RegresionLogistica_professor

October 29, 2020

1 Logistic Regression

```
Notebook version: 2.0 (Nov 21, 2017)
                  2.1 (Oct 19, 2018)
                  2.2 (Oct 09, 2019)
                  2.3 (Oct 27, 2020)

Author: Jesús Cid Sueiro (jcid@tsc.uc3m.es)
        Jerónimo Arenas García (jarenas@tsc.uc3m.es)

Changes: v.1.0 - First version
         v.1.1 - Typo correction. Prepared for slide presentation
         v.2.0 - Prepared for Python 3.0 (backcompmatible with 2.7)
                 Assumptions for regression model modified
         v.2.1 - Minor changes regarding notation and assumptions
         v.2.2 - Updated notation
         v.2.3 - Improved slides format. Backward compatibility removed
```

[1]:
```
# To visualize plots in the notebook
%matplotlib inline

# Imported libraries
import csv
import random
import matplotlib
import matplotlib.pyplot as plt
import pylab

import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from sklearn.preprocessing import PolynomialFeatures
from sklearn import linear_model
```

1.1 1. Introduction

1.1.1 1.1. Binary classification

The **goal** of a classification problem is to assign a *class* or *category* to every *instance* or *observation* of a data collection.

Here, we will assume that

- every instance $\mathbf{x}$ is an N-dimensional vector in $\mathbb{R}^N$, and
- the class y of sample $\mathbf{x}$ is an element of a binary set $\mathcal{Y} = \{0, 1\}$.

The goal of a classifier is to predict the true value of y after observing $\mathbf{x}$.

We will denote as $\hat{y}$ the classifier output or **decision**. If $y = \hat{y}$, the decision is a **hit**, otherwise $y \neq \hat{y}$ and the decision is an **error**.

1.1.2 1.2. Decision theory: the MAP criterion

Decision theory provides a solution to the classification problem in situations where the relation between instance $\mathbf{x}$ and its class y is given by a known probabilistic model.

Assume that every tuple $(\mathbf{x}, y)$ is an outcome of a random vector $(\mathbf{X}, Y)$ with joint distribution $p_{\mathbf{X},Y}(\mathbf{x}, y)$. A natural criteria for classification is to select predictor $\hat{Y} = f(\mathbf{x})$ in such a way that the **probability or error**, $P\{\hat{Y} \neq Y\}$ is minimum.

Noting that

$$P\{\hat{Y} \neq Y\} = \int P\{\hat{Y} \neq Y|\mathbf{x}\} p_{\mathbf{X}}(\mathbf{x}) d\mathbf{x}$$

the optimal decision maker should take, for every sample $\mathbf{x}$, the decision minimizing the **conditional error probability**:

$$\hat{y}^* = \arg\min_{\hat{y}} P\{Y \neq \hat{y}|\mathbf{x}\} \tag{1}$$

$$= \arg\max_{\hat{y}} P\{Y = \hat{y}|\mathbf{x}\} \tag{2}$$

$$\tag{3}$$

Thus, the **optimal decision rule** can be expressed as

$$P_{Y|\mathbf{X}}(1|\mathbf{x}) \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} P_{Y|\mathbf{X}}(0|\mathbf{x})$$

or, equivalently

$$P_{Y|\mathbf{X}}(1|\mathbf{x}) \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} \frac{1}{2}$$

The classifier implementing this decision rule is usually referred to as the MAP (*Maximum A Posteriori*) classifier. As we have seen, the MAP classifier minimizes the error probability for binary classification, but the result can also be generalized to multiclass classification problems.

1.1.3 1.3. Learning

Classical decision theory is grounded on the assumption that the probabilistic model relating the observed sample $\mathbf{X}$ and the true hypothesis Y is known.

Unfortunately, this is unrealistic in many applications, where the only available information to construct the classifier is a **dataset** $\mathcal{D} = \{\mathbf{x}_k, y_k\}_{k=0}^{K-1}$ of instances and their respective class labels.

A more realistic formulation of the classification problem is the following: given a dataset $\mathcal{D} = \{(\mathbf{x}_k, y_k) \in \mathbb{R}^N \times \mathcal{Y},\, k = 0, \ldots, K-1\}$ of independent and identically distributed (i.i.d.) samples from an ***unknown*** distribution $p_{\mathbf{X},Y}(\mathbf{x}, y)$, predict the class y of a new sample $\mathbf{x}$ with the minimum probability of error.

1.1.4 1.4. Parametric classifiers

Since the probabilistic model generating the data is unknown, the MAP decision rule cannot be applied. However, we can use the dataset to **estimate the a posterior class probability model**, and apply it to approximate the MAP decision maker.

Parametric classifiers based on this idea assume, additionally, that the posterior class probabilty satisfies some parametric formula:

$$P_{Y|X}(1|\mathbf{x}, \mathbf{w}) = f_{\mathbf{w}}(\mathbf{x})$$

where $\mathbf{w}$ is a vector of parameters. Given the expression of the MAP decision maker, classification consists in comparing the value of $f_{\mathbf{w}}(\mathbf{x})$ with the threshold $\frac{1}{2}$, and each parameter vector would be associated to a different decision maker.

In practice, the dataset $\mathcal{D}$ is used to select a particular parameter vector $\hat{\mathbf{w}}$ according to certain criterion. Accordingly, the decision rule becomes

$$f_{\hat{\mathbf{w}}}(\mathbf{x}) \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} \frac{1}{2}$$

In this notebook, we explore one of the most popular model-based parametric classification methods: **logistic regression**.

1.2 2. Logistic regression.

1.2.1 2.1. The logistic function

The **logistic regression model** assumes that the binary class label $Y \in \{0, 1\}$ of observation $X \in \mathbb{R}^N$ satisfies the expression.

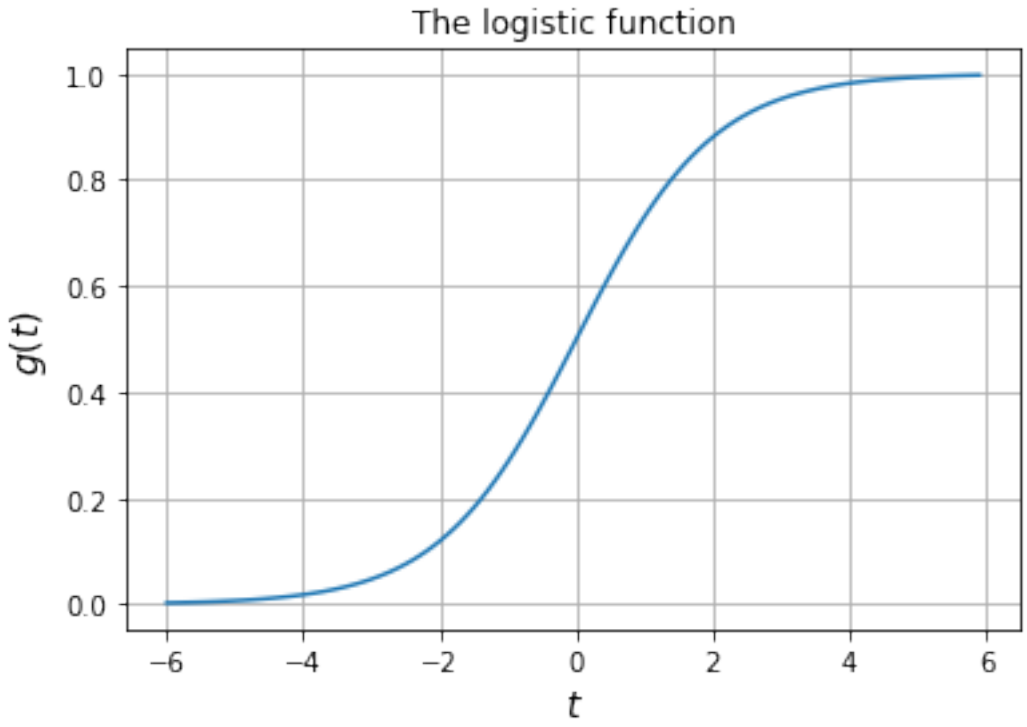
$$P_{Y|\mathbf{X}}(1|\mathbf{x},\mathbf{w}) = g(\mathbf{w}^\intercal\mathbf{x})$$
$$P_{Y|,\mathbf{X}}(0|\mathbf{x},\mathbf{w}) = 1 - g(\mathbf{w}^\intercal\mathbf{x})$$

where $\mathbf{w}$ is a parameter vector and $g(\,\cdot\,)$ is the **logistic** function, which is defined by

$$g(t) = \frac{1}{1+\exp(-t)}$$

In the following we define a logistic function in python, and use it to plot a graphical representation.

[2]:
```
# Define the logistic function
def logistic(t):
    #<SOL>
    return 1.0 / (1 + np.exp(-t))
    #</SOL>

# Plot the logistic function
t = np.arange(-6, 6, 0.1)
z = logistic(t)

plt.plot(t, z)
plt.xlabel('$t$', fontsize=14)
plt.ylabel('$g(t)$', fontsize=14)
plt.title('The logistic function')
plt.grid()
```



It is straightforward to see that the logistic function has the following properties:

- **P1**: Probabilistic output: $0 \leq g(t) \leq 1$
- **P2**: Symmetry: $g(-t) = 1 - g(t)$
- **P3**: Monotonicity: $g'(t) = g(t) \cdot [1 - g(t)] \geq 0$

Exercise 1: Verify properties P2 and P3.

Exercise 2: Implement a function to compute the logistic function, and use it to plot such function in the inverval $[-6, 6]$.

1.2.2 2.2. Classifiers based on the logistic model.

