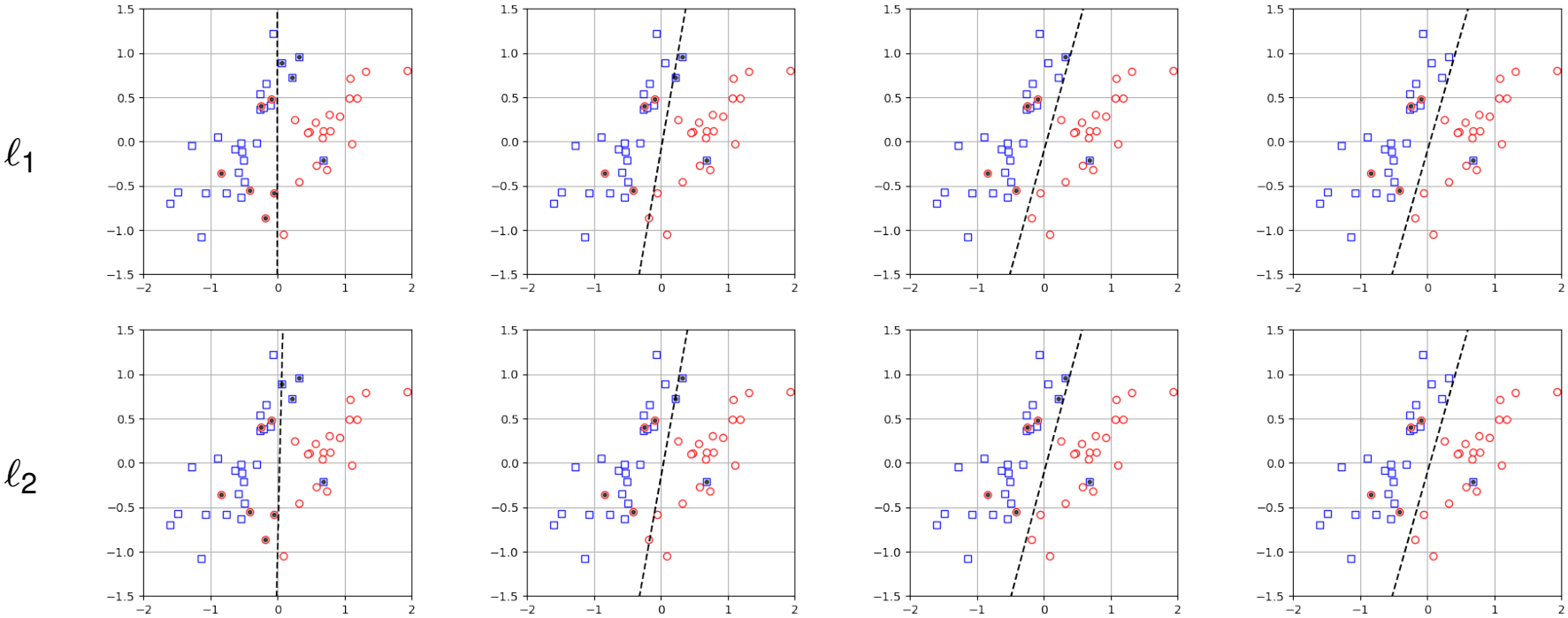
Discriminantes Logísticos - `sklearn`

Exemplo: Classificação binária

DADOS: `binClassData.p`

Pontos bidimensionais, pertencentes a duas classes, divididos em treino e teste.

As seguintes figuras mostram o resultado obtido com um discriminante logístico, com regularização ℓ_1 e ℓ_2, e com diferentes valores para o parâmetro `C`.



Discriminantes Logísticos - sklearn

Exemplo: Classificação binária - Breast Cancer dataset

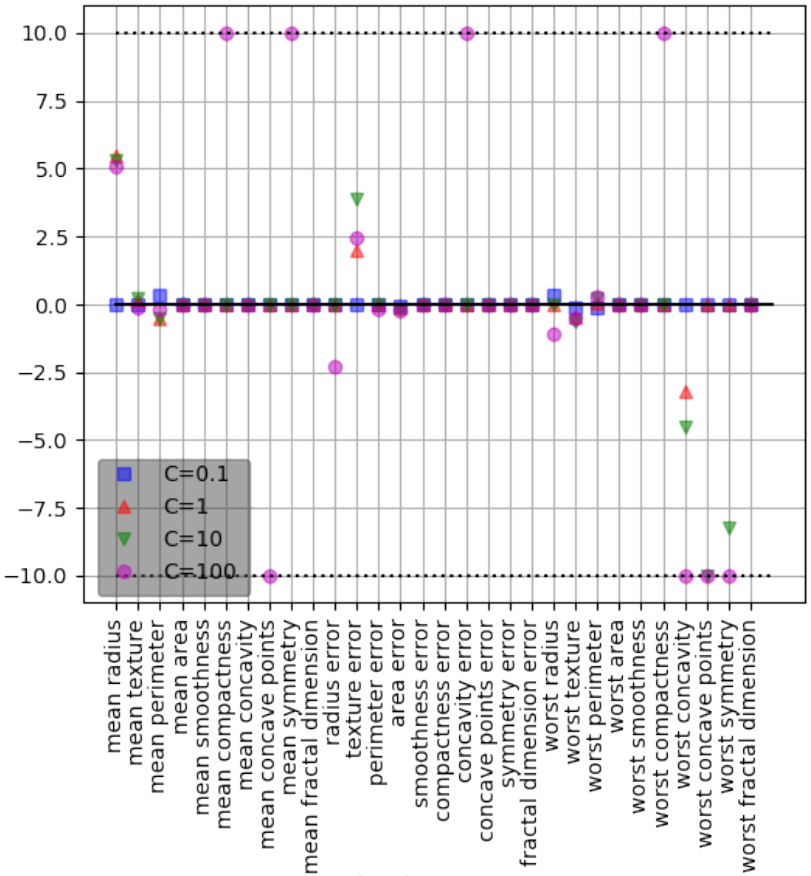
Pontos a 30 dimensões, pertencentes a duas classes. O conjunto foi dividido em treino e teste. De notar que neste caso os pesos são um vetor de 30 dimensões, e cada coeficiente corresponde a uma dimensão dos dados.

