# $A^3$ Aprendizagem Automática Avançada

## Log-Linear Classifiers

## and

## Gradient Descent Methods

𝔾. Marques

# Log-Linear Classifiers

# Log-Linear Classifiers

- Log-linear classifiers are a generalization of linear models. They are also known as **logistic discriminants**, **logistic regression**, and **maximum entropy classifiers**.
- Log-linear have linear decisions functions which have several limitations in complex classification problems (ex: half-moons, xor, etc).
- A log-linear classifier with a single output can be viewed as a *perceptron* or a single *neuron*, the primary building block used in artificial neural networks.
- One of the most popular types of neural networks, the multi-layer perceptron (MLP) is a series of log-linear classifiers chained together.
- Log-linear classifier consist of a linear transformation of the input data followed by non-linear saturation called an *activation function*.

# Log-Linear Classifiers - Binary Case

Consider that a $d$-dimensional vector $\mathbf{x}$, with classes $\Omega = \{\varpi_0, \varpi_1\}$.

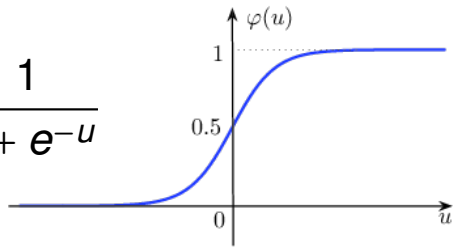
- Model: $\hat{y} = \varphi\left(\mathbf{w}^\top \mathbf{x}\right)$
  Linear transform followed by an saturation:

$$\hat{y} = \varphi(u) \quad \text{with} \quad u = \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_d \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

- Activation function $\varphi()$, is one of the following two functions:

sigmoid

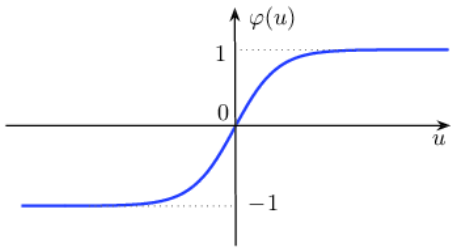
$$\varphi(u) = \frac{1}{1 + e^{-u}}$$




or

hyperbolic tangent

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




- Classification: if $\hat{y} < \beta$, $\mathbf{x} \in \varpi_0$, else $x \in \varpi_1$, where $\beta$ is a predefined threshold value.
  Sigmoid: $\beta = 0.5$
  Hyperbolic tangent: $\beta = 0$.

# Log-Linear Classifiers - Binary Case

- Supervised learning problem:
  - The goal is to determine the vector $\mathbf{w}$ based on a dataset of $N$ vectors $\mathbf{x}[n]$, and the corresponding set of $N$ desired output values $y[n]$, with $n = 1, \ldots, N$.
  - The outputs $y$ have the data class information.
    Sigmoid: $y \in \{0, 1\}$, if $y = 0$, $\mathbf{x} \in \varpi_0$ and if $y = 1$, $\mathbf{x} \in \varpi_1$
    Hyperbolic tangent: $y \in \{-1, +1\}$, if $y = -1$, $\mathbf{x} \in \varpi_0$ and if $y = 1$, $\mathbf{x} \in \varpi_1$
- Minimum Mean Squared Error (MSE) Estimation:
  The coefficient values of the weight vector pesos, $\mathbf{W}$, can be obtained through an adaptive minimization of the MSE functional.
  - $\mathcal{E}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \left( y[n] - \varphi\left(\mathbf{w}^\top \mathbf{x}[n]\right)\right)^2 = \frac{1}{N} \sum_{n=1}^{N} \left( y[n] - \hat{y}[n]\right)^2$
  - The MSE, $\mathcal{E}(\mathbf{w})$ is the average error between the desired responses and the outputs of the logistic discriminant.
  - The coefficients of $\mathbf{w}$ are obtained in an adaptive fashion. In each iteration, the values of $\mathbf{w}$ are modified so that the MSE decreases, compared to the previous iteration.
  - For the iteration $i+1$, $\mathbf{w}$ depends on its previous value (iteration $i$) minus term dependent on the *gradient*.

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla \mathcal{E}(\mathbf{w})$$

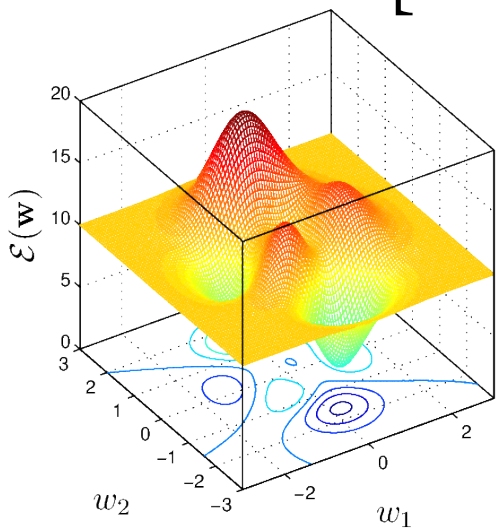
# Log-Linear Classifiers - Binary Case

- Supervised learning problem:
  - The goal is to determine the vector $\mathbf{w}$ based on a dataset of $N$ vectors $\mathbf{x}[n]$, and the corresponding set of $N$ desired output values $y[n]$, with $n = 1, \ldots, N$.
  - The outputs $y$ have the data class information.
    Sigmoid: $y \in \{0, 1\}$, if $y = 0$, $\mathbf{x} \in \varpi_0$ and if $y = 1$, $\mathbf{x} \in \varpi_1$
    Hyperbolic tangent: $y \in \{-1, +1\}$, if $y = -1$, $\mathbf{x} \in \varpi_0$ and if $y = 1$, $\mathbf{x} \in \varpi_1$
- Regularization:
  One can add a regularization term to the MSE function to penalize large value coefficients (in absolute terms).
  - $\mathcal{E}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} (y[n] - \hat{y}[n])^2 + \lambda \sum_{w_i \in \mathbf{w}} |w_i|^\rho$
  - $\lambda$ controls the weight of the regularization term.
  - Quadratic regularization, for $\rho = 2$ (a.k.a. $\ell_2$ or *ridge*)
  - *Lasso* regularization for $\rho = 1$ (a.k.a. $\ell_1$). In Lasso regularization has a tendency to set to zero part of the coefficients, which can be useful to discard irrelevant data dimensions.