The MAP classifier under a logistic model will have the form

$$P_{Y|\mathbf{X}}(1|\mathbf{x}, \mathbf{w}) = g(\mathbf{w}^\intercal \mathbf{x}) \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} \frac{1}{2}$$

Therefore

$$2 \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} 1 + \exp(-\mathbf{w}^\intercal \mathbf{x})$$

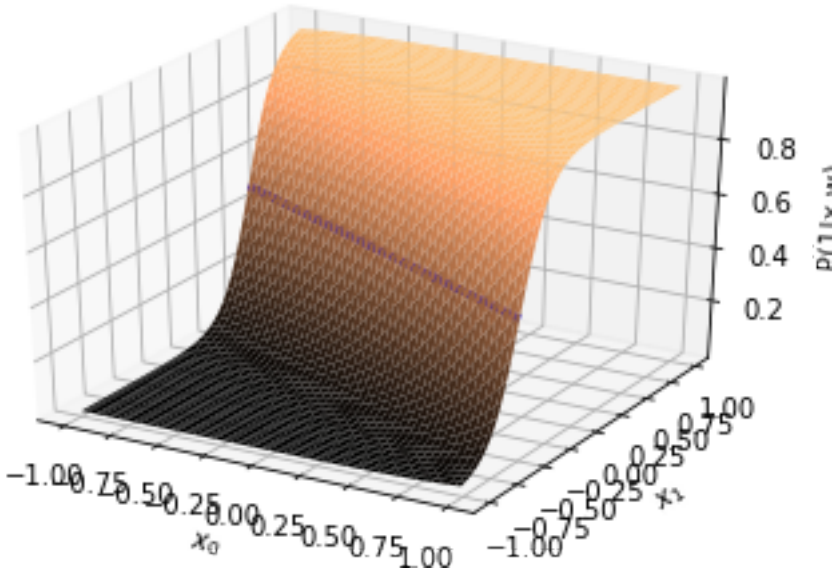
which is equivalent to

$$\mathbf{w}^\intercal \mathbf{x} \underset{\hat{y}=0}{\overset{\hat{y}=1}{\gtrless}} 0$$

Thus, the classifiers based on the logistic model are given by **linear decision boundaries** passing through the origin, $\mathbf{x} = \mathbf{0}$.

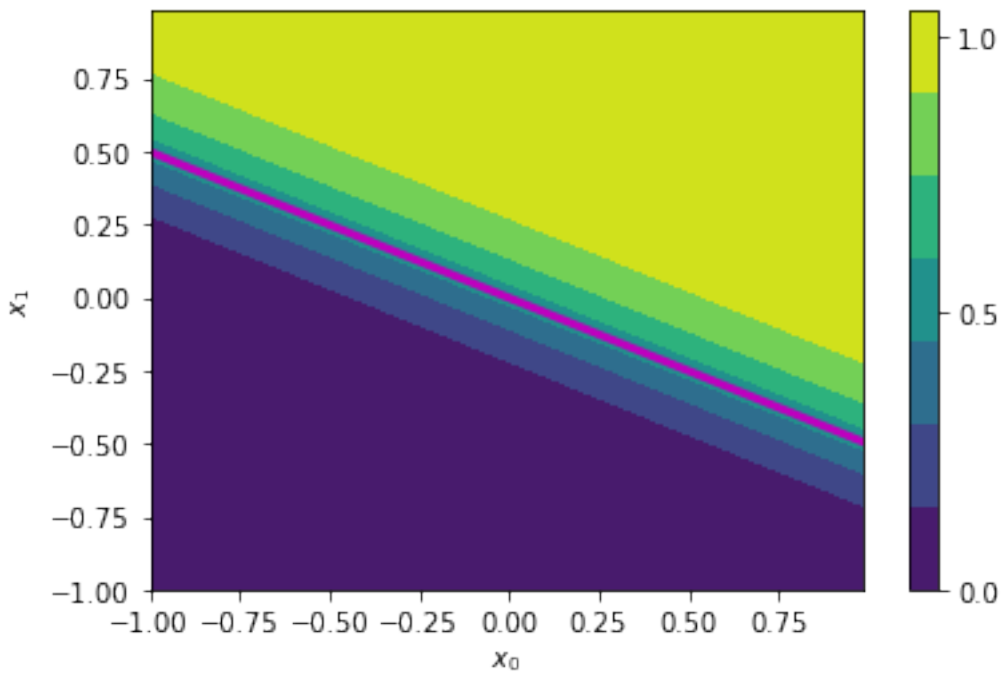
[3]:
```
# Weight vector:
w = [4, 8]   # Try different weights

# Create a rectangular grid.
x_min = -1
x_max = 1
h = (x_max - x_min) / 200
xgrid = np.arange(x_min, x_max, h)
xx0, xx1 = np.meshgrid(xgrid, xgrid)

# Compute the logistic map for the given weights, and plot
Z = logistic(w[0]*xx0 + w[1]*xx1)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(xx0, xx1, Z, cmap=plt.cm.copper)
```

```
ax.contour(xx0, xx1, Z, levels=[0.5], colors='b', linewidths=(3,))
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
ax.set_zlabel('P(1|x,w)')
plt.show()
```



The next code fragment represents the output of the same classifier, representing the output of the logistic function in the x_0-x_1 plane, encoding the value of the logistic function in the color map.

[4]:
```
CS = plt.contourf(xx0, xx1, Z)
CS2 = plt.contour(CS, levels=[0.5], colors='m', linewidths=(3,))
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.colorbar(CS, ticks=[0, 0.5, 1])
plt.show()
```



1.2.3 3.3. Nonlinear classifiers.

The logistic model can be extended to construct non-linear classifiers by using **non-linear data transformations**. A general form for a nonlinear logistic regression model is

$$P_{Y|\mathbf{X}}(1|\mathbf{x}, \mathbf{w}) = g[\mathbf{w}^\intercal \mathbf{z}(\mathbf{x})]$$

where $\mathbf{z}(\mathbf{x})$ is an arbitrary nonlinear transformation of the original variables. The boundary decision in that case is given by equation

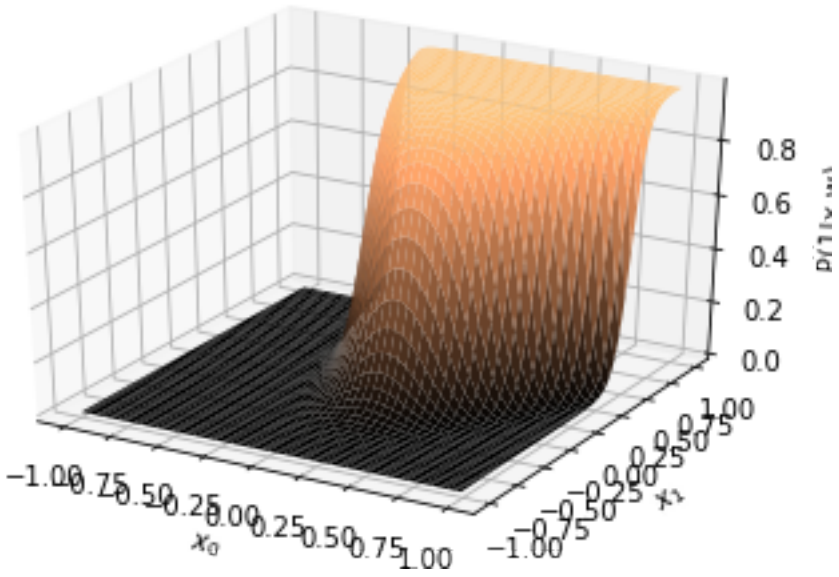
$$\mathbf{w}^\intercal \mathbf{z} = 0$$

Exercise 3: Modify the code above to generate a 3D surface plot of the polynomial logistic regression model given by

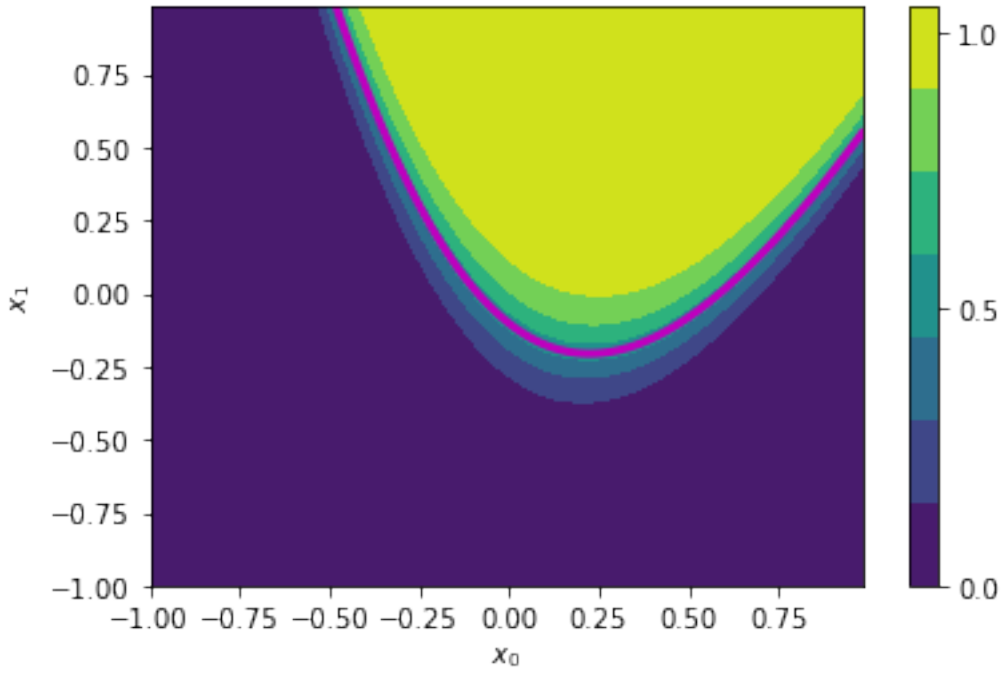
$$P_{Y|\mathbf{X}}(1|\mathbf{x}, \mathbf{w}) = g(1 + 10x_0 + 10x_1 - 20x_0^2 + 5x_0x_1 + x_1^2)$$