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.model_selection import train_test_split
>>> BC=load_breast_cancer()
>>> Xtrain,Xtest,ytrain,ytest=train_test_split(BC.data,BC.target)
>>> Dlog=LogisticRegression.fit(Xtrain,ytrain)
>>> print(Dlog.score(Xtest,ytest))
>>> print(Dlog.coef_)
```

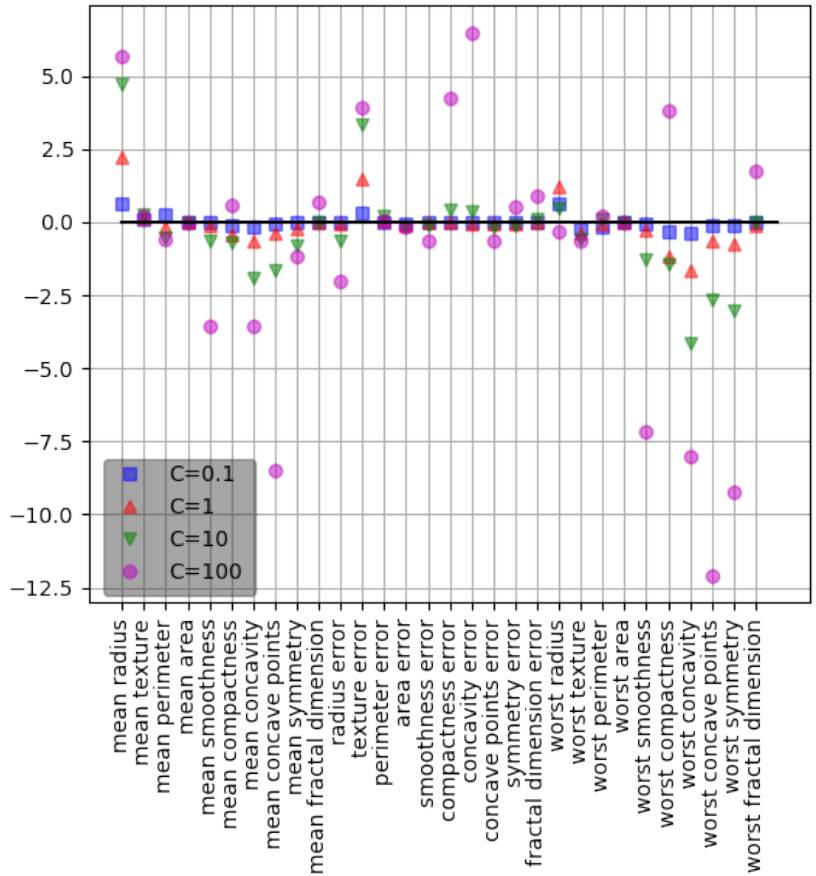
Discriminantes Logísticos - `sklearn`

Exemplo: Classificação binária - Breast Cancer dataset

As seguintes figuras mostram o valor dos pesos obtidos com regularização ℓ_1 e ℓ_2, e com diferentes valores para `C`. Para a regularização ℓ_2, vários coeficientes têm valores pequenos mas raramente chegam a zero. Já no caso da regularização ℓ_1, vários coeficientes são zero, o que pode ser benéfico para descartar dimensões supérfluas dos dados e aferir as características discriminantes.



ℓ_1 ($|w| \leq 10$)



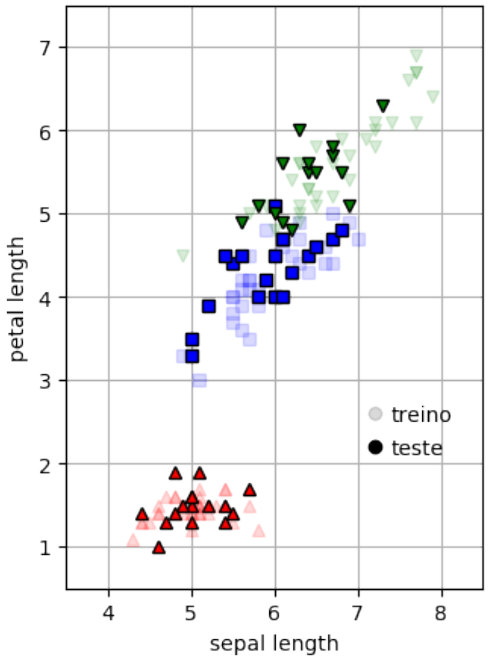
ℓ_2

Discriminantes Logísticos - `sklearn`

Exemplo: Classificação multi-classe - Iris dataset

Pontos a 4 dimensões, pertencentes a três classes. Necessário dividir em treino e teste.

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> Iris=load_iris()
>>> Xtr,Xte,ytr,yte=train_test_split(Iris.data,Iris.target,test_size=1./3,random_state=0)
```



Atenção: O método de otimização `liblinear`, não funciona em modo multi-classe. De notar que o modo binário obtém resultados diferentes do modo multi-classe.

Discriminantes Logísticos - `sklearn`

Exemplo: Classificação multi-classe - Iris dataset

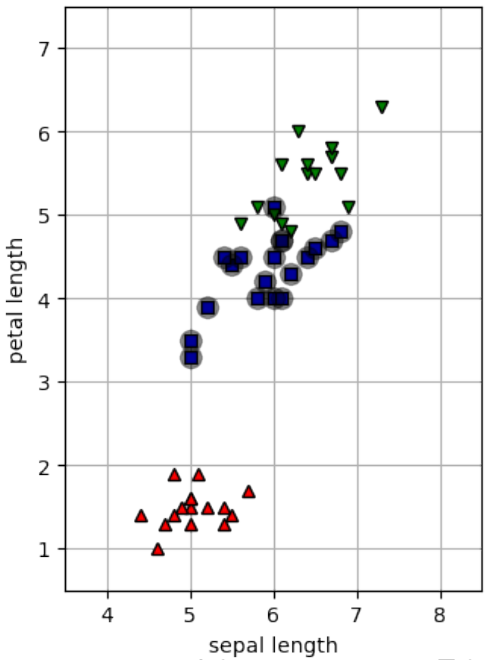
As seguintes figuras mostram o resultado obtido com um discriminante logístico, com regularização ℓ_1 e ℓ_2, com otimização, `liblinear` (classificação binária, um contra o resto), e com `C`= 0.1.

```
>>> Dlog=LogisticRegression(solver='liblinear',C=0.1,penalty='l1')
```

ou

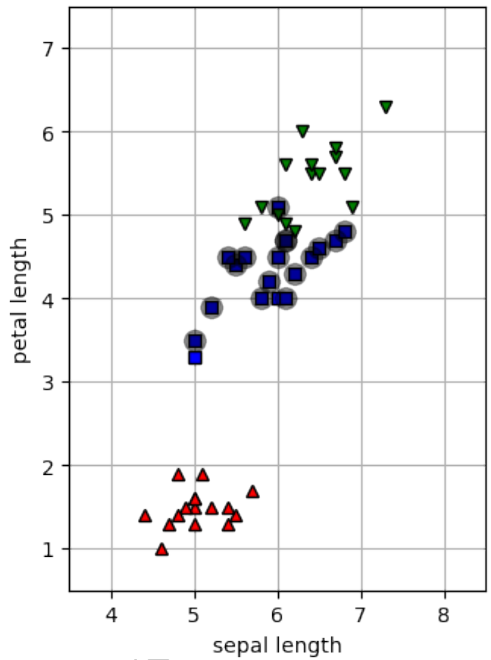
```
>>> Dlog=LogisticRegression.(solver='liblinear',C=0.1,penalty='l2')
```

ℓ_1



19 erros em 50

ℓ_2



17 erros

`Dlog.coef_=`

$$\begin{bmatrix} 0 & 0.88 & -1.09 & 0 \\ 0 & 0.25 & 0 & 0 \\ -0.73 & 0 & 0.97 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0.19 & 0.66 & -1.08 & -0.5 \\ -0.02 & -0.5 & 0.23 & -0.14 \\ -0.53 & -0.48 & 0.83 & 0.62 \end{bmatrix}$$

Discriminantes Logísticos - `sklearn`

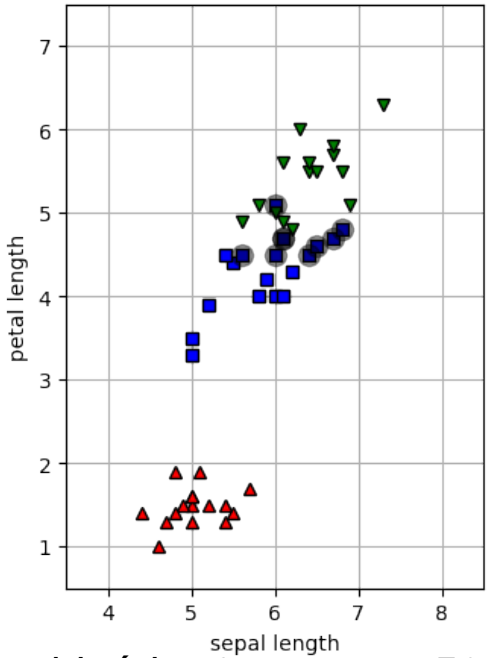
Exemplo: Classificação multi-classe - Iris dataset

Comparação de resultados obtidos com discriminantes logísticos em modo binário e multi-classe. Comparação feita usando o mesmo método de otimização e regularização. Otimização `newton-cg`, com regularização ℓ_2, e com `C`= 0.1.