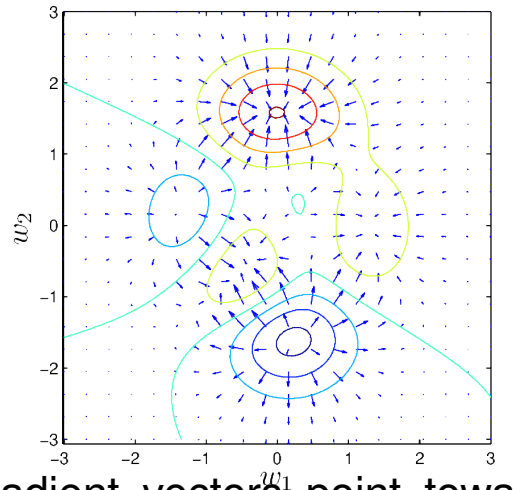
# Gradient Descent

- Gradient descent is a first order optimization technique to find a local minimum of a function.
- The parameters **w** are adapted iteratively so that in each iteration the error value is lower (or equal to) than in the previous iteration.
- The gradient is a multi-variable generalization of the derivative. The gradient of **w** must have same dimensions of **w**:

$$\frac{\partial \mathcal{E}(\mathbf{w})}{\partial \mathbf{w}} = \left[ \frac{\partial \mathcal{E}(\mathbf{w})}{\partial w_0}, \frac{\partial \mathcal{E}(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial \mathcal{E}(\mathbf{w})}{\partial w_d} \right]^T$$




$\mathcal{E}(\mathbf{w})$ in terms of $w_0$ and $w_1$




Gradient vectors point towards the maximums of $\mathcal{E}(\mathbf{w})$

# Gradient Descent

- Gradient descent is a first order optimization technique to find a local minimum of a function.
- The parameters $\mathbf{w}$ are adapted iteratively so that in each iteration the error value is lower (or equal to) than in the previous iteration.
- The gradient is a multi-variable generalization of the derivative. The gradient of $\mathbf{w}$ must have same dimensions of $\mathbf{w}$:

$$\frac{\partial \mathcal{E}(\mathbf{w})}{\partial \mathbf{w}} = \left[ \frac{\partial \mathcal{E}(\mathbf{w})}{\partial w_0}, \frac{\partial \mathcal{E}(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial \mathcal{E}(\mathbf{w})}{\partial w_d} \right]^\top$$

- The gradient vector points towards the direction of maximum growth of the function.
- Adapt the weight in the opposite direction of the gradient:

$$\mathbf{w}(i+1) = \mathbf{w}(i) - \eta \frac{\partial \mathcal{E}(\mathbf{w})}{\partial \mathbf{w}}$$

where $i$ is the current iteration and $\eta$ is the *learning rate* or the *adaptation step* ($0 < \eta << 1$).

# Gradient Descent

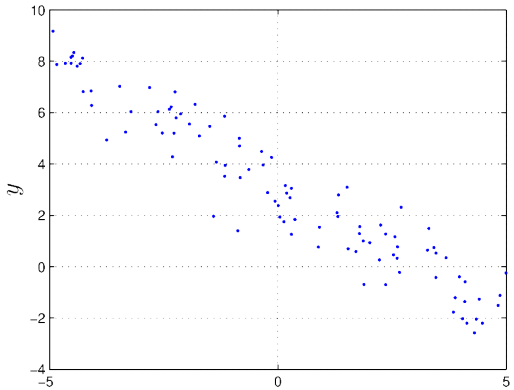
## Pseudo-code for gradient descent:

1. Initialize the weight vector $\mathbf{w}$ and the learning rate $\eta$
2. Determine the error $\mathcal{E}(\mathbf{w})$.
3. Determine the error gradient in terms of the weights: $\frac{\partial \mathcal{E}(\mathbf{w})}{\partial \mathbf{w}}$
4. Adapt the weights: $\mathbf{w}(i+1) = \mathbf{w}(i) - \eta \frac{\partial \mathcal{E}(\mathbf{w}(i))}{\partial \mathbf{w}}$
5. Repeat steps 2-4 until the error cannot be reduced: $\mathcal{E}(\mathbf{w}(i+1)) \approx \mathcal{E}(\mathbf{w}(i))$

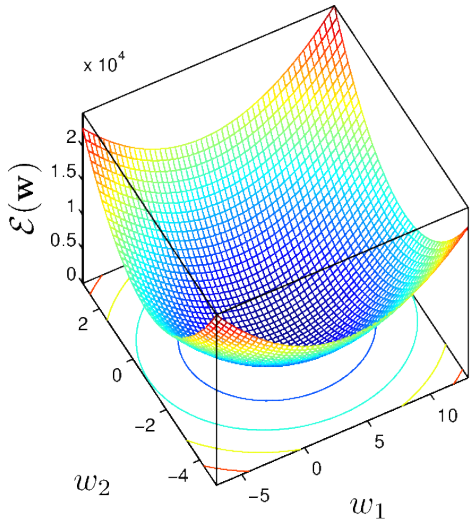
# Gradient Descent

## Simple example: linear regression

- Data distributed along a line ($N$ point pairs $(x, y)$)




data points




MSE function

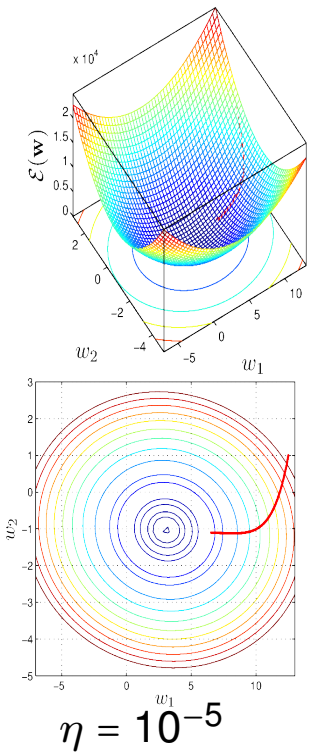
- Model: $\hat{y} = w_1 + w_2 x$
- MSE: $\mathcal{E}(\mathbf{w}) = \frac{1}{N}\sum_n (y[n] - \hat{y}[n])^2 = \frac{1}{N}\sum_n (y[n] - w_1 - w_2 x[n])^2$
- Objective: find the parameters of the line $(w_1, w_2)$
- Adaptations:
  - $w_1(i+1) = w_1(i) - \eta \frac{\partial \mathcal{E}(\mathbf{w}(i))}{\partial w_1} = w_1(i) - \eta \frac{-2}{N}\sum_n (y[n] - \hat{y}[n])$
  - $w_2(i+1) = w_2(i) - \eta \frac{\partial \mathcal{E}(\mathbf{w}(i))}{\partial w_2} = w_2(i) - \eta \frac{-2}{N}\sum_n (y[n] - \hat{y}[n])\, x[n]$