[5]:
```
# Weight vector:
w = [1, 10, 10, -20, 5, 1]   # Try different weights

# Create a regtangular grid.
x_min = -1
x_max = 1
h = (x_max - x_min) / 200
```

```
xgrid = np.arange(x_min, x_max, h)
xx0, xx1 = np.meshgrid(xgrid, xgrid)

# Compute the logistic map for the given weights
# Z = <FILL IN>
Z = logistic(w[0] + w[1]*xx0 + w[2]*xx1 + w[3]*xx0*xx0 + w[4]*xx0*xx1 +␣
 ↪w[5]*xx1*xx1)
# Plot the logistic map
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(xx0, xx1, Z, cmap=plt.cm.copper)
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
ax.set_zlabel('P(1|x,w)')
plt.show()
```



[6]:
```
CS = plt.contourf(xx0, xx1, Z)
CS2 = plt.contour(CS, levels=[0.5], colors='m', linewidths=(3,))
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.colorbar(CS, ticks=[0, 0.5, 1])
plt.show()
```



1.3 3. Inference

Remember that the idea of parametric classification is to use the training data set $\mathcal{D} = \{(\mathbf{x}_k, y_k) \in \mathbb{R}^N \times \{0, 1\}, k = 0, \ldots, K-1\}$ to estimate $\mathbf{w}$. The estimate, $\hat{\mathbf{w}}$, can be used to compute the label prediction for any new observation as

$$\hat{y} = \arg\max_y P_{Y|\mathbf{X}}(y|\mathbf{x}, \hat{\mathbf{w}}).$$

In this notebook, we will discuss two different approaches to the estimation of $\mathbf{w}$:

- **Maximum Likelihood** (ML): $\hat{\mathbf{w}}_{\text{ML}} = \arg\max_{\mathbf{w}} P_{\mathcal{D}|\mathbf{W}}(\mathcal{D}|\mathbf{w})$
- **Maximum *A Posteriori** (MAP): $\hat{\mathbf{w}}_{\text{MAP}} = \arg\max_{\mathbf{w}} p_{\mathbf{W}|\mathcal{D}}(\mathbf{w}|\mathcal{D})$

For the mathematical derivation of the logistic regression algorithm, the following representation of the logistic model will be useful: using the **symmetry** property of the logistic function, we can write

$$P_{Y|\mathbf{X}}(0|\mathbf{x}, \mathbf{w}) = 1 - g\left(\mathbf{w}^\intercal \mathbf{z}(\mathbf{x})\right) = g\left(-\mathbf{w}^\intercal \mathbf{z}(\mathbf{x})\right)$$

thus

$$P_{Y|\mathbf{X}}(y|\mathbf{x}, \mathbf{w}) = g\left(\overline{y}\mathbf{w}^\intercal \mathbf{z}(\mathbf{x})\right)$$

where $\overline{y} = 2y - 1$ is a **symmetrized label** ($\overline{y} \in \{-1, 1\}$).

1.3.1 3.1. Model assumptions

In the following, we will make the following assumptions:

- **A1**. (Logistic Regression): We assume a logistic model for the *a posteriori* probability of Y given $\mathbf{X}$, i.e.,

$$P_{Y|\mathbf{X}}(y|\mathbf{x},\mathbf{w}) = g\left(\bar{y}\cdot\mathbf{w}^\intercal\mathbf{z}(\mathbf{x})\right).$$

- **A2**. All samples in $\mathcal{D}$ have been generated from the same distribution, $p_{\mathbf{X},Y|\mathbf{W}}(\mathbf{x},y|\mathbf{w})$.
- **A3**. Input variables $\mathbf{x}$ do not depend on $\mathbf{w}$. This implies that $p(\mathbf{x}|\mathbf{w}) = p(\mathbf{x})$
- **A4**. Targets $y_0,\cdots,y_{K-1}$ are statistically independent given $\mathbf{w}$ and the inputs $\mathbf{x}_0,\cdots,\mathbf{x}_{K-1}$, that is:

$$P(y_0,\cdots,y_{K-1}|\mathbf{x}_0,\cdots,\mathbf{x}_{K-1},\mathbf{w}) = \prod_{k=0}^{K-1} P(y_k|\mathbf{x}_k,\mathbf{w})$$

1.3.2 3.2. ML estimation.

The ML estimate is defined as

$$\hat{\mathbf{w}}_{\text{ML}} = \arg\max_{\mathbf{w}} P_{\mathcal{D}|\mathbf{W}}(\mathcal{D}|\mathbf{w})$$

Ussing assumptions A2 and A3 above, we have that

$$\begin{aligned} P_{\mathcal{D}|\mathbf{W}}(\mathcal{D}|\mathbf{w}) &= p(y_0,\cdots,y_{K-1},\mathbf{x}_0,\cdots,\mathbf{x}_{K-1}|\mathbf{w}) && (4)\\ &= P(y_0,\cdots,y_{K-1}|\mathbf{x}_0,\cdots,\mathbf{x}_{K-1},\mathbf{w})\; p(\mathbf{x}_0,\cdots,\mathbf{x}_{K-1}|\mathbf{w}) && (5)\\ &= P(y_0,\cdots,y_{K-1}|\mathbf{x}_0,\cdots,\mathbf{x}_{K-1},\mathbf{w})\; p(\mathbf{x}_0,\cdots,\mathbf{x}_{K-1}) && (6)\end{aligned}$$

Finally, using assumption A4, we can formulate the ML estimation of $\mathbf{w}$ as the resolution of the following **optimization problem**

$$\begin{aligned} \hat{\mathbf{w}}_{\text{ML}} &= \arg\max_{\mathbf{w}} P(y_0,\cdots,y_{K-1}|\mathbf{x}_0,\cdots,\mathbf{x}_{K-1},\mathbf{w}) && (7)\\ &= \arg\max_{\mathbf{w}} \prod_{k=0}^{K-1} P(y_k|\mathbf{x}_k,\mathbf{w}) && (8)\\ &= \arg\max_{\mathbf{w}} \sum_{k=0}^{K-1} \log P(y_k|\mathbf{x}_k,\mathbf{w}) && (9)\\ &= \arg\min_{\mathbf{w}} \sum_{k=0}^{K-1} -\log P(y_k|\mathbf{x}_k,\mathbf{w}) && (10)\end{aligned}$$

where the arguments of the maximization or minimization problems of the last three lines are usually referred to as the **likelihood**, **log-likelihood** [$L(\mathbf{w})$], and **negative log-likelihood** [$\mathrm{NLL}(\mathbf{w})$], respectively.

Now, using A1 (the logistic model)

$$\mathrm{NLL}(\mathbf{w}) = -\sum_{k=0}^{K-1} \log\left[g\left(\overline{y}_k \mathbf{w}^\intercal \mathbf{z}_k\right)\right] \tag{11}$$

$$= \sum_{k=0}^{K-1} \log\left[1 + \exp\left(-\overline{y}_k \mathbf{w}^\intercal \mathbf{z}_k\right)\right] \tag{12}$$

where $\mathbf{z}_k = \mathbf{z}(\mathbf{x}_k)$.

It can be shown that $\mathrm{NLL}(\mathbf{w})$ is a **convex** and **differentiable** function of $\mathbf{w}$. Therefore, its minimum is a point with zero gradient.

$$\nabla_{\mathbf{w}} \mathrm{NLL}(\hat{\mathbf{w}}_{\mathrm{ML}}) = -\sum_{k=0}^{K-1} \frac{\exp\left(-\overline{y}_k \hat{\mathbf{w}}_{\mathrm{ML}}^\intercal \mathbf{z}_k\right) \overline{y}_k \mathbf{z}_k}{1 + \exp\left(-\overline{y}_k \hat{\mathbf{w}}_{\mathrm{ML}}^\intercal \mathbf{z}_k\right)} = \tag{13}$$

$$= -\sum_{k=0}^{K-1} \left[y_k - g(\hat{\mathbf{w}}_{\mathrm{ML}}^T \mathbf{z}_k)\right] \mathbf{z}_k = 0 \tag{14}$$

Unfortunately, $\hat{\mathbf{w}}_{\mathrm{ML}}$ cannot be taken out from the above equation, and some iterative optimization algorithm must be used to search for the minimum.

1.3.3 3.3. Gradient descent.

A simple iterative optimization algorithm is gradient descent.

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \rho_n \nabla_{\mathbf{w}} \mathrm{NLL}(\mathbf{w}_n) \tag{15}$$

where $\rho_n > 0$ is the *learning step*.

Applying the gradient descent rule to logistic regression, we get the following algorithm:

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \rho_n \sum_{k=0}^{K-1} \left[y_k - g(\mathbf{w}_n^\intercal \mathbf{z}_k)\right] \mathbf{z}_k \tag{16}$$

Gradient descent in matrix form Defining vectors

$$\mathbf{y} = [y_0, \ldots, y_{K-1}]^\top \tag{17}$$

$$\hat{\mathbf{p}}_n = [g(\mathbf{w}_n^\top \mathbf{z}_0), \ldots, g(\mathbf{w}_n^\top \mathbf{z}_{K-1})]^\top \tag{18}$$

and matrix

$$\mathbf{Z} = [\mathbf{z}_0, \ldots, \mathbf{z}_{K-1}]^\top \tag{19}$$

we can write

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \rho_n \mathbf{Z}^\top (\mathbf{y} - \hat{\mathbf{p}}_n) \tag{20}$$

In the following, we will explore the behavior of the gradient descend method using the Iris Dataset.