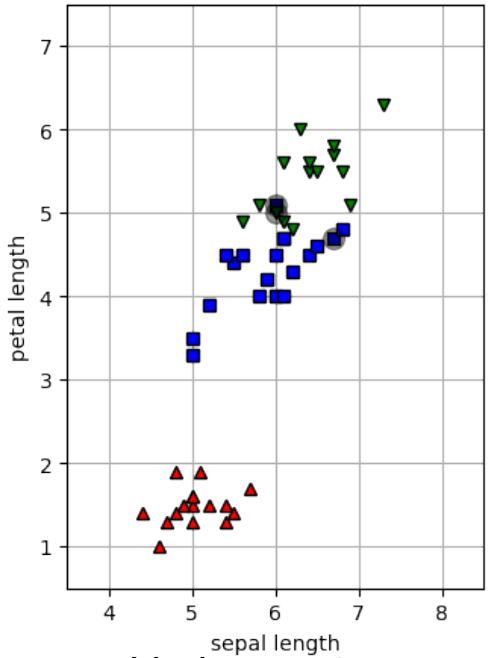
```
>>> Dlog=LogisticRegression(solver='newton-cg',C=0.1,penalty='l2',multi_class='ovr')
```

versus

```
>>> Dlog=LogisticRegression(solver='newton-cg',C=0.1,penalty='l2',multi_class='mulinomial')
```



binário: 9 erros em 50



multi-classe: 3 erros

$$\texttt{Dlog.coef\_}=\begin{bmatrix} -0.32 & 0.29 & -1.14 & -0.47 \\ -0.13 & -0.58 & 0.24 & -0.12 \\ 0.25 & 0.04 & 0.99 & 0.61 \end{bmatrix} \qquad \begin{bmatrix} -0.27 & 0.26 & -1.0 & -0.41 \\ 0.05 & -0.29 & 0.09 & -0.15 \\ 0.23 & 0.03 & 0.91 & 0.56 \end{bmatrix}$$

Discriminantes Logísticos - `sklearn`

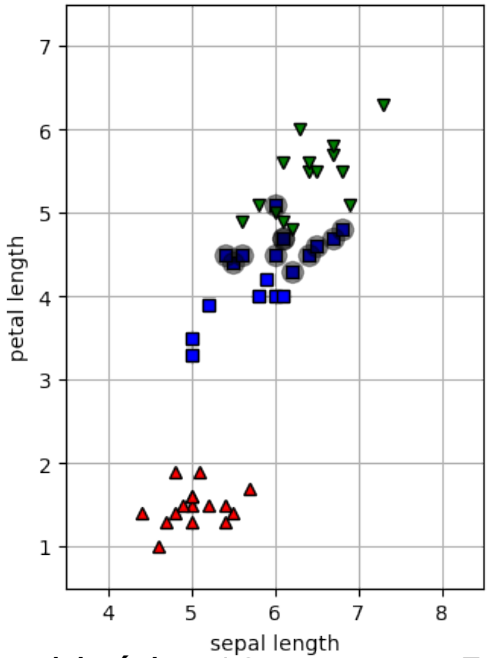
Exemplo: Classificação multi-classe - Iris dataset

Comparação de resultados obtidos com discriminantes logísticos em modo binário e multi-classe. Comparação feita usando o mesmo método de otimização e regularização. Otimização `saga`, com regularização ℓ_1, e com `C`= 0.1.

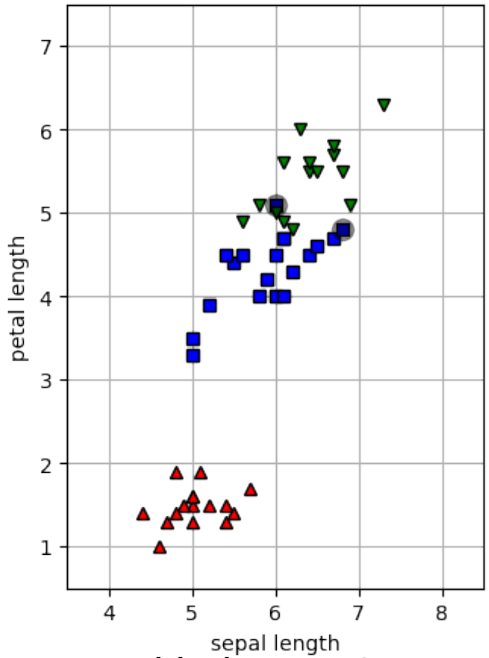
```
>>> Dlog=LogisticRegression(solver='saga',C=0.1,penalty='l1',multi_class='ovr')
```

versus

```
>>> Dlog=LogisticRegression(solver='saga',C=0.1,penalty='l1',multi_class='mulinomial')
```



binário: 12 erros em 50



multi-classe: 2 erros

$$\texttt{Dlog.coef\_}= \begin{bmatrix} 0 & 0 & -1.59 & 0 \\ 0 & 0 & 0.06 & 0 \\ 0 & 0 & 1.40 & 0 \end{bmatrix}$$

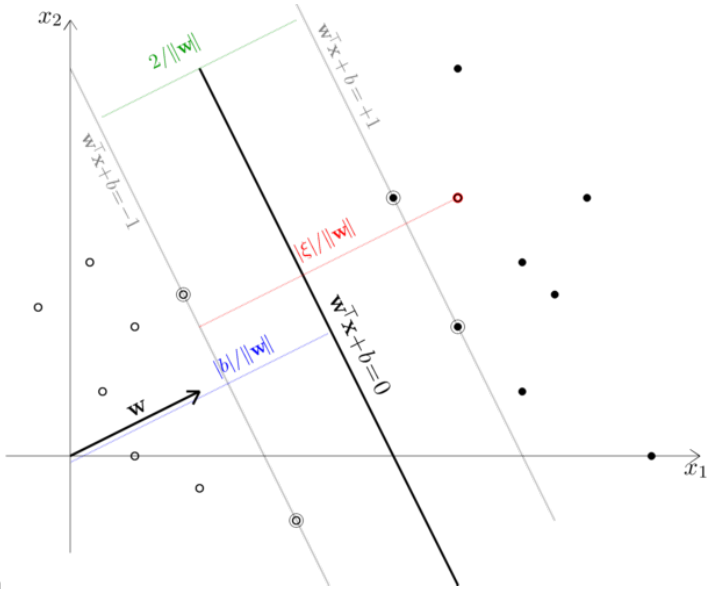
$$\begin{bmatrix} 0 & 0 & -1.40 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1.19 & 0 \end{bmatrix}$$

Máquinas de Suporte Vetorial (SVMs)

Máquinas de Suporte Vetorial

Máquinas de suporte vetorial (SVM do inglês Support Vector Machine), são algoritmos supervisionados de aprendizagem automática aplicáveis a problemas de classificação e regressão. Foi inicialmente desenvolvido para problemas de classificação binária.

- Objetivos:
  - Maximizar a margem de separação entre as duas classes.
  - Os pontos mais perto da margem de separação são os vetores de suporte (pontos com círculos). Estes pontos correspondem aos exemplos mais difíceis de classificar.
  - Os vetores de suporte definem a margem de separação. Mais pontos de treino fora da margem não afetam a classificação.



- Vantagens:
  - Estima o hiperplano ótimo de separação.
  - Lida bem com dados de alta dimensão.
  - Lida com problemas complexos
- Desvantagens:
  - Necessário escolher apropriadamente a função de *kernel*.
  - Não lida bem com quantidades elevadas de dados.

Máquinas de Suporte Vetorial

Máquinas de suporte vetorial (SVM do inglês Support Vector Machine), são algoritmos supervisionados de aprendizagem automática aplicáveis a problemas de classificação e regressão. Foi inicialmente desenvolvido para problemas de classificação binária.

- Funcional a minimizar:
  Para um conjunto de N vetores $\mathbf{x} \in \mathbb{R}^d$ pertencentes a duas classe $y \in \{-1, +1\}$, minimizar:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

  sujeito a: $y_i\left(\mathbf{w}^\top\mathbf{x}_i + b\right) \geq 1-\xi_i, \quad$ com $\xi_i \geq 0 \ \forall i$

  ξ_i: variáveis associadas a cada $\mathbf{x}_i$, que permitem que estes pontos estejam do lado errado da margem (*soft margin classification*: permite lidar com dados que não são linearmente separáveis).