# Gradient Descent

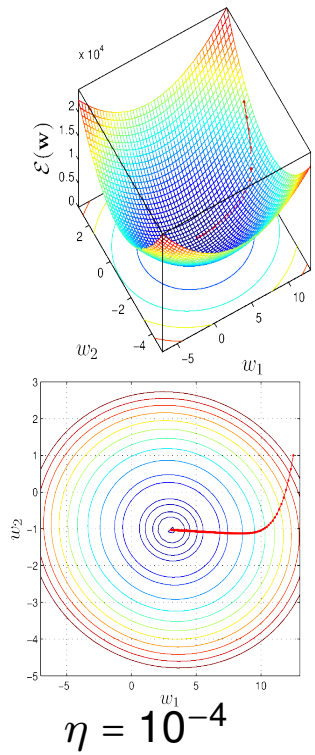
## Simple example: linear regression

### Choosing the value of $\eta$

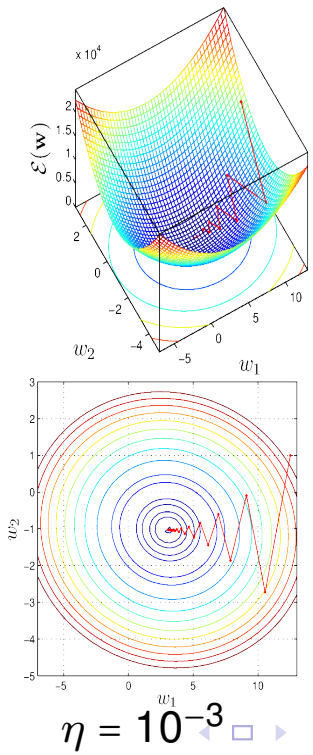
- The value of the learning rate has a significant influence on the convergence of the algorithm.
- If $\eta$ is too small, the minimum of the function is never reached.
- If $\eta$ is too large, the algorithm diverges.



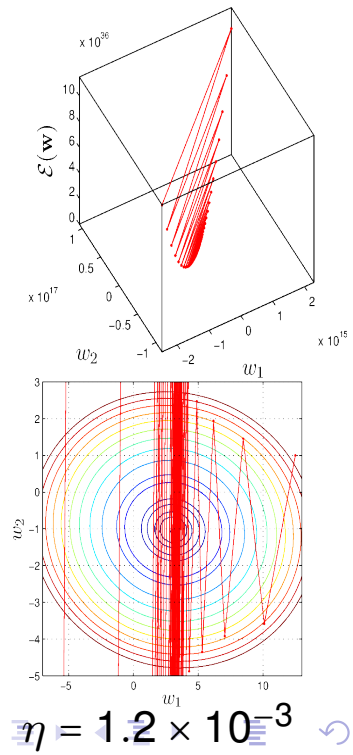

$\eta = 10^{-5}$




$\eta = 10^{-4}$




$\eta = 10^{-3}$




$\eta = 1.2 \times 10^{-3}$

# Gradient Descent
## Momentum Term

- Method to speed up the convergence of the adaptation.
- Modify the weights based on a filter version of the gradient: the momentum term:

$$\mathbf{z}(i) = \frac{\partial \mathcal{E}\left(\mathbf{w}(i)\right)}{\partial \mathbf{w}} + \alpha \mathbf{z}(i-1) = \frac{\partial \mathcal{E}\left(\mathbf{w}(i)\right)}{\partial \mathbf{w}} + \sum_{k=1}^{+\infty} \alpha^k \frac{\partial \mathcal{E}\left(\mathbf{w}(i-k)\right)}{\partial \mathbf{w}}$$

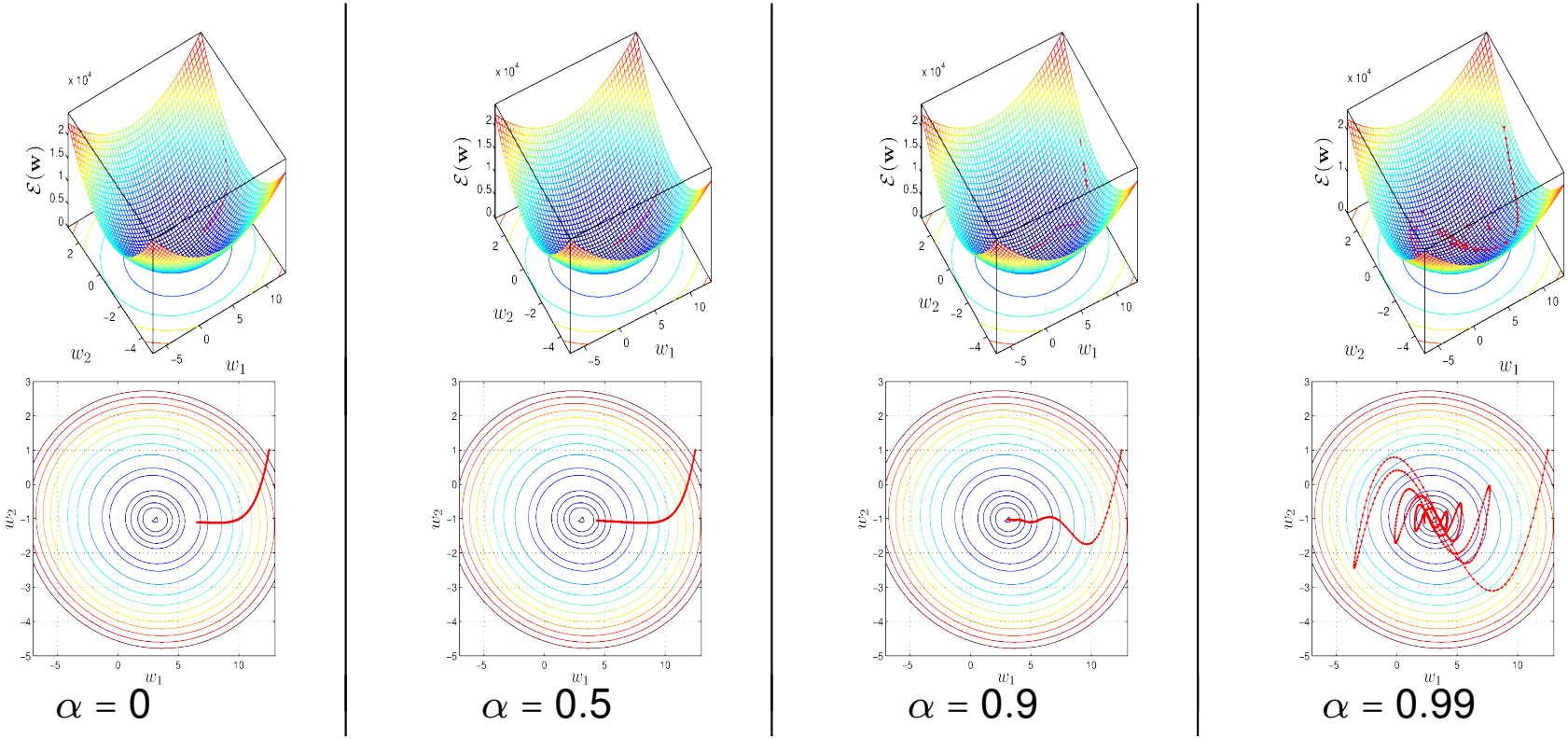
$$\mathbf{w}(i+1) = \mathbf{w}(i) - \eta \mathbf{z}(i)$$

  where $0 \leq \alpha < 1$

- From a signal processing point of view, the momentum term is a low-pass, IIR-filtered version of the gradient.
- When consecutive gradient vectors point in the same direction, the momentum term, $\mathbf{z}$ increases.
- When consecutive gradient vectors point in opposite directions, the momentum term, $\mathbf{z}$ decreases.