[7]:
```
# Adapted from a notebook by Jason Brownlee
def loadDataset(filename, split):
    xTrain, cTrain, xTest, cTest = [], [], [], []

    with open(filename, 'r') as csvfile:
        lines = csv.reader(csvfile)
        dataset = list(lines)
    for i in range(len(dataset)-1):
        for y in range(4):
            dataset[i][y] = float(dataset[i][y])
        item = dataset[i]
        if random.random() < split:
            xTrain.append(item[0:4])
            cTrain.append(item[4])
        else:
            xTest.append(item[0:4])
            cTest.append(item[4])
    return xTrain, cTrain, xTest, cTest

xTrain_all, cTrain_all, xTest_all, cTest_all = loadDataset('iris.data', 0.66)
nTrain_all = len(xTrain_all)
nTest_all = len(xTest_all)
print('Train:', nTrain_all)
print('Test:', nTest_all)
```

```
Train: 98
Test: 52
```

Now, we select two classes and two attributes.

[8]:
```
# Select attributes
i = 0 # Try 0,1,2,3
j = 1 # Try 0,1,2,3 with j!=i

# Select two classes
c0 = 'Iris-versicolor'
c1 = 'Iris-virginica'
# Select two coordinates
ind = [i, j]

# Take training test
X_tr = np.array([[xTrain_all[n][i] for i in ind] for n in range(nTrain_all)
                 if cTrain_all[n]==c0 or cTrain_all[n]==c1])
C_tr = [cTrain_all[n] for n in range(nTrain_all)
          if cTrain_all[n]==c0 or cTrain_all[n]==c1]
Y_tr = np.array([int(c==c1) for c in C_tr])
n_tr = len(X_tr)

# Take test set
X_tst = np.array([[xTest_all[n][i] for i in ind] for n in range(nTest_all)
                 if cTest_all[n]==c0 or cTest_all[n]==c1])
C_tst = [cTest_all[n] for n in range(nTest_all)
         if cTest_all[n]==c0 or cTest_all[n]==c1]
Y_tst = np.array([int(c==c1) for c in C_tst])
n_tst = len(X_tst)
```

3.2.2. Data normalization Normalization of data is a common pre-processing step in many machine learning algorithms. Its goal is to get a dataset where all input coordinates have a similar scale. Learning algorithms usually show less instabilities and convergence problems when data are normalized.

We will define a normalization function that returns a training data matrix with zero sample mean and unit sample variance.

[9]:
```
def normalize(X, mx=None, sx=None):
    # Compute means and standard deviations
    if mx is None:
        mx = np.mean(X, axis=0)
    if sx is None:
        sx = np.std(X, axis=0)

    # Normalize
    X0 = (X-mx)/sx

    return X0, mx, sx
```

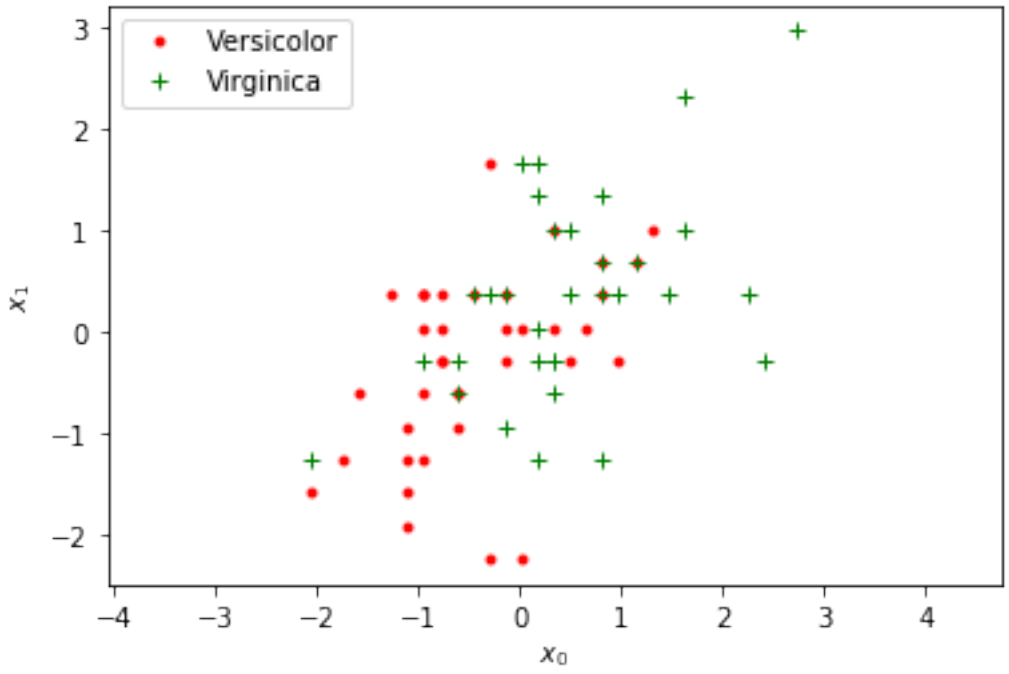
Now, we can normalize training and test data. Observe in the code that **the same transformation**

should be applied to training and test data. This is the reason why normalization with the test data is done using the means and the variances computed with the training set.

[10]:
```
# Normalize data
Xn_tr, mx, sx = normalize(X_tr)
Xn_tst, mx, sx = normalize(X_tst, mx, sx)
```

The following figure generates a plot of the normalized training data.

[11]:
```
# Separate components of x into different arrays (just for the plots)
x0c0 = [Xn_tr[n][0] for n in range(n_tr) if Y_tr[n]==0]
x1c0 = [Xn_tr[n][1] for n in range(n_tr) if Y_tr[n]==0]
x0c1 = [Xn_tr[n][0] for n in range(n_tr) if Y_tr[n]==1]
x1c1 = [Xn_tr[n][1] for n in range(n_tr) if Y_tr[n]==1]

# Scatterplot.
labels = {'Iris-setosa': 'Setosa', 'Iris-versicolor': 'Versicolor',
          'Iris-virginica': 'Virginica'}
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.legend(loc='best')
plt.axis('equal')
plt.show()
```



In order to apply the gradient descent rule, we need to define two methods: - A `fit` method, that receives the training data and returns the model weights and the value of the negative log-likelihood during all iterations. - A `predict` method, that receives the model weight and a set of inputs, and returns the posterior class probabilities for that input, as well as their corresponding class predictions.

[12]:
```
def logregFit(Z_tr, Y_tr, rho, n_it):

    # Data dimension
    n_dim = Z_tr.shape[1]
    # Initialize variables
    nll_tr = np.zeros(n_it)
    pe_tr = np.zeros(n_it)
    Y_tr2 = 2*Y_tr - 1     # Transform labels into binary symmetric.
    w = np.random.randn(n_dim,1)

    # Running the gradient descent algorithm
    for n in range(n_it):
        # Compute posterior probabilities for weight w
        p1_tr = logistic(np.dot(Z_tr, w))
        # Compute negative log-likelihood
        # (note that this is not required for the weight update, only for nll␣
↪tracking)
        nll_tr[n] = np.sum(np.log(1 + np.exp(-np.dot(Y_tr2*Z_tr, w))))
        # Update weights
        w += rho*np.dot(Z_tr.T, Y_tr - p1_tr)

    return w, nll_tr

def logregPredict(Z, w):
    # Compute posterior probability of class 1 for weights w.
    p = logistic(np.dot(Z, w)).flatten()
    # Class
    D = [int(round(pn)) for pn in p]

    return p, D
```

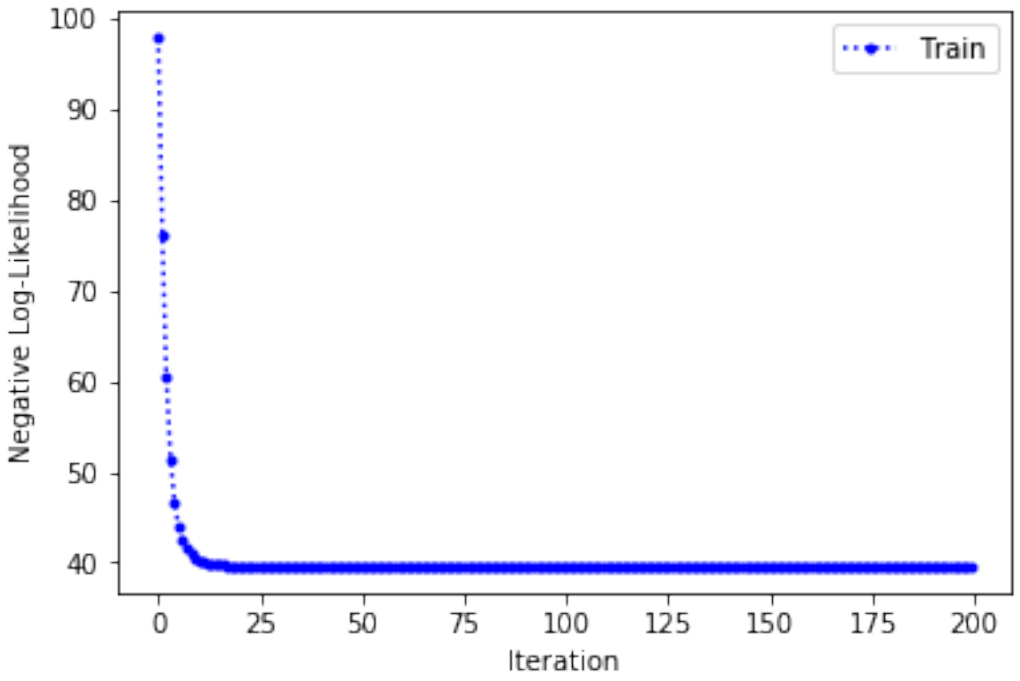
We can test the behavior of the gradient descent method by fitting a logistic regression model with $\mathbf{z}(\mathbf{x}) = (1, \mathbf{x}^\top)^\top$.