  C: inverso do peso dado ao termo de regularização. Valores elevados de C resultam em menos erros mas também numa menor margem de separação. C controla o compromisso entre ter uma margem de separação tão larga quanto possível e limitar o número erros.

Máquinas de Suporte Vetorial

Máquinas de suporte vetorial (SVM do inglês Support Vector Machine), são algoritmos supervisionados de aprendizagem automática aplicáveis a problemas de classificação e regressão. Foi inicialmente desenvolvido para problemas de classificação binária.

- Separações não-lineares:
  Para conjuntos que não são linearmente separáveis, pode-se usar transformações não-lineares $\Phi(\mathbf{x}_i)$ dos vetores de dados $\mathbf{x}_i$. A função a minimizar passa a ser:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

  sujeito a: $y_i\left(\mathbf{w}^\top\Phi(\mathbf{x}_i) + b\right) \geq 1-\xi_i, \quad \text{com } \xi_i \geq 0 \ \forall i$

  - As funções Φ mapeiam os dados de entrada de um espaço $\mathbb{R}^d$ para um espaço de maior dimensão $\mathbb{R}^k$ com $k >> d$ (k pode ser infinito).
  - O treino dos SVMs passa a depender do produto interno: $\Phi(\mathbf{x}_i)^\top\Phi(\mathbf{x}_j)$.
  - A estimação do hiper-plano de máxima margem no espaço $\mathbb{R}^k$ resulta numa separação não linear no espaço dos dados.
  - Calcular explicitamente o produto interno $\Phi(\mathbf{x}_i)^\top\Phi(\mathbf{x}_j)$ pode não ser possível dado o custo computacional.

Máquinas de Suporte Vetorial

Máquinas de suporte vetorial (SVM do inglês Support Vector Machine), são algoritmos supervisionados de aprendizagem automática aplicáveis a problemas de classificação e regressão. Foi inicialmente desenvolvido para problemas de classificação binária.

- Separações não-lineares:
  Para conjuntos que não são linearmente separáveis, pode-se usar transformações não-lineares $\Phi(\mathbf{x}_i)$ dos vetores de dados $\mathbf{x}_i$. A função a minimizar passa a ser:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

  sujeito a: $y_i\left(\mathbf{w}^\top\Phi(\mathbf{x}_i) + b\right) \geq 1-\xi_i, \quad$ com $\xi_i \geq 0 \ \forall i$

- Adicionar características não-lineares tornas os modelos mais flexíveis e capazes de lidar com problemas mais complexos. No entanto, não se sabe à partida quais não-linearidades adicionar, e pode não ser possível fazer-lo devido ao custo computacional (por exemplo, adicionar todas as combinações de 3ª ordem num espaço de características de 100 dimensões).

- *Kernel trick*:
  Uma função de *kernel*, $K(\mathbf{x}_1, \mathbf{x}_2)$ é capaz de calcular o produto interno $\Phi(\mathbf{x}_1)^\top\Phi(\mathbf{x}_2)$, baseado somente nos vetores $\mathbf{x}_1$ e $\mathbf{x}_2$ - sem ter que calcular ou até saber a transformação $\Phi(\mathbf{x})$.

Máquinas de Suporte Vetorial

Máquinas de suporte vetorial (SVM do inglês Support Vector Machine), são algoritmos supervisionados de aprendizagem automática aplicáveis a problemas de classificação e regressão. Foi inicialmente desenvolvido para problemas de classificação binária.

- Separações não-lineares:
  Para conjuntos que não são linearmente separáveis, pode-se usar transformações não-lineares $\Phi(\mathbf{x}_i)$ dos vetores de dados $\mathbf{x}_i$. A função a minimizar passa a ser:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

  sujeito a: $y_i\left(\mathbf{w}^\top\Phi(\mathbf{x}_i) + b\right) \geq 1-\xi_i, \quad \text{com } \xi_i \geq 0 \ \forall i$

- Funções de *Kernel*: $K(\mathbf{x}_1,\mathbf{x}_2) = \Phi(\mathbf{x}_1)^\top\Phi(\mathbf{x}_2)$

  Existe um grande número de destas funções, em que as mais comuns são:

Linear: $K(\mathbf{x}_1,\mathbf{x}_2) = \mathbf{x}_1^\top\mathbf{x}_2 + \beta$

Polinomial: $K(\mathbf{x}_1,\mathbf{x}_2) = \left(\gamma\mathbf{x}_1^\top\mathbf{x}_2 + \beta\right)^p$

Gaussian RBF: $K(\mathbf{x}_1,\mathbf{x}_2) = \exp\left(-\gamma\|\mathbf{x}_1-\mathbf{x}_2\|^2\right)$

Sigmoidal: $K(\mathbf{x}_1,\mathbf{x}_2) = \tanh\left(\gamma\mathbf{x}_1^\top\mathbf{x}_2 + \beta\right)$

Máquinas de Suporte Vetorial - `sklearn`

Em `sklearn` existem várias implementações de máquinas de suporte vetorial para problemas de classificação e regressão.

- SVMs para classificação
  - `LinearSVC`:
    Classificador SVM com *kernel* linear
  - `SVC`:
    Classificador SVM genérico. Pode-se escolher vários tipos de funções de *kernel*.
  - `NuSVC`:
    SVM genérico modificado para a regularização ser controlada por um parâmetro ν (valores entre $[0, 1]$) em vez de `C` (valores entre $[0, +\infty[$).
  - `SGDClassifier`:
    SGD - Stochastic Gradient Descent - classificador (SVM ou discriminante logístico). Convergência mais lenta, mas o método é útil para lidar com grandes quantidades de dados.
- SVMs para regressão
  - `LinearSVR`:
  - `SVR`:
  - `NuSVR`:
  - `SGDRegressor`:

Máquinas de Suporte Vetorial - `sklearn`

- `LinearSVC`

```
>>> from sklearn.svm import LinearSVC
>>> svm=LinearSVC().fit(Xtrain,ytrain)
>>> print(svm.score(Xtest,ytest))
```

- Vários parâmetros a ter em conta.
  - `penalty`: ℓ_2 (`default`). Regularização ℓ_1 (*lasso*) ou ℓ_2 (*ridge*)
  - `C`: 1.0 (`default`). Inverso do peso dado ao termo de regularização.
  - `tol`: 10^{-4} (`default`). Critério de paragem.
  - `max_iter`: 1000 (`default`). Número máximo de iterações.
  - `multi_class`: `ovr` (`default`). Classificação binária ou multi-classe.
    `ovr` problema binário - um contra o resto.
    `crammer_singer` problema multi-classe.
  - `loss`: `squared_hinge` (`default`).
    Função de custo que limita o número de erros.
    Pôr parâmetro `loss=hinge` (formulação original dos SVMs)
  - `dual`: `True` (`default`). Pôr `dual=False` para $N > d$.
  - `class_weight`: `None` (`default`).
    Quando "`balanced`" o parâmetro `C` é pesado pela probabilidade a priori de cada classe.

Máquinas de Suporte Vetorial - `sklearn`

- SVC

```
>>> from sklearn.svm import SVC
>>> svm=SVC().fit(Xtrain,ytrain)
>>> print(svm.score(Xtest,ytest))
```

- Vários parâmetros a ter em conta
  - `C`: 1.0 (`default`). Inverso do peso dado ao termo de regularização.
  - `tol`: 10^{-3} (`default`). Critério de paragem.
  - `max_iter`: -1 (`default`). Número máximo de iterações.
  - `class_weight`: `None` (`default`).
    Quando "`balanced`" o parâmetro `C` é pesado pela probabilidade a priori de cada classe.
  - `decision_function_shape`: `ovr` (`default`). Só classificação binária.
    `ovr` - *one versus the rest*. `ovo` - *one versus one*
  - `kernel`: `rbf` (`default`).
    Função de *kernel*: `linear`, `poly`, `rbf`, e `sigmiod`.
  - `gamma`: `auto` (`default`).
    Parâmetro γ para os *kernels* `poly`, `rbf`, e `sigmiod`.
    `auto` $\Longrightarrow \gamma = \frac{1}{N}$, `scale` $\Longrightarrow \gamma = \frac{1}{N\sigma_x}$
  - `degree`: 3 (`default`). Grau do polinómio (só para `kernel=poly`).