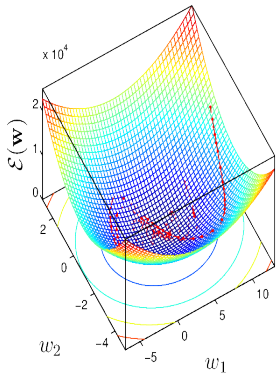
# Gradient Descent
## Momentum Term

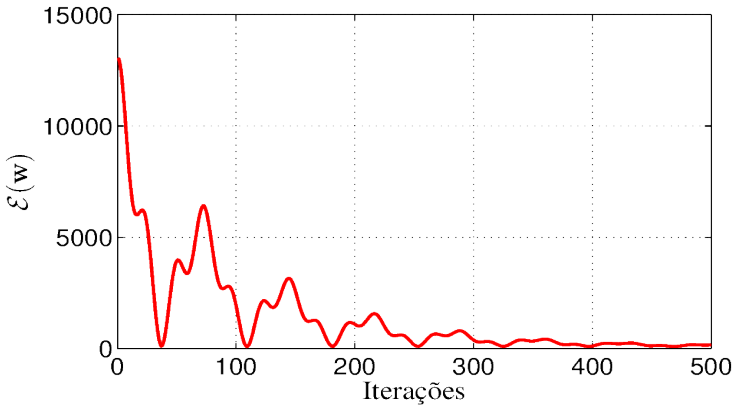
- Previous example for different values of $\alpha$ and with $\eta = 10^{-5}$.

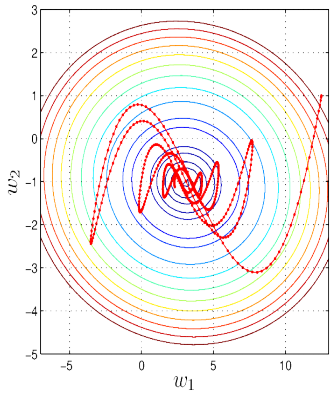


# Gradient Descent
## Momentum Term

- Previous example with $\eta = 10^{-5}$ and $\alpha = 0.99$
- $\alpha$ values close to 1 make it difficult for the algorithm to stop ($\alpha > 1$: unstable).




One way to counteract this behavior is to:

- save **w** and the gradient every time the error minimum is reached.
- if in the current iteration the error is higher than the minimum value, go back:
  - Restore the best weights
  - Re-initialize the momentum $\mathbf{z}(i-1) = 0$

# Gradient Descent
## Batch and On-line Adaptations:

**Batch** Batch gradient descent uses every point in the training set to calculate the error and the gradients. This can slow down significantly the convergence process, in particular for large size datasets.

**On-line** On-line or *stochastic* gradient descent, the error and the gradients are estimated using a single instance.

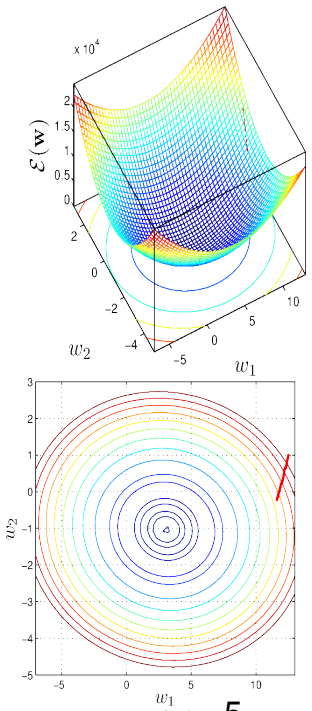
$$\hat{\mathcal{E}}_n(\mathbf{w}) = \left(y_n - \mathbf{w}^\top \mathbf{x}_n\right)^2$$

$$\frac{\partial \hat{\mathcal{E}}_n(\mathbf{w})}{\partial \mathbf{w}} = -2\left(y_n - \mathbf{w}^\top \mathbf{x}_n\right)\mathbf{x}_n$$

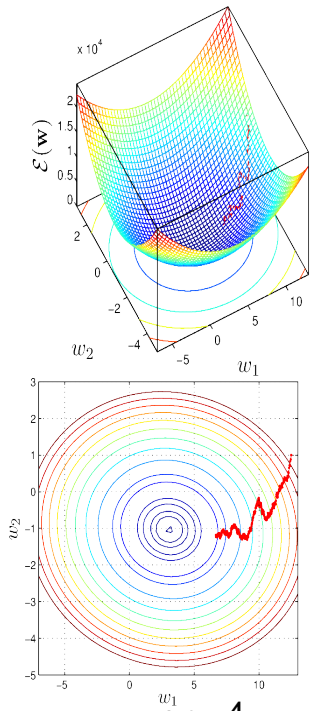
- The error and the gradients are noisy estimation of the true error and gradients (obtained with all the points).
- In stochastic adaptations, the use of th momentum term is advisable to filter out the noise oscillations.
- Stochastic adaptations are less prone to get stuck in local minima than the batch counterpart.
- To guaranty convergence, the learning rate must reduce during the adaptation. $\eta(i+1) = \eta(i)i^{-1}$ ou $\eta(i+1) = \eta(i)\beta^i$ com $0 < \beta < 1$
- The type of gradient descent is widely used in deep learning and for large datasets.

# Gradient Descent
## Batch and On-line Adaptations:

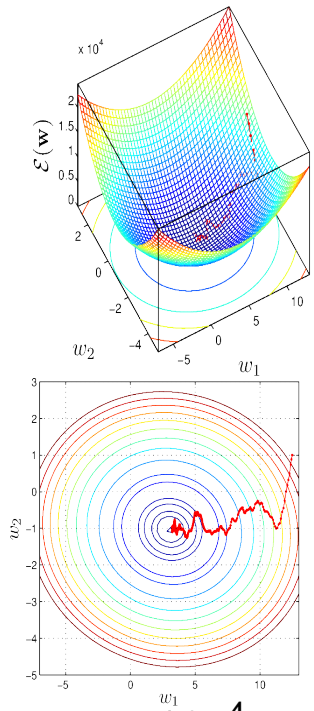
- Previous example with stochastic adaptation.