[13]:
```
# Parameters of the algorithms
rho = float(1)/50    # Learning step
n_it = 200   # Number of iterations

# Compute Z's
Z_tr = np.c_[np.ones(n_tr), Xn_tr]
Z_tst = np.c_[np.ones(n_tst), Xn_tst]
n_dim = Z_tr.shape[1]
```

```
# Convert target arrays to column vectors
Y_tr2 = Y_tr[np.newaxis].T
Y_tst2 = Y_tst[np.newaxis].T

# Running the gradient descent algorithm
w, nll_tr = logregFit(Z_tr, Y_tr2, rho, n_it)

# Classify training and test data
p_tr, D_tr = logregPredict(Z_tr, w)
p_tst, D_tst = logregPredict(Z_tst, w)

# Compute error rates
E_tr = D_tr!=Y_tr
E_tst = D_tst!=Y_tst

# Error rates
pe_tr = float(sum(E_tr)) / n_tr
pe_tst = float(sum(E_tst)) / n_tst
```

[14]:
```
# NLL plot.
plt.plot(range(n_it), nll_tr,'b.:', label='Train')
plt.xlabel('Iteration')
plt.ylabel('Negative Log-Likelihood')
plt.legend()

print(f'The optimal weights are: {w}')
print('The final error rates are:')
print(f'- Training: {pe_tr}')
print(f'- Test: {pe_tst}')
print(f'The NLL after training is {nll_tr[len(nll_tr)-1]}')
```

```
The optimal weights are: [[-0.16132146]
 [ 0.98393267]
 [ 0.29041333]]
The final error rates are:
- Training: 0.30434782608695654
- Test: 0.25806451612903225
The NLL after training is 39.55894949397751
```



3.2.3. Free parameters Under certain conditions, the gradient descent method can be shown to converge asymptotically (i.e. as the number of iterations goes to infinity) to the ML estimate of the logistic model. However, in practice, the final estimate of the weights **w** depend on several factors:

- Number of iterations
- Initialization
- Learning step

Exercise 4: Visualize the variability of gradient descent caused by initializations. To do so, fix the number of iterations to 200 and the learning step, and execute the gradient descent 100 times, storing the training error rate of each execution. Plot the histogram of the error rate values.

Note that you can do this exercise with a loop over the 100 executions, including the code in the previous code slide inside the loop, with some proper modifications. To plot a histogram of the values in array `p` with `n`bins, you can use `plt.hist(p, n)`

[ ]:

3.2.3.1. Learning step The learning step, ρ, is a free parameter of the algorithm. Its choice is critical for the convergence of the algorithm. Too large values of ρ make the algorithm diverge. For too small values, the convergence gets very slow and more iterations are required for a good convergence.

Exercise 5: Observe the evolution of the negative log-likelihood with the number of iterations for different values of ρ. It is easy to check that, for large enough ρ, the gradient descent method does not converge. Can you estimate (through manual observation) an approximate value of ρ stating a boundary between convergence and divergence?

[ ]:

Exercise 6: In this exercise we explore the influence of the learning step more sistematically. Use the code in the previouse exercises to compute, for every value of ρ, the average error rate over 100 executions. Plot the average error rate vs. ρ.

Note that you should explore the values of ρ in a logarithmic scale. For instance, you can take $\rho = 1, \frac{1}{10}, \frac{1}{100}, \frac{1}{1000}, \ldots$

[ ]:

In practice, the selection of ρ may be a matter of trial an error. Also there is some theoretical evidence that the learning step should decrease along time up to cero, and the sequence ρ_n should satisfy two conditions: - C1: $\sum_{n=0}^{\infty} \rho_n^2 < \infty$ (decrease slowly) - C2: $\sum_{n=0}^{\infty} \rho_n = \infty$ (but not too slowly)

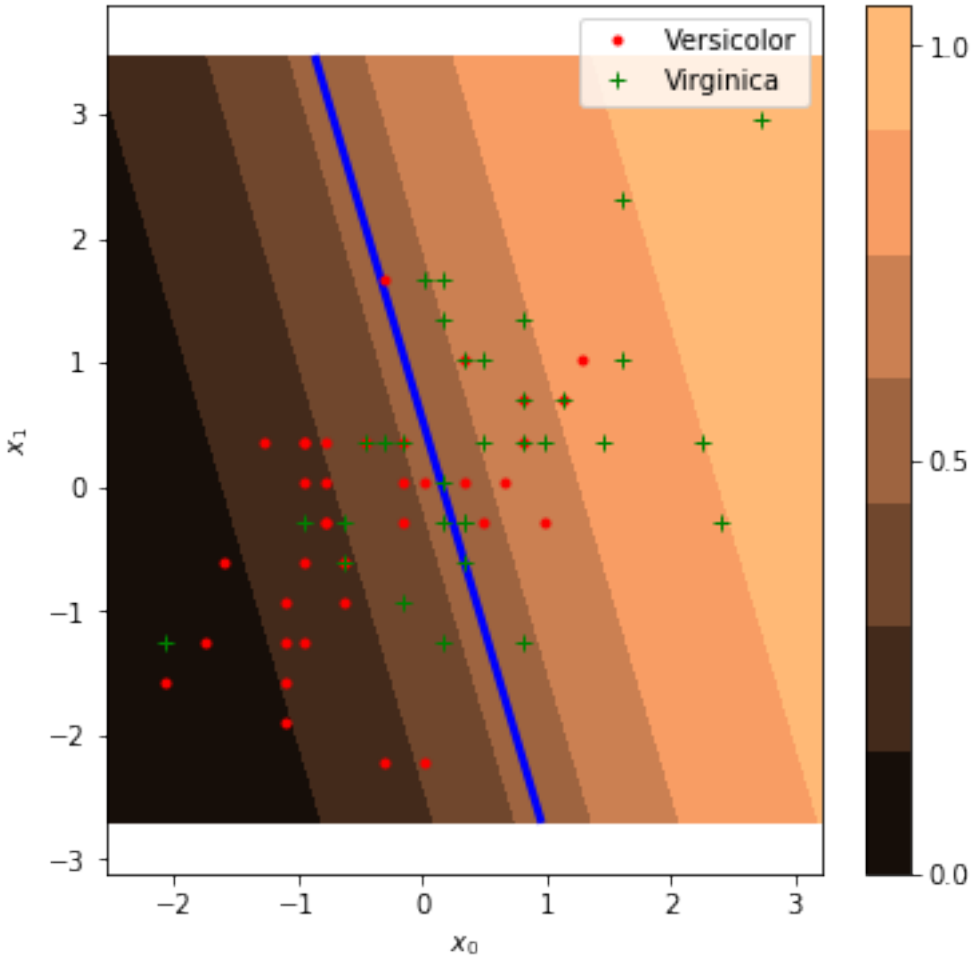
For instance, we can take $\rho_n = \frac{1}{n}$. Another common choice is $\rho_n = \frac{\alpha}{1+\beta n}$ where α and β are also free parameters that can be selected by trial and error with some heuristic method.

3.2.4. Visualizing the posterior map. We can also visualize the posterior probability map estimated by the logistic regression model for the estimated weights.

[15]:
```
# Create a regtangular grid.
x_min, x_max = Xn_tr[:, 0].min(), Xn_tr[:, 0].max()
y_min, y_max = Xn_tr[:, 1].min(), Xn_tr[:, 1].max()
dx = x_max - x_min
dy = y_max - y_min
h = dy /400
xx, yy = np.meshgrid(np.arange(x_min - 0.1 * dx, x_max + 0.1 * dx, h),
                     np.arange(y_min - 0.1 * dx, y_max + 0.1 * dy, h))
X_grid = np.array([xx.ravel(), yy.ravel()]).T

# Compute Z's
Z_grid = np.c_[np.ones(X_grid.shape[0]), X_grid]

# Compute the classifier output for all samples in the grid.
pp, dd = logregPredict(Z_grid, w)
```

[16]:
```
# Paint output maps
pylab.rcParams['figure.figsize'] = 6, 6  # Set figure size
# Color plot
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
```

```
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.legend(loc='best')
plt.axis('equal')
pp = pp.reshape(xx.shape)
CS = plt.contourf(xx, yy, pp, cmap=plt.cm.copper)
plt.contour(xx, yy, pp, levels=[0.5], colors='b', linewidths=(3,))
plt.colorbar(CS, ticks=[0, 0.5, 1])
plt.show()
```



3.2.5. Polynomial Logistic Regression The error rates of the logistic regression model can be potentially reduced by using polynomial transformations.