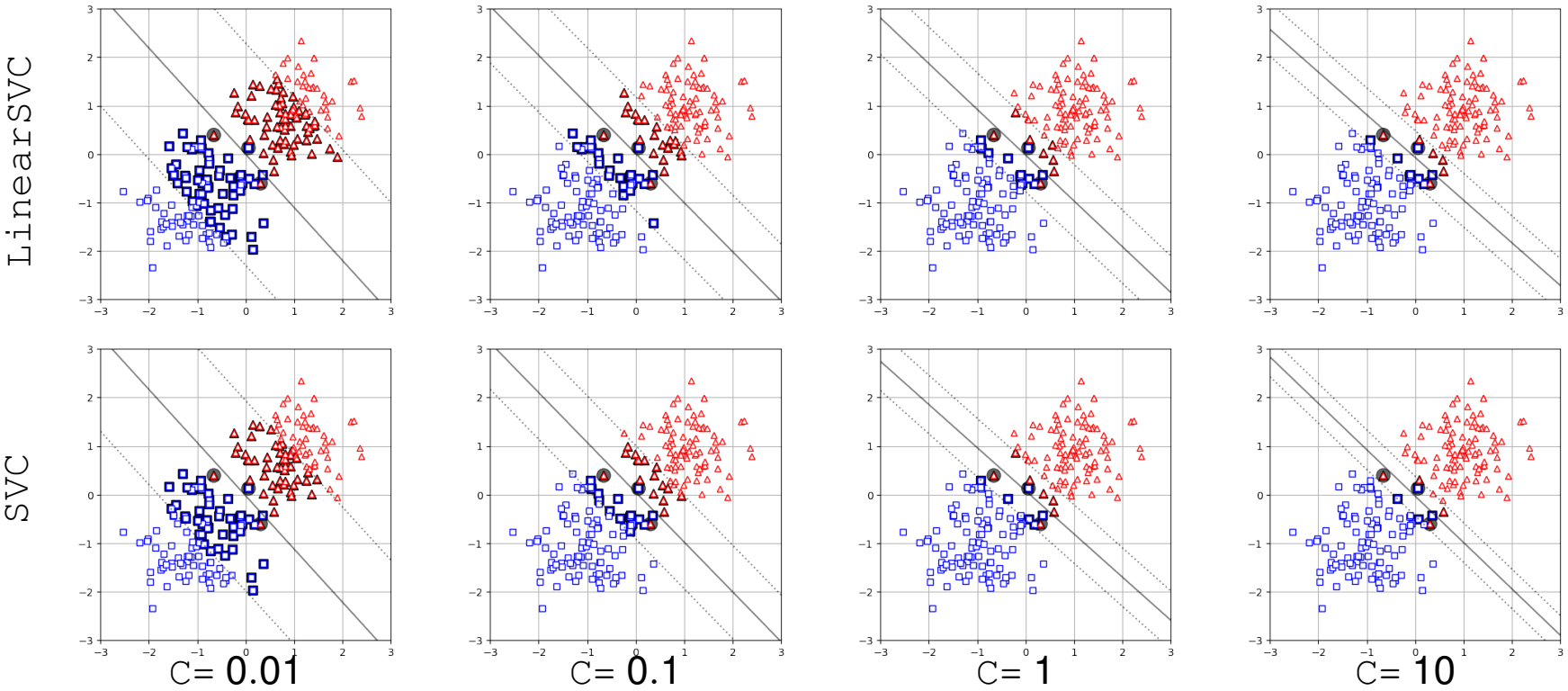
Máquinas de Suporte Vetorial - `sklearn`

Exemplo: SVMs lineares - Classificação binária

Pontos bidimensionais, pertencentes a duas classes, com:

$p(\mathbf{x}|\varpi_1) = \mathcal{N}(\mu_1, \mathbf{I})$ e $p(\mathbf{x}|\varpi_2) = \mathcal{N}(\mu_2, \mathbf{I})$, com $\mu_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$ e $\mu_2 = \begin{bmatrix} +1 \\ +1 \end{bmatrix}$, e com $p(\varpi_1) = p(\varpi_2)$.

As seguintes figuras mostram os resultados obtidos com SVMs lineares (`LinearSVC` ou `SVC(kernel='linear')`), e com diferentes valores para o parâmetro `C`.



- A função `LinearSVC` faz regularização do peso w_0, e é aconselhável subtrair a média aos dados antes de treinar os SVMs.

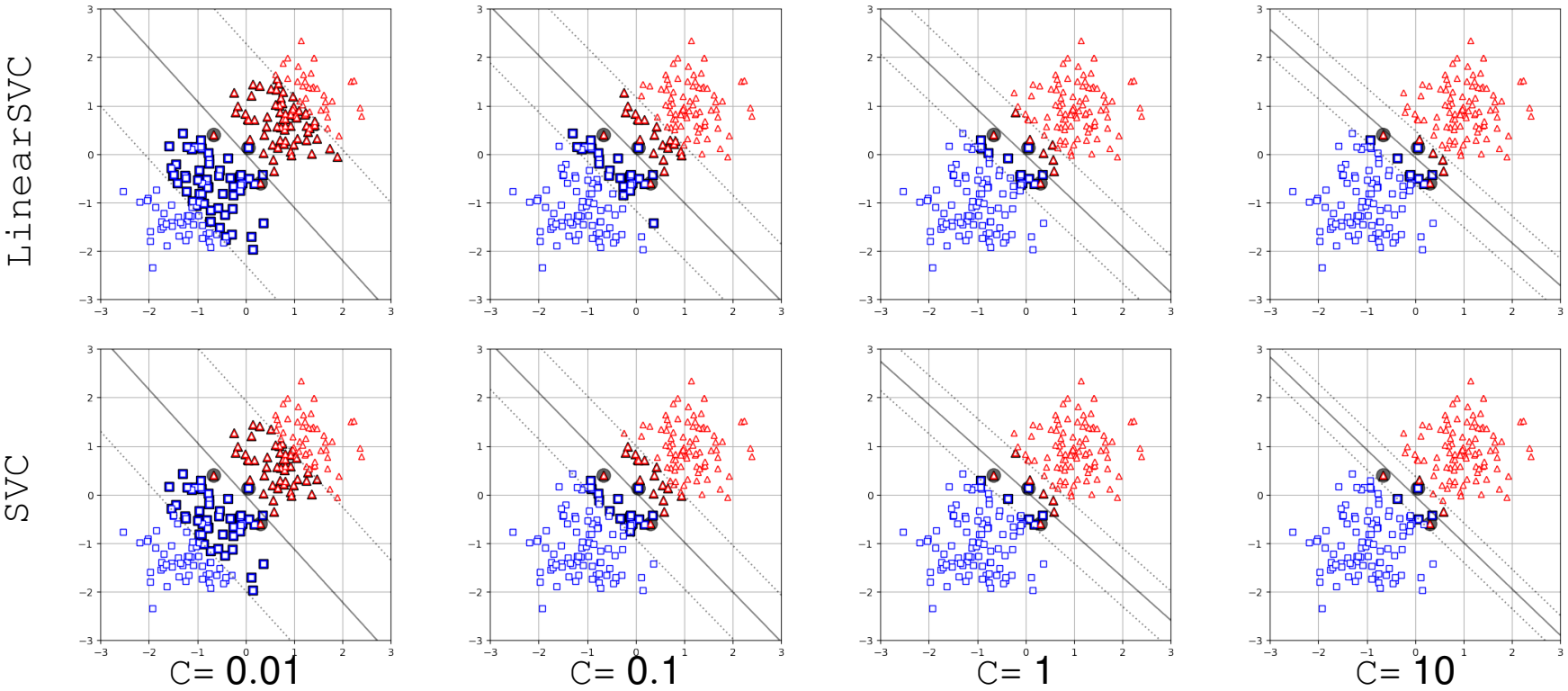
Máquinas de Suporte Vetorial - `sklearn`

Exemplo: SVMs lineares - Classificação binária

Pontos bidimensionais, pertencentes a duas classes, com:

$p(\mathbf{x}|\varpi_1) = \mathcal{N}(\mu_1, \mathbf{I})$ e $p(\mathbf{x}|\varpi_2) = \mathcal{N}(\mu_2, \mathbf{I})$, com $\mu_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$ e $\mu_2 = \begin{bmatrix} +1 \\ +1 \end{bmatrix}$, e com $p(\varpi_1) = p(\varpi_2)$.