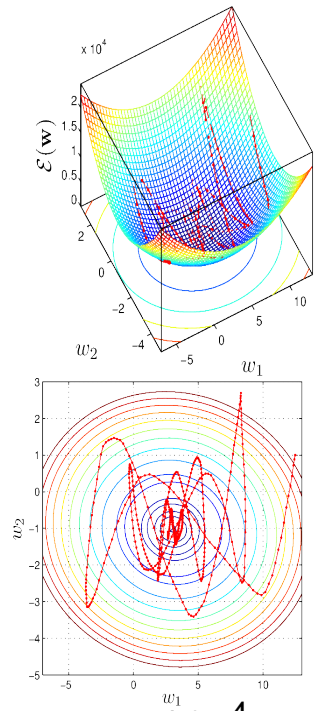

$\eta = 10^{-5}$
$\alpha = 0$




$\eta = 10^{-4}$
$\alpha = 0$




$\eta = 10^{-4}$
$\alpha = 0.8$

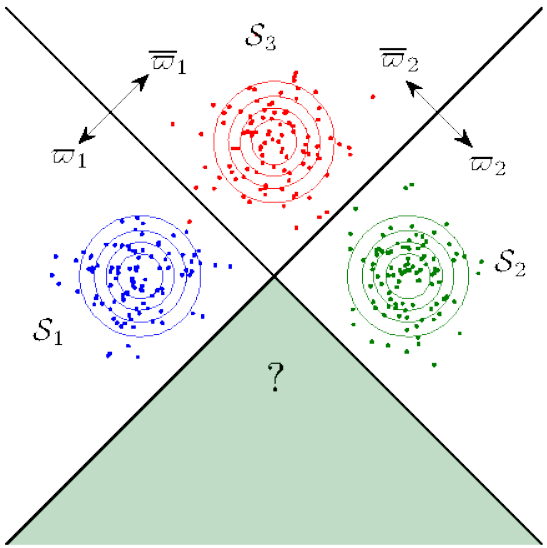



$\eta = 10^{-4}$
$\alpha = 0.99$

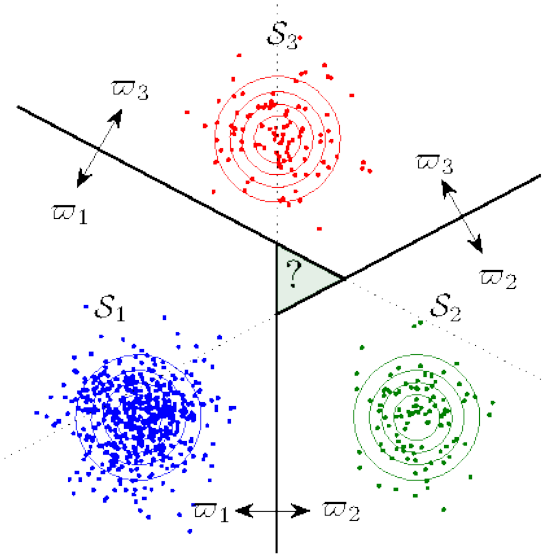
# Log-Linear Classifiers - multi-class case

- The goal in multi-class classification is to categorize an observation into one of a set of mutually exclusive classes. Binary classifiers can be adapted to tackle a multi-class problems. For that, two types of strategies can be considered:

One Versus the Rest




One Versus the Other




- Both of these strategies have limitations.
- Solution: use classifiers with as many outputs as the number of classes.

# Log-Linear Classifiers - multi-class case

- Model:
  Linear transform followed by an saturation: $\hat{\mathbf{y}} = \varphi\left(\mathbf{W}^\top \mathbf{x}\right)$

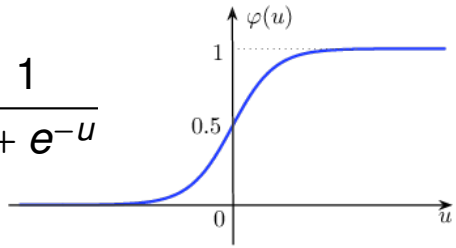
$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_c \end{bmatrix} = \begin{bmatrix} \varphi(u_1) \\ \varphi(u_2) \\ \vdots \\ \varphi(u_c) \end{bmatrix} = \varphi\left( \underbrace{\begin{bmatrix} w_{01} & w_{11} & w_{21} & \cdots & w_{d1} \\ w_{02} & w_{12} & w_{22} &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

  with $u_k = \mathbf{w}_k^\top \mathbf{x} = w_{0k} + w_{1k} x_1 + \ldots + w_{dk} x_d, \quad k = 1, \ldots, c$, where $\mathbf{w}_k$ is the $k^{\text{th}}$ column vector of the matrix $\mathbf{W}$ ($\mathbf{u}$ - vector of dimension $c \times 1$: $\mathbf{u} = \mathbf{W}^\top \mathbf{x}$).

- Activation function $\varphi()$, is one of the following two functions:

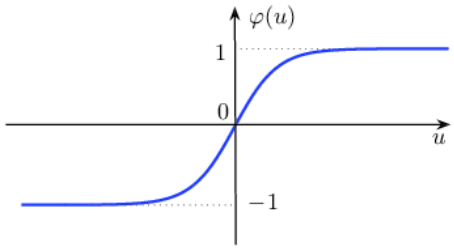
sigmoid

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




hyperbolic tangent

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

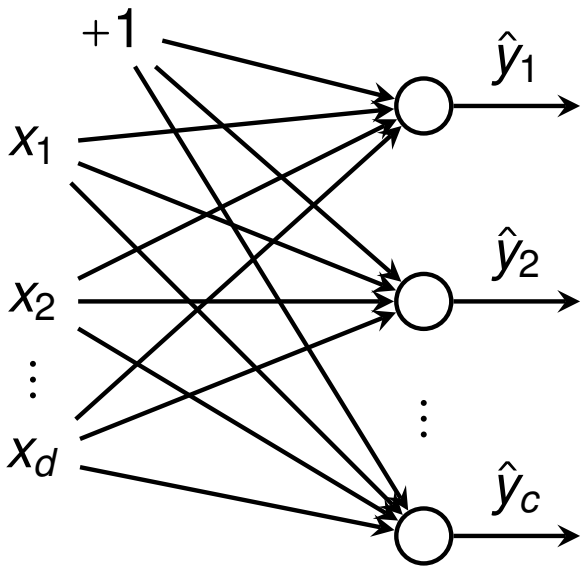


# Log-Linear Classifiers - multi-class case

- Model:
  Linear transform followed by an saturation: $\hat{\mathbf{y}} = \varphi\left(\mathbf{W}^\top \mathbf{x}\right)$

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

- Graphical representation:



# Log-Linear Classifiers - multi-class case

- Supervised learning problem:
  - The goal is to determine **W** based on a dataset of $N$ vectors **x**, and the corresponding set of $N$ desired output vectors **y**.
  - Note that **y** are $c \times 1$ vectors. Typically the class labels are integer numbers and they need to be converted to vectors (this is also true for categorical variables). One simple way is to create a $c$-dimensional vector, and set all coefficient to zero (or $-1$) except the one corresponding to the class label. This technique is commonly known as "**one hot encoding**".
  - The output vectors **y** have the class information about **x**. These are $c$-dimensional vectors, with one of its coefficient values set to "$+1$" in the $k^{\text{th}}$ entry, indicating that $\mathbf{x} \in \varpi_k$. The remaining vector entries are either "0" or "$-1$", depending on the chosen activation:

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

sigmoid hyperbolic tangent

# Log-Linear Classifiers - multi-class case

- Supervised learning problem:
  - The goal is to determine $\mathbf{W}$ based on a dataset of $N$ vectors $\mathbf{x}$, and the corresponding set of $N$ desired output vectors $\mathbf{y}$.
- MSE Estimation:
  Same as the two-class example, but now a weight matrix (instead of a vector) is calculated
  - $\mathcal{E}(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\left\|\mathbf{y}[n] - \hat{\mathbf{y}}[n]\right\|^2) + \lambda \sum_{w_{ij}\in\mathbf{W}} |w_{ij}|^{\rho}$
  - The weight matrix $\mathbf{W}$ is obtained in an iterative fashion (through gradient descent).

$$\mathbf{W}_{i+1} = \mathbf{W}_i - \eta\frac{\partial\mathcal{E}(\mathbf{W}_i)}{\partial\mathbf{W}}$$

# Log-Linear Classifiers - `sklearn`

- `LogisticRegression`
  Dispite its name, the method is used for classification, not regression.

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

- Several parameters to consider.
  - `tol`: $10^{-4}$ (`default`). Stopping criterion.
  - `max_iter`: 100 (`default`). Maximum number of iterations.
  - `solver`: `liblinear` (`default`). Optimization method.
    `liblinear` limited to binary classification.
    For multi-class, use `newton-cg, lbfgs, sag, saga`.
    You need to initialize the parameter `multi_class`.
  - `multi_class`: `ovr` (`default`). Binary or multi-class classification.
    `ovr` One Versus the Rest - binary.
    `multinomial` multi-class.
    `auto` automatic adjustment.

# Log-Linear Classifiers - `sklearn`

- `LogisticRegression`
  Dispite its name, the method is used for classification, not regression.

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

- Regularization:
  By default, log-linear classifiers in `sklearn` uses *ridge* (a.k.a $\ell_2$) regularization. The weight of the regularization term is controlled by the parameter `C`, and the type of regularization, *lasso* or *ridge*, by the term `penalty`.
  - `C`: 1.0 (`default`). Inversely proportional to the regularization weight. Small values correspond to a "strong" regularization.
  - `penalty`: `l2` (`default`). Type of regularization.
    *lasso*: `l1`.
    *ridge*: `l2`.
  - The adaptations `newton-cg`, `lbfgs` and `sag` only support $\ell_2$ regularization. `liblinear` and `saga` support both types of regularization.

# Log-Linear Classifiers - `sklearn`

- `LogisticRegression`
  Dispite its name, the method is used for classification, not regression.

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

- Outputs:
  After training the classifier (done with the function `.fit()`), one can access the matrix **W** and the number of iterations that were done during adaptation.
  - `Dlog.coef_`: Matrix **W**, (excluding the bias terms - 1$^{st}$ line of the matrix).
  - `Dlog.intercept_`: Biases terms (matrix coefficients $w_{0i}$, $i = 1, \ldots, c$)
  - `Dlog.n_iter_`: number of performed iterations

  ATTENTION: For a two-class problem, the weights are stored in a vector.

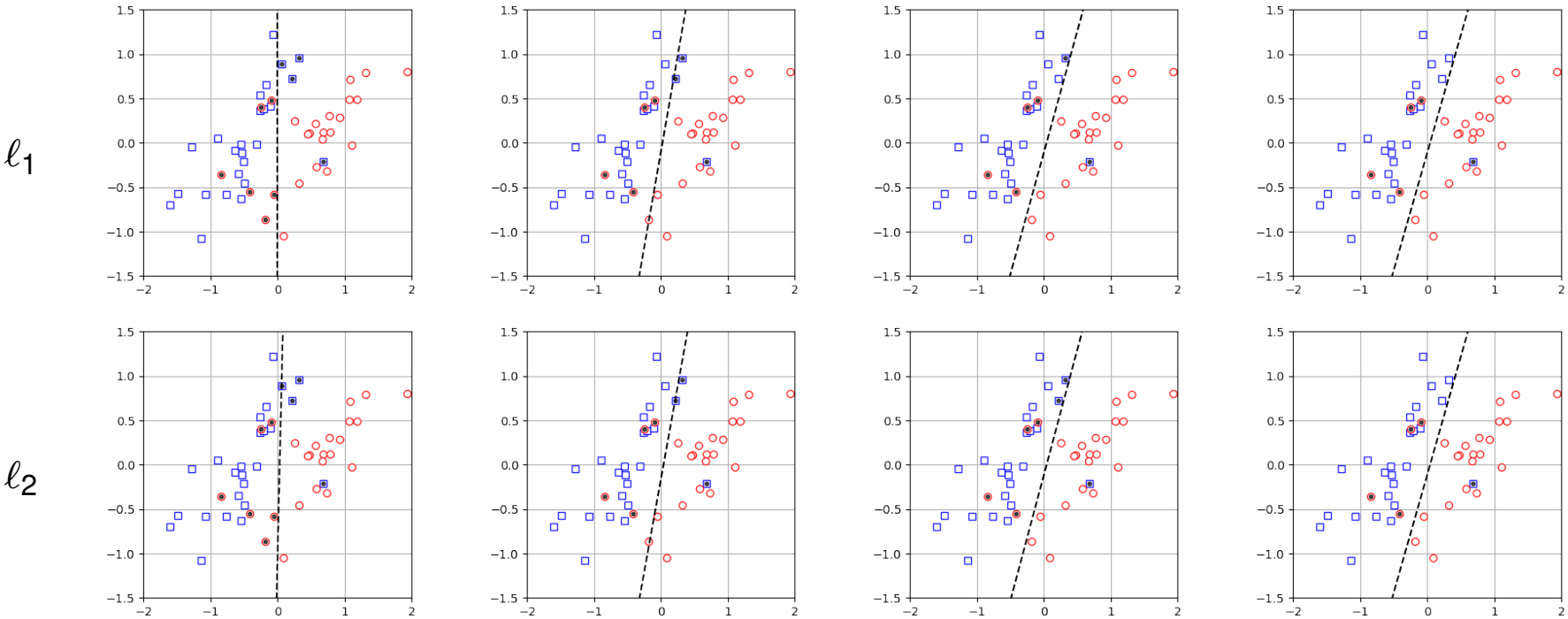
# Log-Linear Classifiers - `sklearn`

## Example: Binary Classification

DATA: `binClassData.p`

Two-dimensional data, divided in two classes.

In the next figures are the results of a log-linear classifier with $\ell_1$ e $\ell_2$ regularization, and different values for the parameter `C`.



# Log-Linear Classifiers - sklearn

## Example: Binary Classification - Breast Cancer dataset

30-dimensional data, divided in two classes..

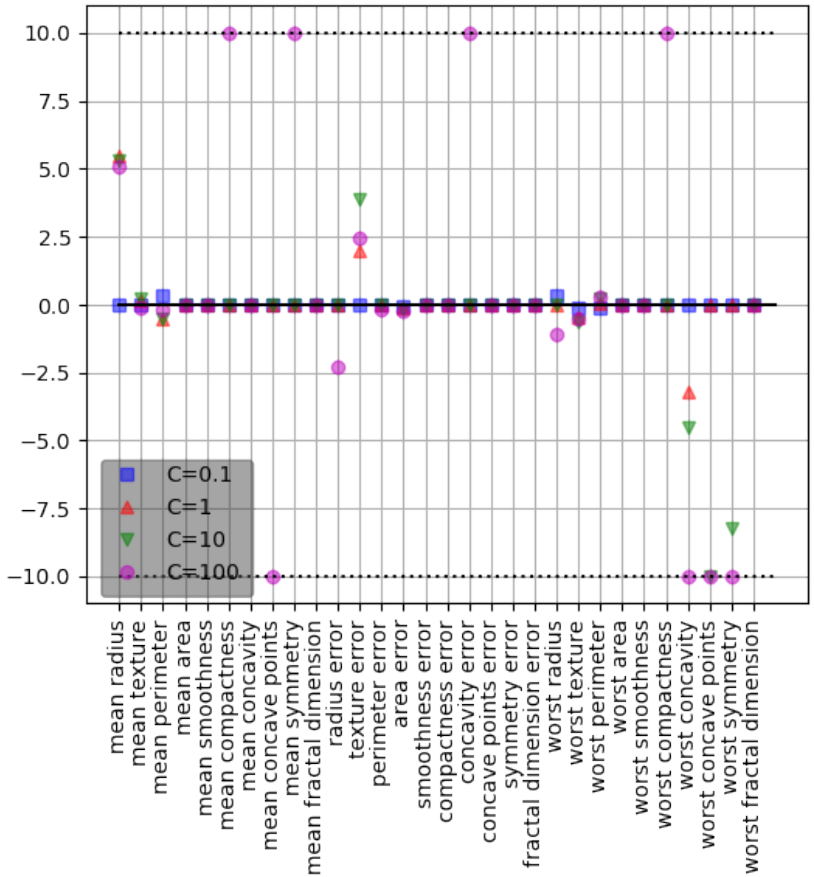
Note that the weights are stored in a 30-dimensional vector, and each coefficient multiplies one of the data dimensions.

```
>>> 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_)
```

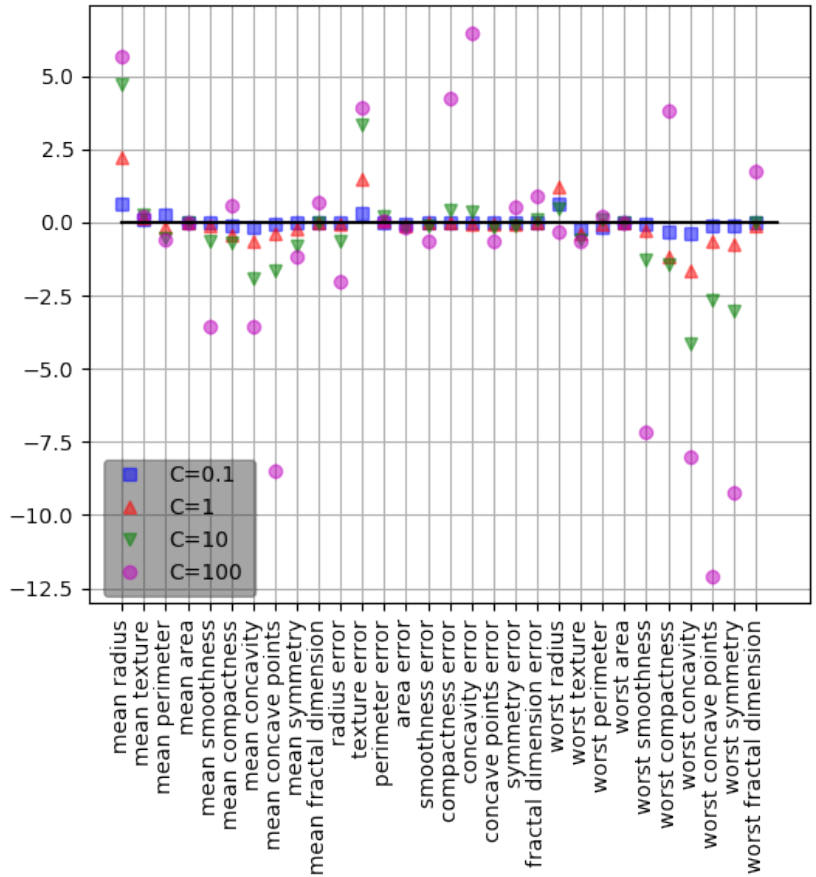
# Log-Linear Classifiers - `sklearn`

## Example: Binary Classification - Breast Cancer dataset

In the next figures are the results of a log-linear classifier with $\ell_1$ e $\ell_2$ regularization, and different values for the parameter `C`. For the $\ell_2$ regularization, several weights have small values but are rarely equal to zero. For the $\ell_1$ regularization, several weights are set to zero, which can be used to discard superfluous dimensions and find the most discriminative ones.




$\ell_1$ ($|w| \leq 10$)



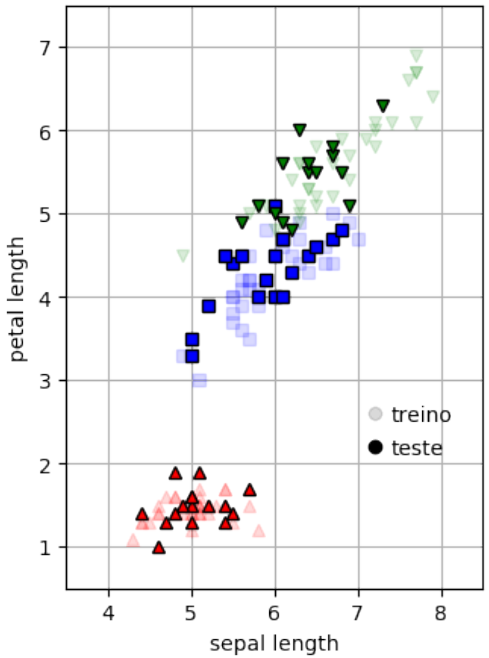

$\ell_2$

# Log-Linear Classifiers - `sklearn`

## Example: Multi-Class Classification - Iris dataset

Four-dimensional data divided in three classes.

```
>>> 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)
```



# Log-Linear Classifiers - `sklearn`

## Example: Multi-Class Classification - Iris dataset

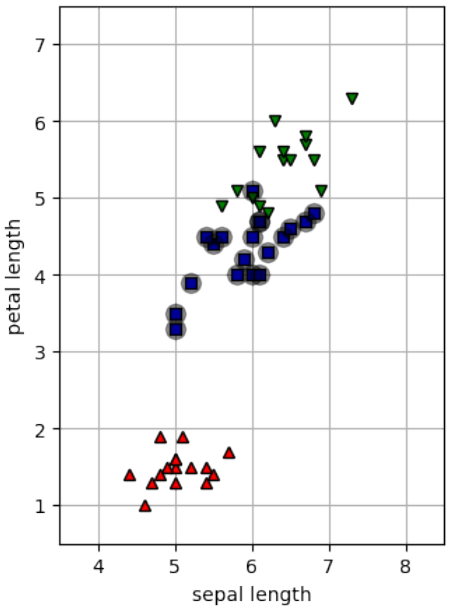
In the following figures are the results obtained with a log-linear classifier with $\ell_1$ e $\ell_2$ regularization, `liblinear` optimization (one versus the rest binary classification), and with `C`= 0.1.

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

or

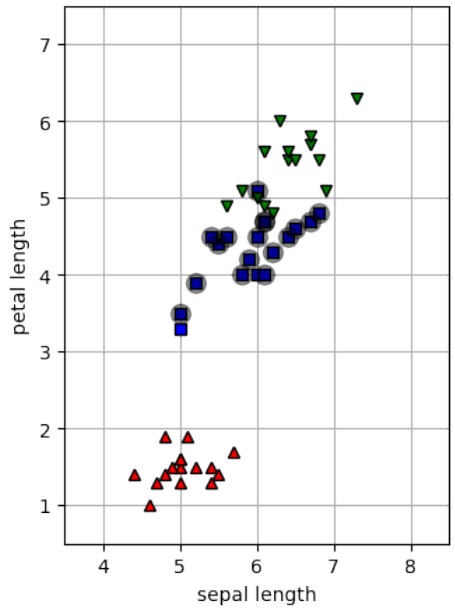
```
>>> Dlog=LogisticRegression.(solver='liblinear',C=0.1,penalty='l2')
```

$\ell_1$




19 errors in 50

$\ell_2$




17 errors

`Dlog.coef_`$= \begin{bmatrix} 0 & 0.88 & -1.09 & 0 \\ 0 & 0.25 & 0 & 0 \\ -0.73 & 0 & 0.97 & 0 \end{bmatrix}$

$\begin{bmatrix} 0.19 & 0.66 & -1.08 & -0.5 \\ -0.02 & -0.5 & 0.23 & -0.14 \\ -0.53 & -0.48 & 0.83 & 0.62 \end{bmatrix}$

# Log-Linear Classifiers - sklearn

## Example: Multi-Class Classification - Iris dataset

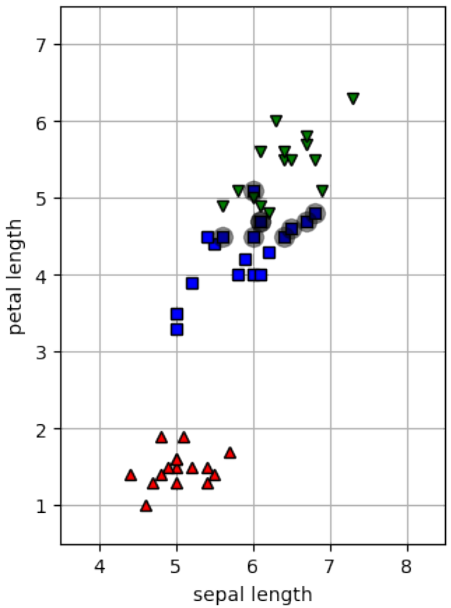
Log-linear classifier results trained in binary versus multi-class mode. In both cases, the same regularization and optimization was used.

`newton-cg` optimization, $\ell_2$ regularization, and with `C=` 0.1.

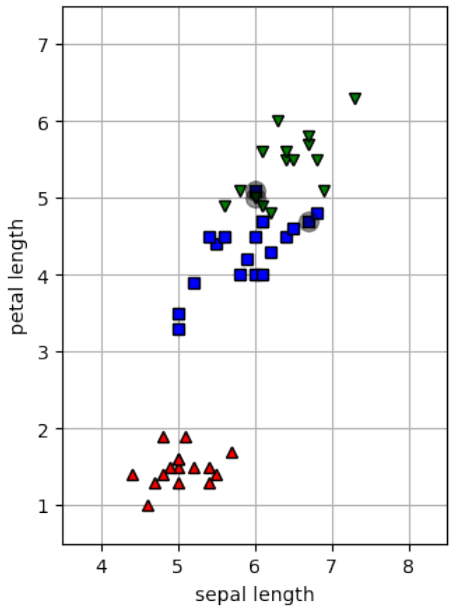
```
>>> Dlog=LogisticRegression(solver='newton-cg',C=0.1,penalty='l2',multi_class='ovr')
```

versus

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




binary: 9 errors in 50




multi-class: 3 errors

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

# Log-Linear Classifiers - `sklearn`

## Example: Multi-Class Classification - Iris dataset

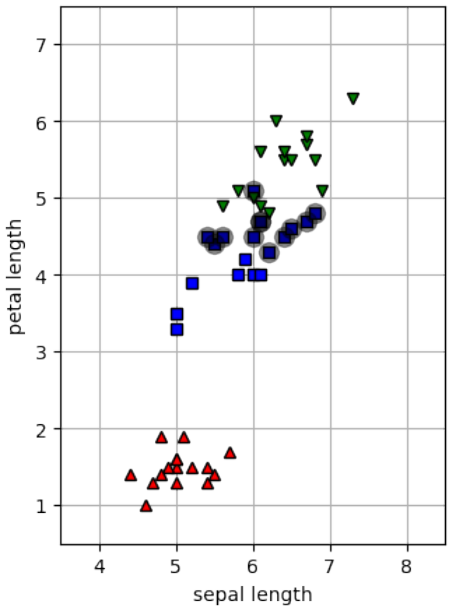
Log-linear classifier results trained in binary versus multi-class mode. In both cases, the same regularization and optimization was used.

`saga` optimization, $\ell_1$ regularization, and with `C`= 0.1.

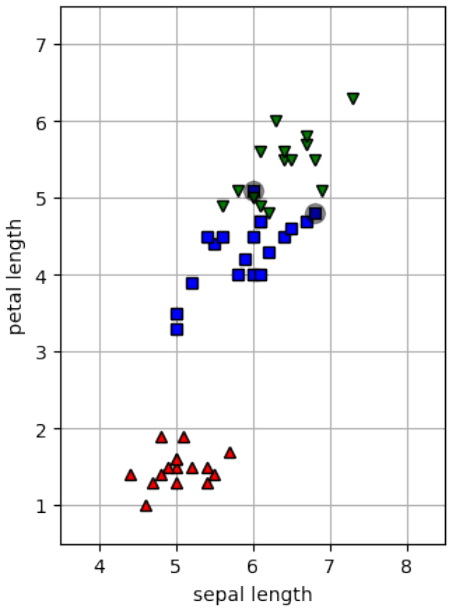
```
>>> Dlog=LogisticRegression(solver='saga',C=0.1,penalty='l1',multi_class='ovr')
```

versus

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




binary: 12 errors in 50




multi-class: 2 errors

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

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