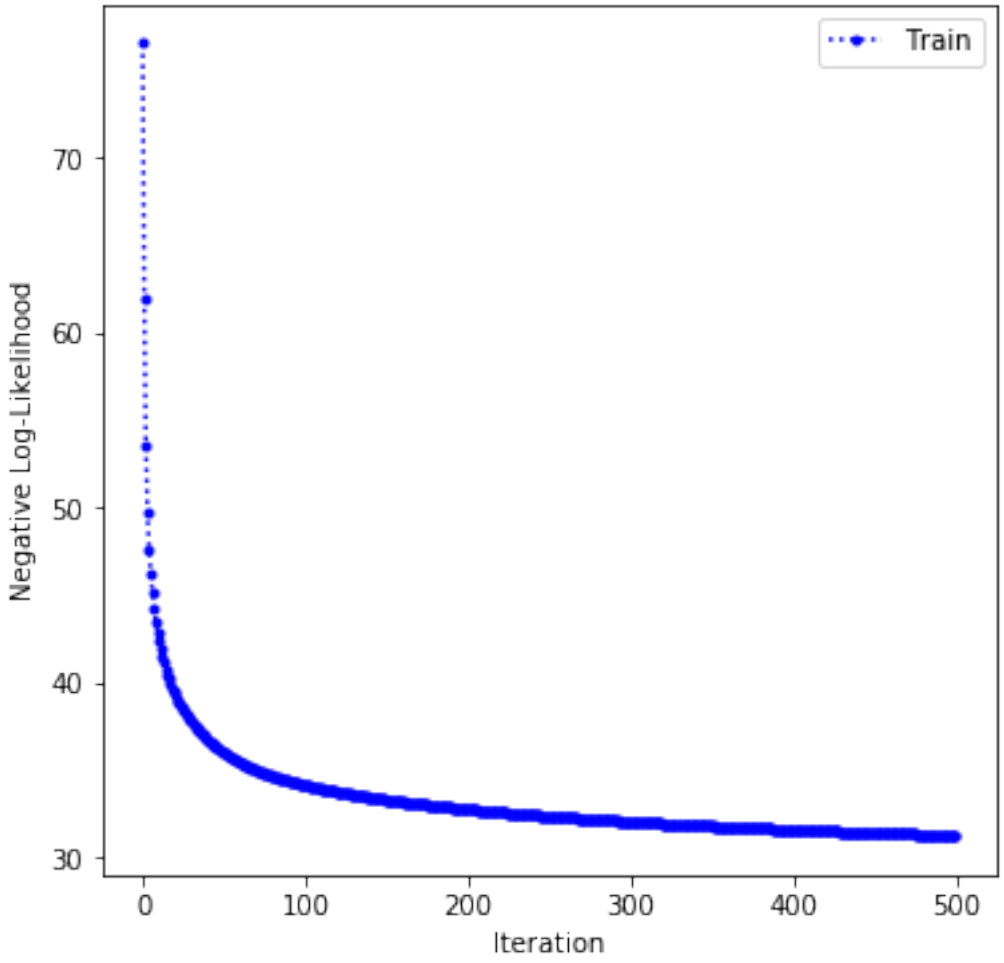
To compute the polynomial transformation up to a given degree, we can use the `PolynomialFeatures` method in `sklearn.preprocessing`.

[17]:
```
# Parameters of the algorithms
rho = float(1)/50    # Learning step
n_it = 500   # Number of iterations
g = 5 # Degree of polynomial

# Compute Z_tr
poly = PolynomialFeatures(degree=g)
Z_tr = poly.fit_transform(Xn_tr)
# Normalize columns (this is useful to make algorithms more stable).)
Zn, mz, sz = normalize(Z_tr[:,1:])
Z_tr = np.concatenate((np.ones((n_tr,1)), Zn), axis=1)

# Compute Z_tst
Z_tst = poly.fit_transform(Xn_tst)
Zn, mz, sz = normalize(Z_tst[:,1:], mz, sz)
Z_tst = np.concatenate((np.ones((n_tst,1)), Zn), axis=1)

# Convert target arrays to column vectors
Y_tr2 = Y_tr[np.newaxis].T
Y_tst2 = Y_tst[np.newaxis].T

# Running the gradient descent algorithm
w, nll_tr = logregFit(Z_tr, Y_tr2, rho, n_it)

# Classify training and test data
p_tr, D_tr = logregPredict(Z_tr, w)
p_tst, D_tst = logregPredict(Z_tst, w)

# Compute error rates
E_tr = D_tr!=Y_tr
E_tst = D_tst!=Y_tst

# Error rates
pe_tr = float(sum(E_tr)) / n_tr
pe_tst = float(sum(E_tst)) / n_tst
```

[18]:
```
# NLL plot.
plt.plot(range(n_it), nll_tr,'b.:', label='Train')
plt.xlabel('Iteration')
plt.ylabel('Negative Log-Likelihood')
plt.legend()
print(f'The optimal weights are: {w.T}')
print('The final error rates are:')
print(f'- Training: {pe_tr} \n- Test: {pe_tst}')
print('The NLL after training is', nll_tr[len(nll_tr)-1])
```

```
The optimal weights are: [[ 8.97889053e-01  1.05134807e+00 -5.19699534e-01
```

```
-2.50966785e+00
  -2.30522385e-01  9.52281126e-01  3.00177884e-03 -1.95873703e+00
   3.87635666e+00  9.63700935e-01  4.91323271e+00  7.69712008e-02
   1.76849820e-02  4.38962909e-01 -1.80335683e+00  3.10220248e-01
  -1.15599652e+00 -1.24876888e-01  1.37205298e+00  2.17191221e+00
   1.18379706e+00]]
The final error rates are:
- Training: 0.2608695652173913
- Test: 0.2903225806451613
The NLL after training is 31.25997997355869
```

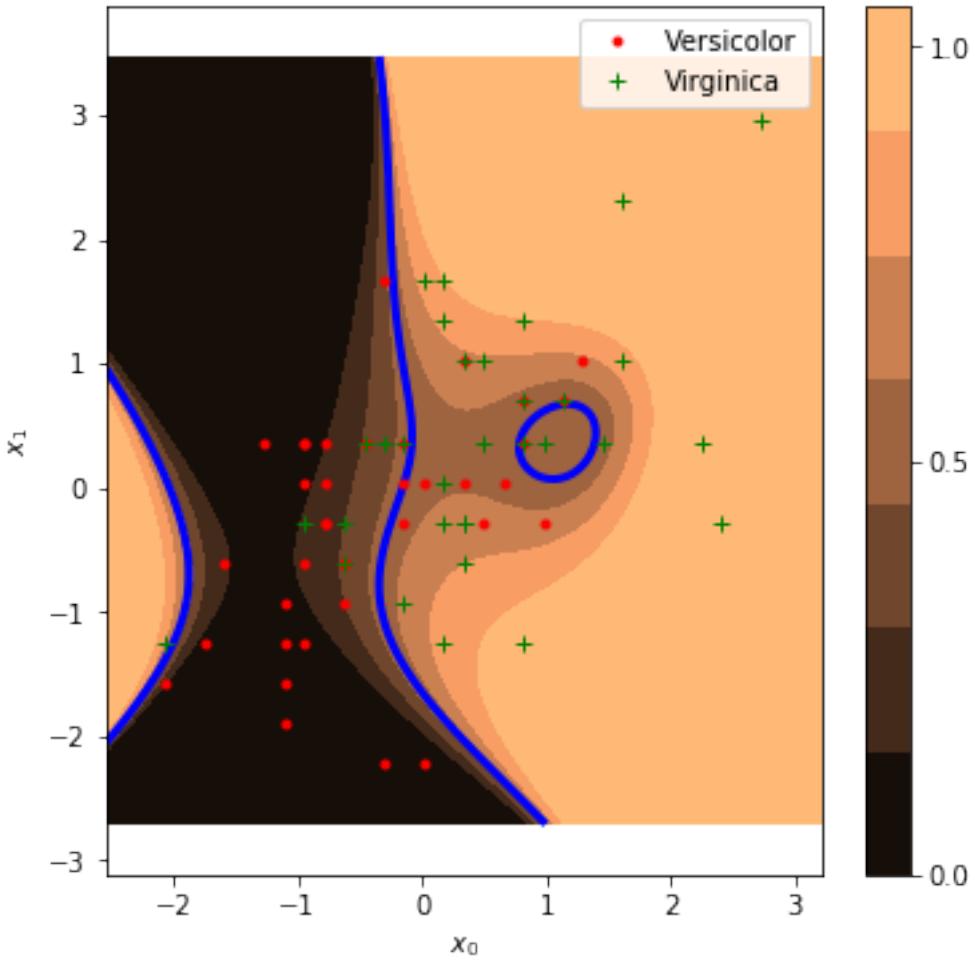


Visualizing the posterior map we can se that the polynomial transformation produces nonlinear decision boundaries.

[19]:
```
# Compute Z_grid
Z_grid = poly.fit_transform(X_grid)
Zn, mz, sz = normalize(Z_grid[:,1:], mz, sz)
```

```
Z_grid = np.concatenate((np.ones((Z_grid.shape[0],1)), Zn), axis=1)

# Compute the classifier output for all samples in the grid.
pp, dd = logregPredict(Z_grid, w)
pp = pp.reshape(xx.shape)
```

[20]:
```
# Paint output maps
pylab.rcParams['figure.figsize'] = 6, 6  # Set figure size
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.axis('equal')
plt.legend(loc='best')
CS = plt.contourf(xx, yy, pp, cmap=plt.cm.copper)
plt.contour(xx, yy, pp, levels=[0.5], colors='b', linewidths=(3,))
plt.colorbar(CS, ticks=[0, 0.5, 1])
plt.show()
```



1.4 4. Regularization and MAP estimation.

1.4.1 4.1 MAP estimation

An alternative to the ML estimation of the weights in logistic regression is Maximum A Posteriori estimation. Modelling the logistic regression weights as a random variable with prior distribution $p_{\mathbf{W}}(\mathbf{w})$, the **MAP estimate** is defined as

$$\hat{\mathbf{w}}_{\text{MAP}} = \arg\max_{\mathbf{w}} p(\mathbf{w}|\mathcal{D})$$

The posterior density $p(\mathbf{w}|\mathcal{D})$ is related to the likelihood function and the prior density of the weights, $p_{\mathbf{W}}(\mathbf{w})$ through the **Bayes rule**

$$p(\mathbf{w}|\mathcal{D}) = \frac{P\left(\mathcal{D}|\mathbf{w}\right)\ p_{\mathbf{W}}(\mathbf{w})}{p\left(\mathcal{D}\right)}$$

In general, the denominator in this expression cannot be computed analytically. However, it is not required for MAP estimation because it does not depend on $\mathbf{w}$. Therefore, the MAP solution is given by

$$\begin{aligned}
\hat{\mathbf{w}}_{\text{MAP}} &= \arg\max_{\mathbf{w}} \left\{P\left(\mathcal{D}|\mathbf{w}\right)\ p_{\mathbf{W}}(\mathbf{w})\right\} && (21)\\
&= \arg\max_{\mathbf{w}} \left\{L(\mathbf{w}) + \log p_{\mathbf{W}}(\mathbf{w})\right\} && (22)\\
&= \arg\min_{\mathbf{w}} \left\{\text{NLL}(\mathbf{w}) - \log p_{\mathbf{W}}(\mathbf{w})\right\} && (23)
\end{aligned}$$

In the light of this expression, we can conclude that the MAP solution is affected by two terms: - The likelihood, which takes large values for parameter vectors $\mathbf{w}$ that fit well the training data (smaller NLL values) - The prior distribution of weights $p_{\mathbf{W}}(\mathbf{w})$, which expresses our *a priori* preference for some solutions.