As seguintes figuras mostram os resultados obtidos com SVMs lineares (`LinearSVC` ou `SVC(kernel='linear')`), e com diferentes valores para o parâmetro `C`.



- A função `LinearSVC` é mais eficiente que `SVC(kernel='linear')`. Usar a primeira em problemas com grandes quantidades de dados ou com dados de alta dimensão.

Máquinas de Suporte Vetorial - `sklearn`

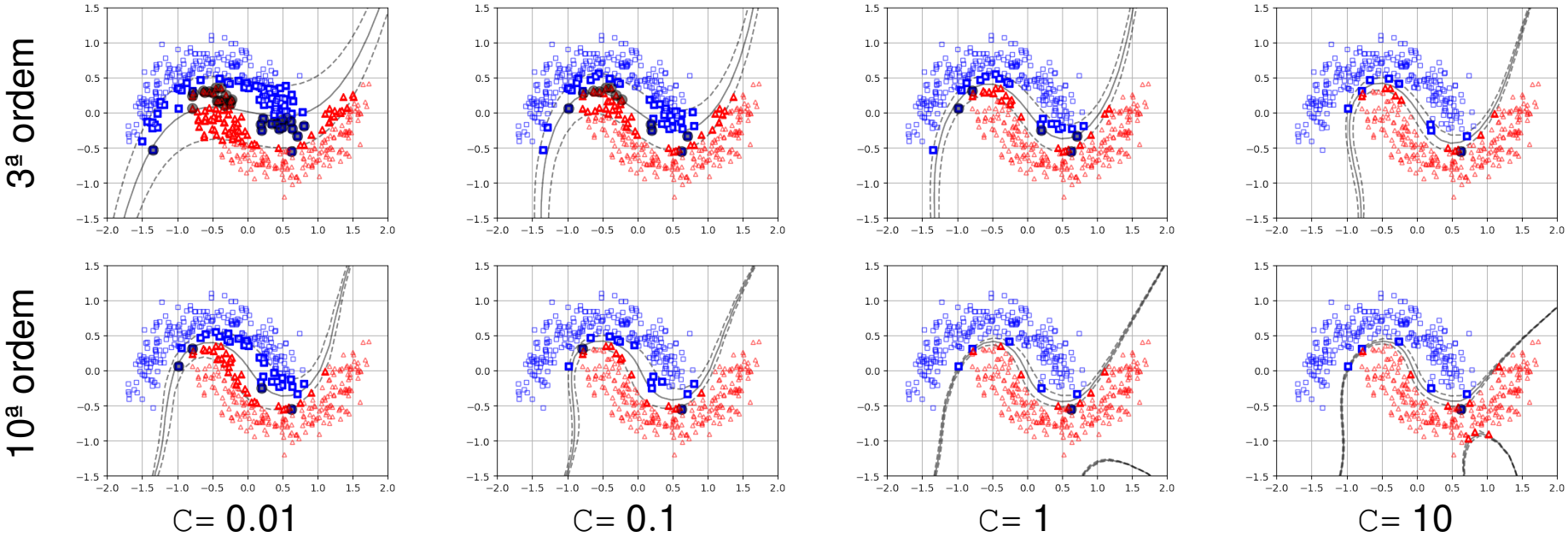
Exemplo: SVMs não-lineares - Classificação binária

Dados bidimensionais, pertencentes a duas classes, gerados com `make_moons`.

```
>>> from sklearn.datasets import make_moons
>>> X,y=make_moons(n_samples=500,noise=0.15,random_state=0)
```

As seguintes figuras mostram os resultados obtidos com *kernels* **polinomiais** de 3ª e 10ª ordem, e com diferentes valores para o parâmetro `C`.

```
>>> from sklearn.svm import SVC
>>> svm=SVC(kernel='poly',C=c,gamma='auto',degree=d,coef0=1)
```



- O parâmetro `coef0` é só usado nos *kernels* polinomiais e sigmoidais. Por omissão `coef0=0`. Atenção que valores diferentes de 0 podem alterar significativamente os resultados.

Máquinas de Suporte Vetorial - sklearn

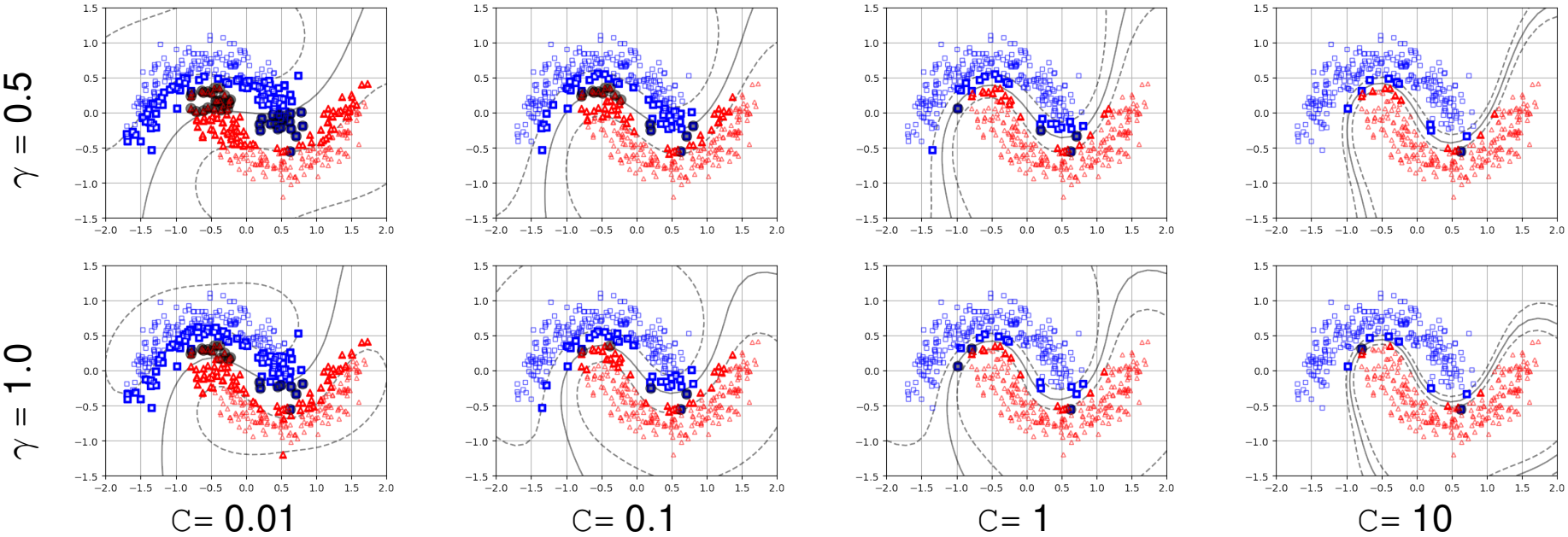
Exemplo: SVMs não-lineares - Classificação binária

Dados bidimensionais, pertencentes a duas classes, gerados com `make_moons`.

```
>>> from sklearn.datasets import make_moons
>>> X,y=make_moons(n_samples=500,noise=0.15,random_state=0)
```

As seguintes figuras mostram os resultados obtidos com *kernels* **RBF-Gausssianos**, com diferentes valores de γ, e com diferentes valores para o parâmetro `C`.

```
>>> from sklearn.svm import SVC
>>> svm=SVC(kernel='rbf',C=c,gamma=g)
```



- O parâmetro γ é inversamente proporcional à variância da gaussiana. Valores pequenos de γ correspondem a gaussianas com um raio grande o que implica que muitos pontos são considerados próximos uns dos outros.