1.4.2 4.2. Regularization

Even though the prior distribution has a natural interpretation as a model of our knowledge about $p(\mathbf{w})$ before observing the data, its choice is frequenty motivated by the need to avoid data **overfitting**.

Data overfitting is a frequent problem in ML estimation when the dimension of $\mathbf{w}$ is much higher that the dimension of the input $\mathbf{x}$: the ML solution can be too adjusted to the training data, while the test error rate is large.

In practice **we recur to prior distributions that take large values when $\|\mathbf{w}\|$ is small** (associated to smooth classification borders). This helps to improve **generalization**.

In this way, the MAP criterion adds a **penalty term** to the ML objective, that penalizes parameter vectors for which the prior distribution of weights takes small values.

In machine learning, the process of introducing penalty terms to avoid overfitting is usually named **regularization**.

1.4.3 4.3 MAP estimation with Gaussian prior

If we assume that $\mathbf{W}$ follows a **zero-mean Gaussian** random variable with variance matrix $v\mathbf{I}$,

$$p_{\mathbf{W}}(\mathbf{w}) = \frac{1}{(2\pi v)^{N/2}} \exp\left(-\frac{1}{2v}\|\mathbf{w}\|^2\right)$$

the **MAP estimate** becomes

$$\hat{\mathbf{w}}_{\text{MAP}} = \arg\min_{\mathbf{w}} \left\{ \text{NLL}(\mathbf{w}) + \frac{1}{C}\|\mathbf{w}\|^2 \right\} \tag{24}$$

where $C = 2v$. Note that the **regularization term** associated to the prior penalizes parameter vectors with large components. Parameter C controls the regularizatin, and it is named the **inverse regularization strength**.

Noting that

$$\nabla_{\mathbf{w}} \left\{ \text{NLL}(\mathbf{w}) + \frac{1}{C}\|\mathbf{w}\|^2 \right\} = -\mathbf{Z}\left(\mathbf{y} - \hat{\mathbf{p}}_n\right) + \frac{2}{C}\mathbf{w},$$

we obtain the following **gradient descent rule** for MAP estimation

$$\mathbf{w}_{n+1} = \left(1 - \frac{2\rho_n}{C}\right)\mathbf{w}_n + \rho_n \mathbf{Z}\left(\mathbf{y} - \hat{\mathbf{p}}_n\right) \tag{25}$$

Note that the regularization term "pushes" the weights towards zero.

1.4.4 4.4 MAP estimation with Laplacian prior

If we assume that $\mathbf{W}$ follows a multivariate zero-mean Laplacian distribution given by

$$p_{\mathbf{W}}(\mathbf{w}) = \frac{1}{(2C)^N} \exp\left(-\frac{1}{C}\|\mathbf{w}\|_1\right)$$

(where $\|\mathbf{w}\| = |w_1| + \ldots + |w_N|$ is the L_1 norm of $\mathbf{w}$), the MAP estimate becomes

$$\hat{\mathbf{w}}_{\text{MAP}} = \arg\min_{\mathbf{w}} \left\{ \text{NLL}(\mathbf{w}) + \frac{1}{C}\|\mathbf{w}\|_1 \right\} \tag{26}$$

Parameter C is named the *inverse regularization strength*.

Exercise 7: Derive the gradient descent rules for MAP estimation of the logistic regression weights with Laplacian prior.

[ ]:

1.5 5. Other optimization algorithms

1.5.1 5.1. Stochastic Gradient descent.

Stochastic gradient descent (SGD) is based on the idea of using a single sample at each iteration of the learning algorithm. The SGD rule for ML logistic regression is

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \rho_n \mathbf{z}_n \left(y_n - \hat{p}_n\right) \tag{27}$$

Once all samples in the training set have been applied, the algorith can continue by applying the training set several times.

The computational cost of each iteration of SGD is much smaller than that of gradient descent, though it usually needs many more iterations to converge.

Exercise 8: Modify logregFit to implement an algorithm that applies the SGD rule.

1.5.2 5.2. Newton's method

Assume that the function to be minimized, $C(\mathbf{w})$, can be approximated by its **second order Taylor series expansion** around $\mathbf{w}_0$

$$C(\mathbf{w}) \approx C(\mathbf{w}_0) + \nabla_{\mathbf{w}}^\top C(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^\top \mathbf{H}(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0)$$

where $\mathbf{H}(\mathbf{w})$ is the **Hessian matrix** of C at $\mathbf{w}$. Taking the gradient of $C(\mathbf{w})$, and setting the result to $\mathbf{0}$, the minimum of C around $\mathbf{w}_0$ can be approximated as

$$\mathbf{w}^* = \mathbf{w}_0 - \mathbf{H}(\mathbf{w}_0)^{-1} \nabla_{\mathbf{w}}^\top C(\mathbf{w}_0)$$

Since the second order polynomial is only an approximation to C, $\mathbf{w}^*$ is only an approximation to the optimal weight vector, but we can expect $\mathbf{w}^*$ to be closer to the minimizer of C than $\mathbf{w}_0$. Thus, we can repeat the process, computing a second order approximation around $\mathbf{w}^*$ and a new approximation to the minimizer.

Newton's method is based on this idea. At each optimization step, the function to be minimized is approximated by a second order approximation using a Taylor series expansion around the current estimate. As a result, the learning rule becomes

$$\hat{\mathbf{w}}_{n+1} = \hat{\mathbf{w}}_n - \rho_n \mathbf{H}(\mathbf{w}_n)^{-1} \nabla_{\mathbf{w}} C(\mathbf{w}_n)$$

5.2.1. Example: MAP estimation with Gaussian prior.

For instance, for the MAP estimate with Gaussian prior, the *Hessian* matrix becomes

$$\mathbf{H}(\mathbf{w}) = \frac{2}{C}\mathbf{I} + \sum_{k=0}^{K-1} g(\mathbf{w}^\top \mathbf{z}_k) \left[1 - g(\mathbf{w}^\top \mathbf{z}_k)\right] \mathbf{z}_k \mathbf{z}_k^\top$$

Defining diagonal matrix

$$\mathbf{S}(\mathbf{w}) = \text{diag}\left[g(\mathbf{w}^\top \mathbf{z}_k)\left(1 - g(\mathbf{w}^\top \mathbf{z}_k)\right)\right]$$

the Hessian matrix can be written in more compact form as

$$\mathbf{H}(\mathbf{w}) = \frac{2}{C}\mathbf{I} + \mathbf{Z}^\top \mathbf{S}(\mathbf{w}) \mathbf{Z}$$

Therefore, the Newton's algorithm for logistic regression becomes

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \rho_n \left(\frac{2}{C}\mathbf{I} + \mathbf{Z}^\top \mathbf{S}(\mathbf{w}_n)\mathbf{Z}\right)^{-1} \mathbf{Z}^\top (\mathbf{y} - \hat{\mathbf{p}}_n) \tag{28}$$

Some variants of the Newton method are implemented in the Scikit-learn package.

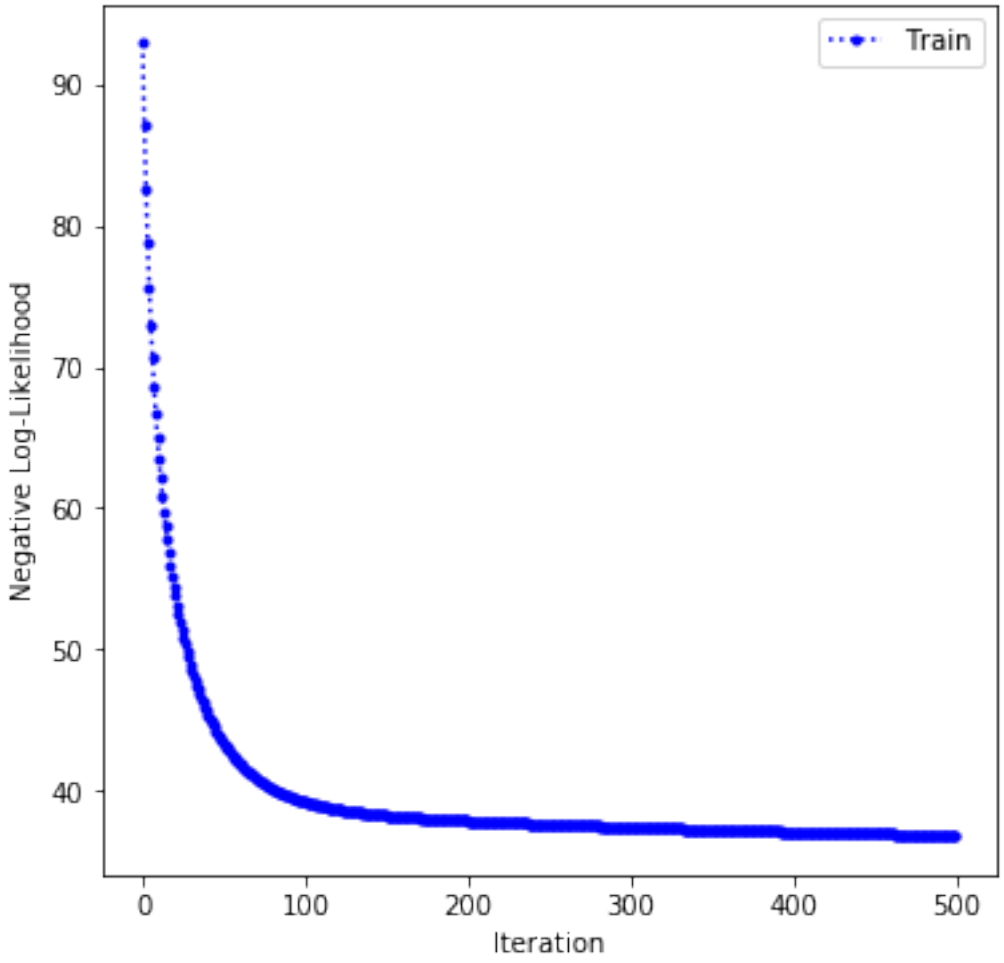
[21]:
```
def logregFit2(Z_tr, Y_tr, rho, n_it, C=1e4):

    # Compute Z's
    r = 2.0/C
    n_dim = Z_tr.shape[1]

    # Initialize variables
    nll_tr = np.zeros(n_it)
    pe_tr = np.zeros(n_it)
    w = np.random.randn(n_dim,1)

    # Running the gradient descent algorithm
    for n in range(n_it):
        p_tr = logistic(np.dot(Z_tr, w))

        sk = np.multiply(p_tr, 1-p_tr)
        S = np.diag(np.ravel(sk.T))

        # Compute negative log-likelihood
```

```
        nll_tr[n] = - np.dot(Y_tr.T, np.log(p_tr)) - np.dot((1-Y_tr).T, np.
↪log(1-p_tr))

        # Update weights
        invH = np.linalg.inv(r*np.identity(n_dim) + np.dot(Z_tr.T, np.dot(S,␣
↪Z_tr)))

        w += rho*np.dot(invH, np.dot(Z_tr.T, Y_tr - p_tr))

    return w, nll_tr
```