Máquinas de Suporte Vetorial - `sklearn`

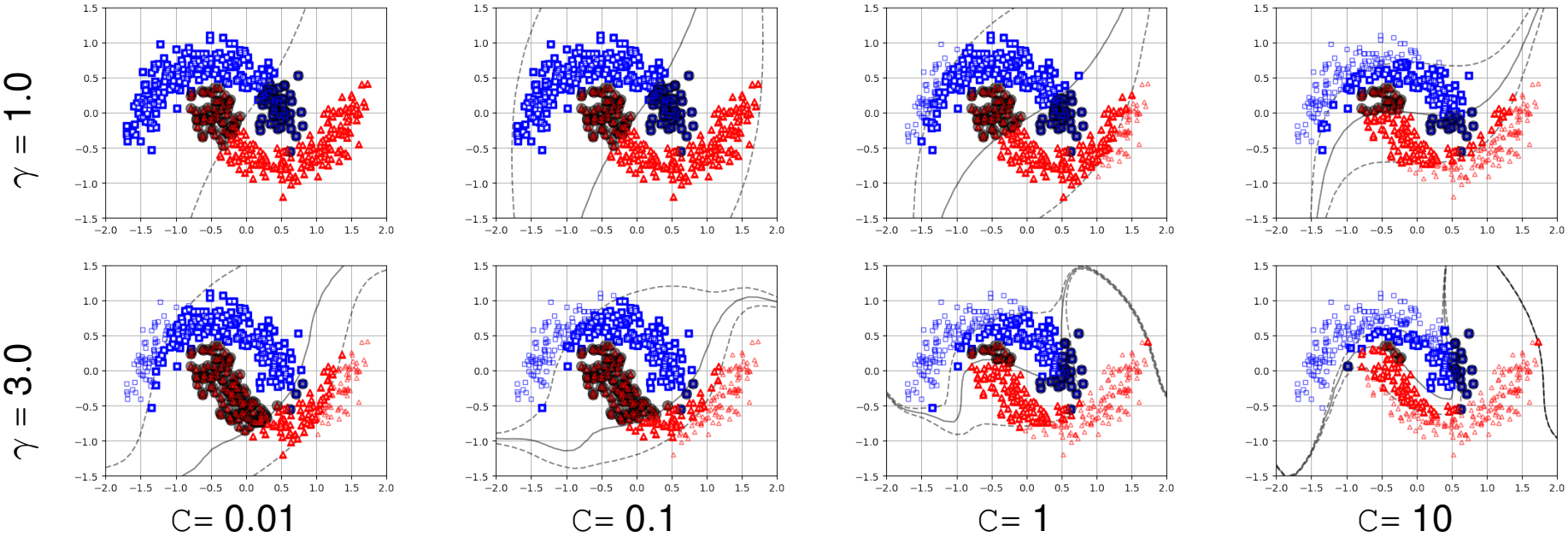
Exemplo: SVMs não-lineares - Classificação binária

Dados bidimensionais, pertencentes a duas classes, gerados com `make_moons`.

```
>>> from sklearn.datasets import make_moons
>>> X,y=make_moons(n_samples=500,noise=0.15,random_state=0)
```

As seguintes figuras mostram os resultados obtidos com *kernels* **sigmoidais**, com diferentes valores de γ, e com diferentes valores para o parâmetro `C`.

```
>>> from sklearn.svm import SVC
>>> svm=SVC(kernel='sigmoid',C=c,gamma=g,coef0=-5)
```



- O parâmetro `coef0` é só usado nos *kernels* polinomiais e sigmoidais. Por omissão `coef0=0`. Atenção que valores diferentes de 0 podem alterar significativamente os resultados.

Máquinas de Suporte Vetorial - sklearn

Exemplo: Classificação binária - Breast Cancer dataset

Pontos a 30 dimensões, pertencentes a duas classes. O conjunto foi dividido em treino e teste.

```
>>> from sklearn.datasets import load_breat_cancer
>>> from sklearn.model_selection import train_test_split
>>> BC=load_breast_cancer()
>>> X1,X2,y1,y2=train_test_split(BC.data,BC.target,test_size=.5,stratify=BC.target)
>>> svm=SVC(kernel='rbf',gamma='auto').fit(X1,y1)
>>> print('Treino - prob.  de acertos:  %.2f'%(svm.score(X1,y1)*100))
>>> print('Teste - prob.  de acertos:  %.2f'%(svm.score(X2,y2)*100))
Treino - prob.  de acertos:  100.00
Teste - prob.  de acertos:  62.81
```

ATENÇÃO: SVMs são classificadores bastante potentes mas também são muito sensíveis à escolha dos parâmetros e ao escalamento dos dados.

Neste caso, uma análise dos resultados revela que o SVM estimado é um classificador trivial, em que todas as predições são da classe dos positivos (não tem cancro).

```
>>> print(svm.predict(X2))
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ..., 1])
```

Máquinas de Suporte Vetorial - sklearn

Exemplo: Classificação binária - Breast Cancer dataset

Pontos a 30 dimensões, pertencentes a duas classes. O conjunto foi dividido em treino e teste.

```
>>> from sklearn.datasets import load_breat_cancer
>>> from sklearn.model_selection import train_test_split
>>> BC=load_breast_cancer()
>>> X1,X2,y1,y2=train_test_split(BC.data,BC.target,test_size=.5,stratify=BC.target)
>>> svm=SVC(kernel='rbf',gamma='auto').fit(X1,y1)
>>> print('Treino - prob. de acertos: %.2f'%(svm.score(X1,y1)*100))
>>> print('Teste - prob. de acertos: %.2f'%(svm.score(X2,y2)*100))
Treino - prob. de acertos: 100.00
Teste - prob. de acertos: 62.81
```

ATENÇÃO: SVMs são classificadores bastante potentes mas também são muito sensíveis à escolha dos parâmetros e ao escalamento dos dados.

Resultados com dados normalizados:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc=StandardScaler().fit(X1,y1)
>>> X1n=sc.transform(X1);X2n=sc.transform(X2)
>>> svm=SVC(kernel='rbf',gamma='auto').fit(X1n,y1)
>>> print('Treino - prob. de acertos: %.2f'%(svm.score(X1n,y1)*100))
>>> print('Teste - prob. de acertos: %.2f'%(svm.score(X2n,y2)*100))
Treino - prob. de acertos: 99.65
Teste - prob. de acertos: 96.49
```

Ajuste de Parâmetros de Modelos

Pesquisa em Grelha

O desempenho de muitos modelos, como discriminantes logísticos ou máquinas de suporte vetorial, depende da escolha (acertada) dos parâmetros do modelo. Descobrir os valores dos parâmetros que resultam no melhor desempenho e capacidade de generalização dos modelos não é uma tarefa trivial. Para a maioria dos modelos e dos conjuntos de dados é necessário este ajuste de parâmetros. Uma estratégia é testar todas as combinações possíveis dos valores dos parâmetros de interesse. Este processo é denominado "pesquisa em grelha" ou *grid search*, e o `sklearn` disponibiliza a classe `GridSearchCV` para este efeito.

Pesquisa em Grelha

Exemplo: Classificação binária - Breast Cancer dataset

Para encontrar os valores dos parâmetros γ e `C` para o classificador SVM, pode-se fazer um duplo ciclo `for`. Dividir o conjunto em treino e teste e normalizar os dados.

```
>>> from sklearn.datasets import load_breat_cancer
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> BC=load_breast_cancer()
>>> X1,X2,y1,y2=train_test_split(BC.data,BC.target,test_size=.5,stratify=BC.target)
>>> sc=StandardScaler().fit(X1)
>>> X1n=sc.transform(X1);X2n=sc.transform(X2)
>>> scoreTop=0.0
>>> parList=[0.001,0.01,0.1,1,10,100]
>>> for g in parList:
>>>    for C in parList:
>>>      svm=SVC(kernel='rbf',gamma=g,C=C).fit(X1n,y1)
>>>      score=SVC.score(X2n,y2)
>>>      if score>scoreTop:
>>>        scoreTop=score
>>>        param=(C,g)
Treino - prob.  de acertos:  99.30
Teste - prob.  de acertos:  96.84
Parâmetros:  C=100, gamma=0.001
```

Conjunto de Validação

No exemplo anterior, os resultados indicam que o desempenho do classificador SVM é aproximadamente 97%. No entanto esta estimativa pode ser demasiado otimista (ou até incorreta). Foram selecionados os parâmetros que resultam no melhor desempenho no conjunto de teste, e visto que usamos este conjunto para ajustar os parâmetros, já não pode ser usado para aferir o desempenho do modelo.

SOLUÇÃO: Dividir os dados em três conjuntos:

- Treino: serve para estimar os parâmetros do modelo.
- Validação: serve para selecionar os parâmetros do modelo.
- Teste: serve para avaliar o desempenho do modelo.

Depois de encontrar os parâmetros, re-treinar modelo com os conjuntos de treino e validação.

Conjunto de Validação

Exemplo: Classificação binária - Breast Cancer dataset

Encontrar os valores dos parâmetros γ e C para o classificador SVM. Dividir o conjunto em treino, validação e teste e normalizar os dados.

```
>>> BC=load_breast_cancer()
>>> X1,X2,y1,y2=train_test_split(BC.data,BC.target,test_size=1./3,stratify=BC.target)
>>> X1a,X1b,y1a,y1b=train_test_split(X1,y1,test_size=.5,stratify=y1)
>>> sc=StandardScaler().fit(X1a)
>>> X1an=sc.transform(X1a);X1bn=sc.transform(X1b)
>>> scoreTop=0.0
>>> parList=[0.001,0.01,0.1,1,10,100]
>>> for g in parList:
>>>    for C in parList:
>>>      svm=SVC(kernel='rbf',gamma=g,C=C).fit(X1an,y1a)
>>>      score=SVC.score(X1bn,y1b)
>>>      if score>scoreTop:
>>>        scoreTop=score
>>>        param=(g,C)
>>> X1n=sc.transform(X1);X2n=sc.transform(X2)
>>> svm=SVC(kernel='rbf',gamma=param[0],C=param[1]).fit(X1n,y1)
Treino - prob. de acertos: 100.00
Validação - prob. de acertos: 97.89
Teste - prob. de acertos: 94.73
Parâmetros: C=100, gamma=0.01
```

Pesquisa em grelha com validação cruzada

Na pesquisa em grelha, dividir uma única vez o conjunto de dados em treino, validação e teste pode resultar em estimativas ruidosas do desempenho, visto estas dependerem dos exemplos que foram atribuídos a cada um dos conjuntos.

Para obter uma estimativa do desempenho mais fidedigna, é melhor usar técnicas de validação cruzada.

ATENÇÃO: o ajuste dos parâmetros do modelo é um processo computacionalmente pesado, em particular se se usar validação cruzada. É boa ideia começar com grelhas pequenas, e depois afinar as buscas.

Pesquisa em grelha com validação cruzada

Exemplo: Classificação binária - Breast Cancer dataset

Encontrar os valores dos parâmetros γ e C para o classificador SVM, usando técnicas de validação cruzada.

```
>>> BC=load_breast_cancer()
>>> X1,X2,y1,y2=train_test_split(BC.data,BC.target,test_size=0.25,stratify=BC.target)
>>> sc=StandardScaler().fit(X1)
>>> X1n=sc.transform(X1);X2n=sc.transform(X2)
>>> scoreTop=0.0
>>> parList=[0.001,0.01,0.1,1,10,100]
>>> for g in parList:
>>>    for C in parList:
>>>      svm=SVC(kernel='rbf',gamma=g,C=C)
>>>      scores=cross_val_score(svm,X1n,y1,cv=3)
>>>      if np.mean(scores)>scoreTop:
>>>        scoreTop=np.mean(scores)
>>>        param=(g,C)
>>> svm=SVC(kernel='rbf',gamma=param[0],C=param[1]).fit(X1n,y1)
Treino - prob.  de acertos:  99.06
Teste - prob.  de acertos:  97.20
Parâmetros:  C=10, gamma=0.01
```

Pesquisa em grelha com validação cruzada

Exemplo: Classificação binária - Breast Cancer dataset

Encontrar os valores dos parâmetros γ e `C` para o classificador SVM, usando técnicas de validação cruzada.

Pode-se usar a classe `GridSearchCV` do sub-módulo `model_selection` para ajustar os parâmetros. Para tal, é necessário especificar quais parâmetro se pretende ajustar, instanciando-os com um dicionário, em que as chaves são os nomes dos parâmetros e os valores são os valores a testar.

```
>>> from sklearn.model_selection import train_test_split
>>> BC=load_breast_cancer()
>>> X1,X2,y1,y2=train_test_split(BC.data,BC.target,test_size=0.25,stratify=BC.target)
>>> sc=StandardScaler().fit(X1)
>>> X1n=sc.transform(X1);X2n=sc.transform(X2)
>>> parList=[0.001,0.01,0.1,1,10,100]
>>> grelha={'C':parList,'gamma':parList}
>>> gSearch=GridSearchCV(SVC(kernel='rbf'),grelha,cv=3)
>>> gSearch.fit(X1n,y1)
>>> gSearch.score(X1n,y1)
>>> gSearch.score(X2n,y1)
>>> gSearch.best_params_
Treino - prob.  de acertos:  99.06
Teste - prob.  de acertos:  97.20
Parâmetros:  C=10, gamma=0.01
```

Pesquisa em grelha com validação cruzada

Exemplo: Classificação binária - Breast Cancer dataset

Encontrar os valores dos parâmetros γ e `C` para o classificador SVM, usando técnicas de validação cruzada.

Pode-se usar a classe `GridSearchCV` do sub-módulo `model_selection` para ajustar os parâmetros. Para tal, é necessário especificar quais parâmetro se pretende ajustar, instanciando-os com um dicionário, em que as chaves são os nomes dos parâmetros e os valores são os valores a testar.
No exemplo prévio, os dados foram divididos em treino e teste, e foi implementado um esquema de validação cruzada usando o `GridSearchCV` no conjunto de treino. No entanto só foi feito uma divisão em treino e teste, e os resultados no conjunto de teste podem ser instáveis devido a esta única divisão. Pode-se então usar a classe `cross_val_score` em conjunto com `GridSearchCV` para fazer também validação cruzada no conjunto de teste.

```
>>> sc=StandardScaler().fit(BC.data)
>>> Xn=sc.transform(BC.data)
>>> parList=[0.001,0.01,0.1,1,10,100]
>>> grelha={'C':parList,'gamma':parList}
>>> scores=cross_val_score(GridSearchCV(SVC(kernel='rbf'),grelha,cv=3),Xn,BC.target,cv=5)
Prob.  de acertos/fold :  [97.39 97.39 98.23 97.35 99.12]
Prob.  média de acerto :  97.89
```

ATENÇÃO: este processo é extremamente pesado do ponto de vista computacional. Neste caso foram treinados e testados $6 \times 6 \times 3 \times 5 = 540$ modelos!