[22]:
```
# Parameters of the algorithms
rho = float(1)/50    # Learning step
n_it = 500   # Number of iterations
C = 1000
g = 4

# Compute Z_tr
poly = PolynomialFeatures(degree=g)
Z_tr = poly.fit_transform(X_tr)
# Normalize columns (this is useful to make algorithms more stable).)
Zn, mz, sz = normalize(Z_tr[:,1:])
Z_tr = np.concatenate((np.ones((n_tr,1)), Zn), axis=1)

# Compute Z_tst
Z_tst = poly.fit_transform(X_tst)
Zn, mz, sz = normalize(Z_tst[:,1:], mz, sz)
Z_tst = np.concatenate((np.ones((n_tst,1)), Zn), axis=1)

# Convert target arrays to column vectors
Y_tr2 = Y_tr[np.newaxis].T
Y_tst2 = Y_tst[np.newaxis].T

# Running the gradient descent algorithm
w, nll_tr = logregFit2(Z_tr, Y_tr2, rho, n_it, C)

# Classify training and test data
p_tr, D_tr = logregPredict(Z_tr, w)
p_tst, D_tst = logregPredict(Z_tst, w)

# Compute error rates
E_tr = D_tr!=Y_tr
E_tst = D_tst!=Y_tst

# Error rates
pe_tr = float(sum(E_tr)) / n_tr
pe_tst = float(sum(E_tst)) / n_tst
```

[23]:
```
# NLL plot.
plt.plot(range(n_it), nll_tr,'b.:', label='Train')
plt.xlabel('Iteration')
plt.ylabel('Negative Log-Likelihood')
plt.legend()

print('The final error rates are:')
print('- Training:', str(pe_tr))
print('- Test:', str(pe_tst))
print('The NLL after training is:', str(nll_tr[len(nll_tr)-1]))
```

```
The final error rates are:
- Training: 0.2753623188405797
- Test: 0.25806451612903225
The NLL after training is: 36.826370750198315
```



1.6 6. Logistic regression in Scikit Learn.

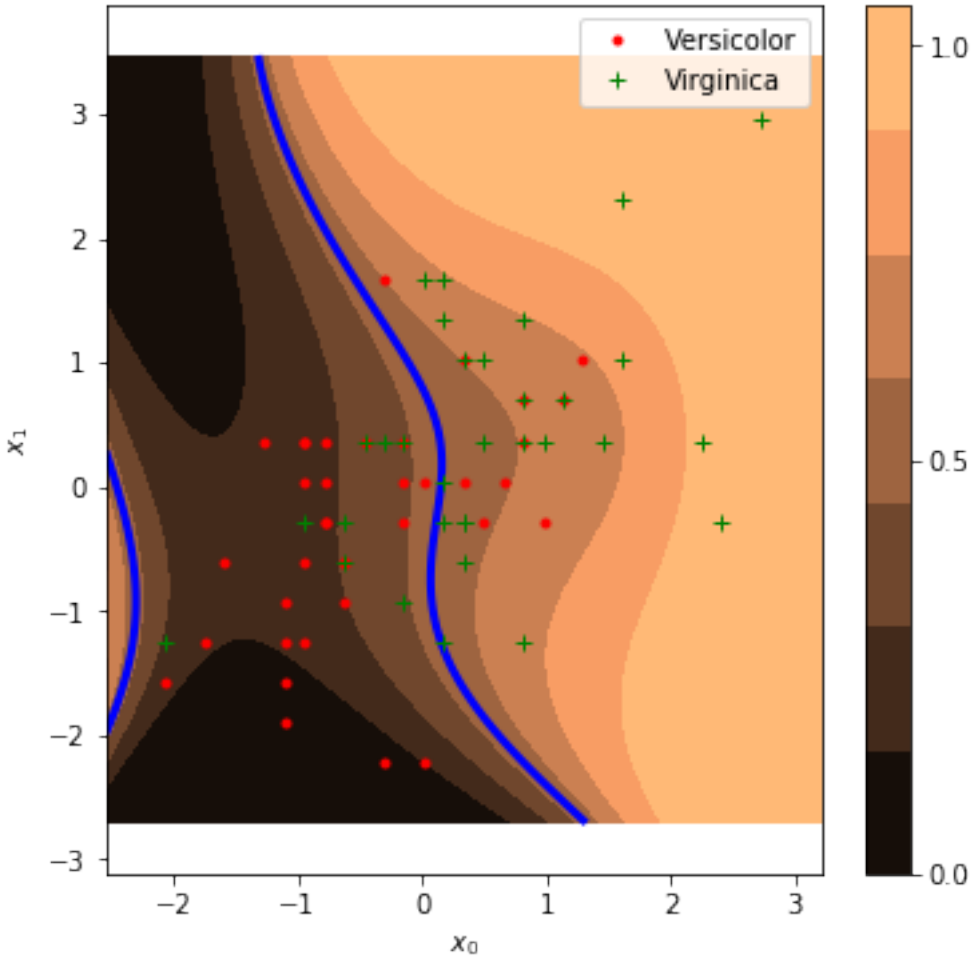
The scikit-learn package includes an efficient implementation of logistic regression. To use it, we must first create a classifier object, specifying the parameters of the logistic regression algorithm.

[24]:
```
# Create a logistic regression object.
LogReg = linear_model.LogisticRegression(C=1.0)

# Compute Z_tr
poly = PolynomialFeatures(degree=g)
Z_tr = poly.fit_transform(Xn_tr)
# Normalize columns (this is useful to make algorithms more stable).)
Zn, mz, sz = normalize(Z_tr[:,1:])
Z_tr = np.concatenate((np.ones((n_tr,1)), Zn), axis=1)

# Compute Z_tst
Z_tst = poly.fit_transform(Xn_tst)
Zn, mz, sz = normalize(Z_tst[:,1:], mz, sz)
Z_tst = np.concatenate((np.ones((n_tst,1)), Zn), axis=1)

# Fit model to data.
LogReg.fit(Z_tr, Y_tr)

# Classify training and test data
D_tr = LogReg.predict(Z_tr)
D_tst = LogReg.predict(Z_tst)
```

[25]:
```
# Compute error rates
E_tr = D_tr!=Y_tr
E_tst = D_tst!=Y_tst

# Error rates
pe_tr = float(sum(E_tr)) / n_tr
pe_tst = float(sum(E_tst)) / n_tst

print('The final error rates are:')
print('- Training:', str(pe_tr))
print('- Test:', str(pe_tst))

# Compute Z_grid
Z_grid = poly.fit_transform(X_grid)
n_grid = Z_grid.shape[0]
Zn, mz, sz = normalize(Z_grid[:,1:], mz, sz)
Z_grid = np.concatenate((np.ones((n_grid,1)), Zn), axis=1)
```

```
The final error rates are:
- Training: 0.2608695652173913
- Test: 0.25806451612903225
```

[26]:
```
# Compute the classifier output for all samples in the grid.
dd = LogReg.predict(Z_grid)
pp = LogReg.predict_proba(Z_grid)[:,1]
pp = pp.reshape(xx.shape)

# Paint output maps
pylab.rcParams['figure.figsize'] = 6, 6  # Set figure size
plt.plot(x0c0, x1c0,'r.', label=labels[c0])
plt.plot(x0c1, x1c1,'g+', label=labels[c1])
plt.xlabel('$x_' + str(ind[0]) + '$')
plt.ylabel('$x_' + str(ind[1]) + '$')
plt.axis('equal')
plt.contourf(xx, yy, pp, cmap=plt.cm.copper)
plt.legend(loc='best')
plt.contour(xx, yy, pp, levels=[0.5], colors='b', linewidths=(3,))
plt.colorbar(CS, ticks=[0, 0.5, 1])
plt.show()
```



[ ]: