Linear and logistic regression

Linear regression; logistic regression;

training algorithms and loss functions;

Python implementation

Dataset representation

In the next models we will use the following representation for the data:

- We assume that all features are numerical
- Consider *m* examples and *n* features
- **Matrix representation**: X – matrix with input features and their values; y – vector with output for the feature values

$$x_j^{(i)}, y_i$$

represent respetively the value of the j-th feature for the i-th example (X_{ij}); and the value for the output feature of the i-th example

Classical regression models

They represent a relationship between the **inputs** $x_1,\ldots,x_n$ (independente variables), and the **output** y (dependente variable).

Prediction of the model given by (for the i-th example):

$$\hat{y}^{(i)} = h_\theta(x_1^{(i)},\ldots,x_n^{(i)})$$

n – nº of inputs
θ - parameters of the model

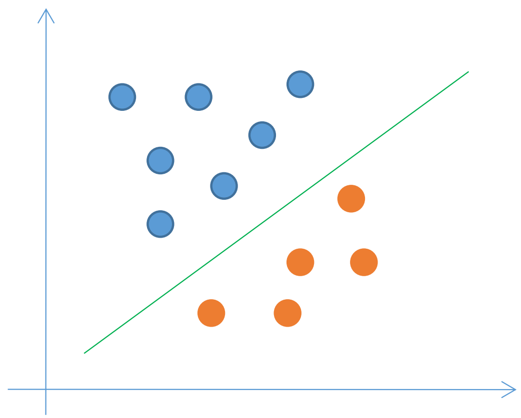
Linear models

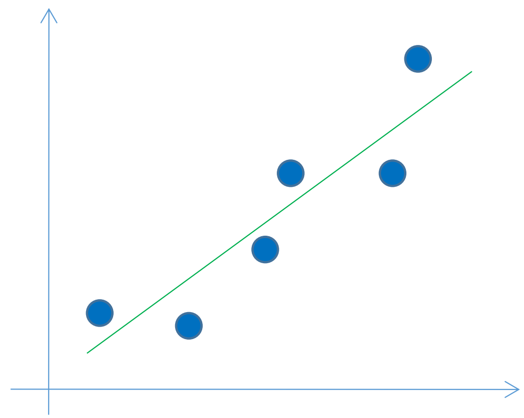
Attractive, given the simplicity of calculation and analysis

Linearity is defined in terms of functions with the properties:

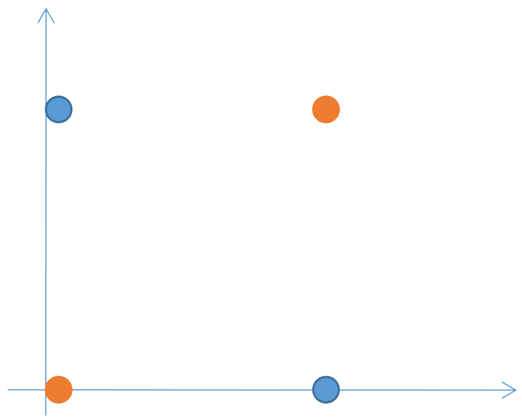
$f(x + y) = f(x) + f(y)$ and $f(ax)=af(x)$;

Can be used for **classification** (class discrimination) or **regression**.

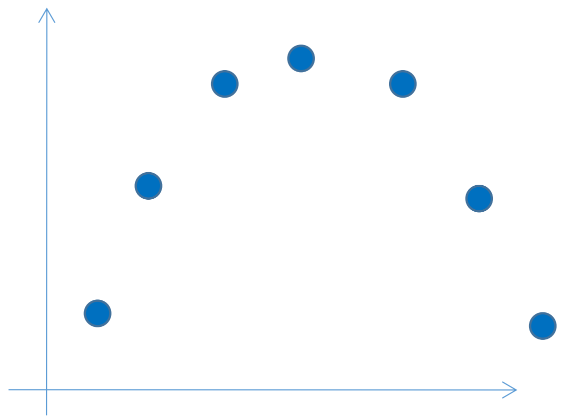




Non linear problems



Classification
(e.g. XOR)



Regression

Limitation:
In these cases, linear models may be insufficient …

Linear regression models

General case: **regression** models

$$\hat{y}^{(i)} = h_\theta(x^{(i)}) = \theta_0 + \sum_{j=1}^{n} \theta_j x_j^{(i)}$$

If n = 1: **linear regression**

If n >= 2: **multiple linear regression**

θ_i – model parameters

Linear regression models

Vector / matrix formulation

$$h_\theta(x) = \theta^T x = \theta_0 x_0 + \ldots + \theta_n x_n$$

where x is a vector of size n representing an example, and ***x_0* is considered equal to 1** by convenience (needs to be added to each example in the original dataset)

θ – vector of size $n+1$ with the model parameters

Linear regression models

Cost function: MSE – mean of squared errors

$$J_\theta = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Predicted value by the model: $\hat{y}^{(i)}$

Real value

J is a function of the model parameters $\theta_1, \ldots, \theta_n$

Objective: identify the model parameters that **minimize J**

Logistic regression

Discrete dependent variable: **classification** problem

Logistic regression: uses regression models for binary classification by interpreting the output of the model to extract a class

$$h_\theta(x) = g(\ \theta^T x\ ) \qquad\qquad 0 \leq h_\theta(x) \leq 1$$

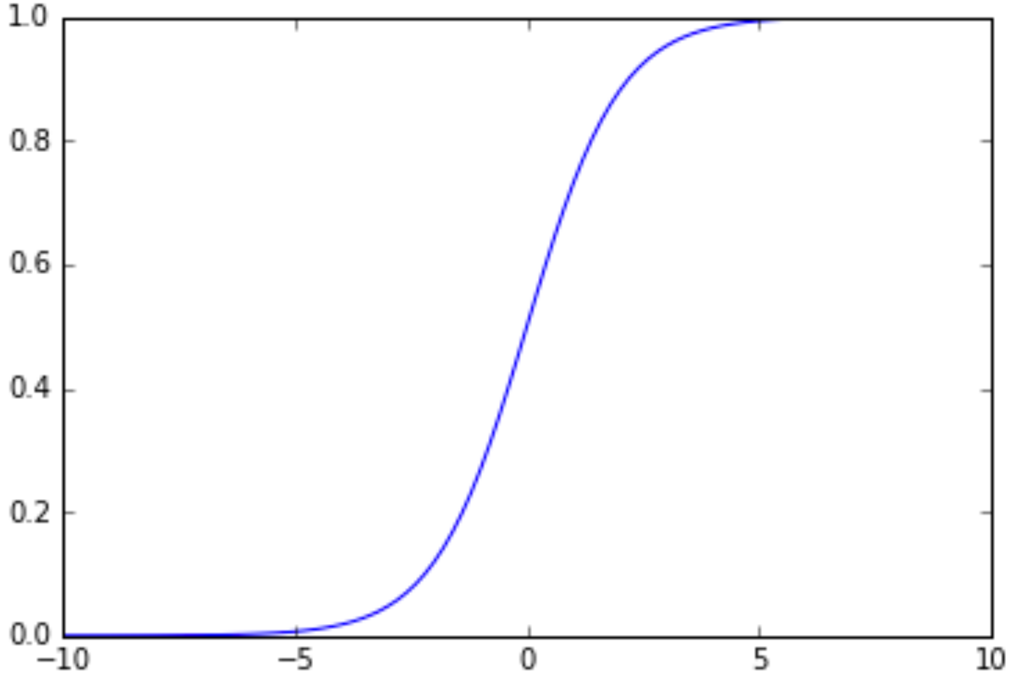
$$\boxed{h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}}$$

$$\boxed{\frac{1}{1+e^{-z}}}$$ **Sigmoid function**

Model is given by the application of the sigmoid function to the linear regression output

Interpretation: h estimates the **probability** of y (output) being 1 for the example *x*

Sigmoid function



For z = 0 => sigmoid(z) = 0.5
For z << 0 => sigmoid(z) approximates 0
For z >> 0 => sigmoid(z) approximates 1

Logistic regression: multiple classes

Logistic regression can be applied to cases with more than two classes

In this case, the strategy is to train a **"binary" model for each class** separately (considering the others as a single class)

Each model estimates the probability of the example being of a given class

When predicting new examples, each model is applied and then we choose the class whose predicted value is higher

Logistic regression: cost function

Loss function (for each example x):

$$\begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

If y = 1: If the prediction is correct: error is zero
Otherwise, as the prediction gets closer to 0, error tends to infinity

If y = 0: If the prediction is correct: error is zero
Otherwise, as the prediction gets closer to 1, error tends to infinity

Cost function

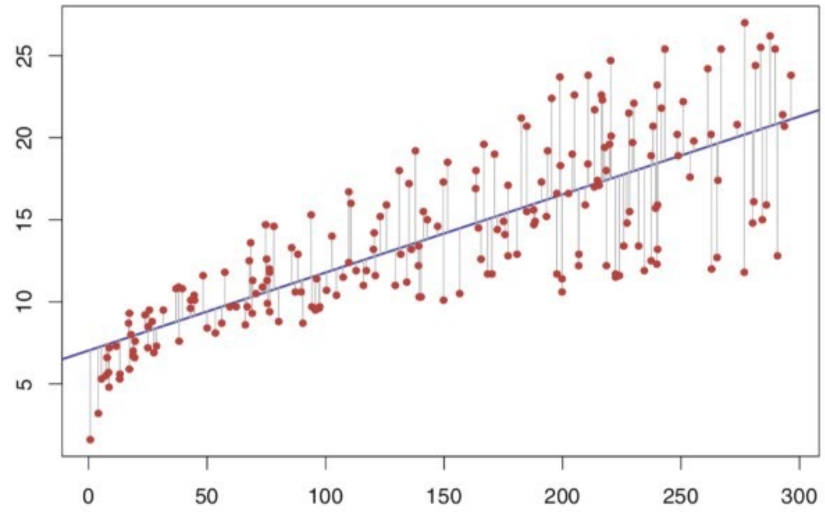
$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

Parameter estimation: numerical optimization

Knowing the structure of the model-> estimation of the parameters is a **numerical optimization** problem – minimization of the cost function

In the case of linear models, we can use the **analytical method** of **minimum squares**, minimizing the error function (MSE)

There are also alternative iterative methods as **gradient descent**



Parameter estimation with analytical method for linear regression

An algebraic method that involves solving a system of equations given by:

$$\frac{\partial}{\partial \theta_j} J(\theta) = 0, j = 1, \ldots, n$$

$$\theta = (X^T X)^{-1} X^T y$$

Matrix version
X includes examplos +
1st column with 1's

Parameter estimation: gradient descent for linear regression

Method that depends on the error function being **differentiable**

Iterative method, which in each iteration changes the values of each of the parameters θ_j

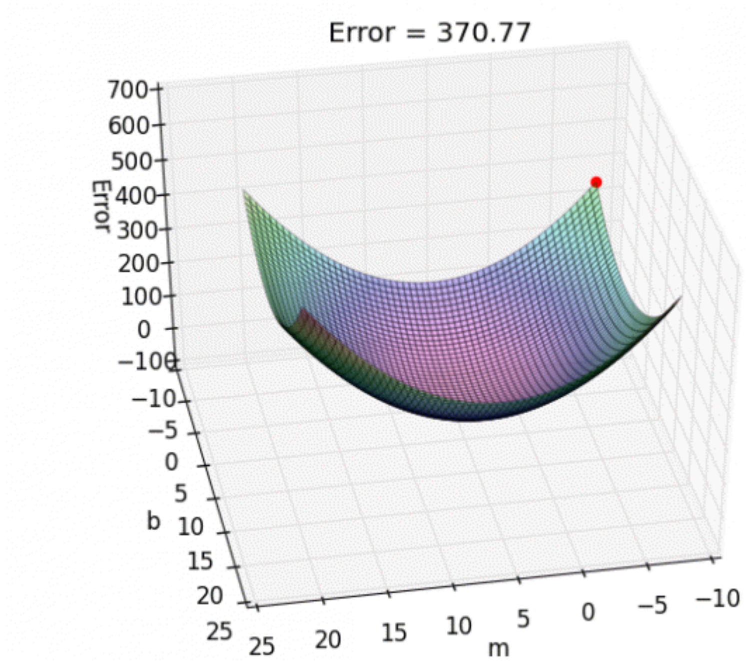
For each θ_j the update rule is the following:

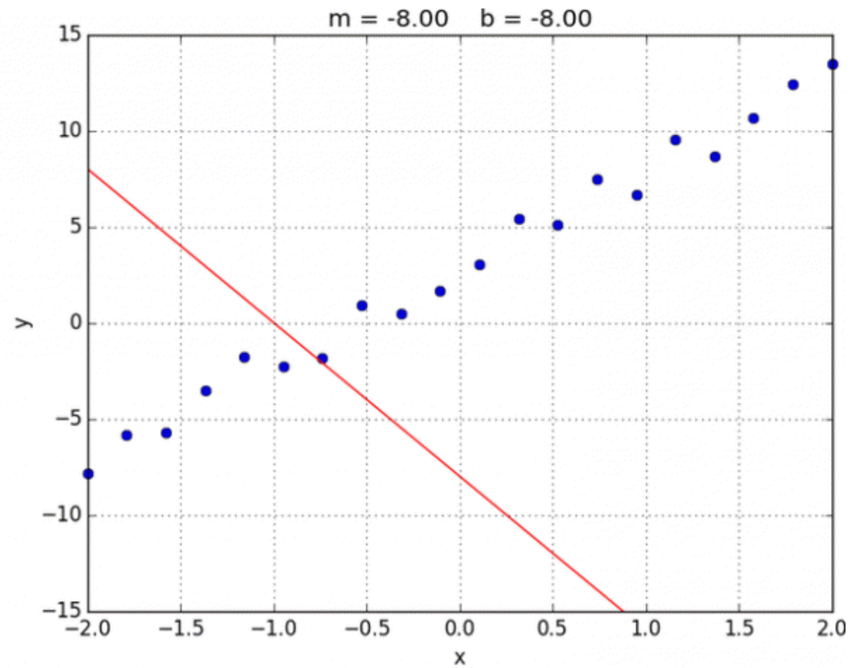
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Simultaneous updates in all parameters

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Parameter estimation:
gradient descent for linear regression





Logistic regression: parameter estimation

As in linear regression, parameters are estimated minimizing the J error function

Process can be performed using gradient descent method

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Algorithm identical to the previous one, but function *h* is different

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Gradient descent: learning rate

The α parameter is called the **learning rate** and it controls the "speed" of parameter updates

High values for α can lead to faster convergence, but they carry risks of divergence

Low values for α ensure convergence, but it can be slow

Parameter estimation: gradient descent vs analytical method

Analytical method ensures the optimal solution; GD may not converge

In the analytical method there are no parameters; GD may take time to converge

Analytical method can become slow with large n (matrices n x n may become untreatable for $n > 10^5$)

GD more generic and applicable to other types of models

Parameter estimation: advanced methods

In many cases, gradient descent is too slow in its convergence to be used in practice without modifications

Other more advanced numerical optimization methods may be used, most of them based on gradient descent as its core

Note:

Python example with the **fmin** function - package **optimize** (In this case, you don't need derivatives)

Other alternatives are available in the same package

Non-linear models - examples

Generalized linear regression

$$\hat{y} = \beta_0 + \sum_{i=1}^{p} \beta_i f_i(x_i)$$

f_i – non-linear functions

Polynomial regression

$$\hat{y} = \beta_0 + \sum_{i=1}^{O_1} \beta_i x_1^i + \sum_{j=1}^{O_2} \beta_j x_2^j + \cdots$$

They can be obtained using linear/logistic regression, but considering previous transformations in the input variables

Data preparation: handling nominal features

In functional models, all features will have to be **numerical** – need to convert nominal values to numerical

Hypothesis 1: divide numerical range allowed by the number of features values: 1 nominal feature => 1 numerical feature

Hypothesis 2: binarize the feature: 1 nominal feture with M possible values => M numerical features with binary values (**one-hot encoding**)

Standardization

Transformations in the data often needed for the learning algorithm to work (better).

Gradient descent algorithms may work worse with variables with very different scales

Various possible methods:

- Convert to mean 0 and standard deviation 1
- Convert to a range [0,1] or [-1, 1]

Numpy implementation

- Let's implement the linear and logistic regression methods in two simple Python classes
- The basis for the data representation will rest on the data structures from the numpy package
- Implementation of the code will be vectorized to be more efficient

Package ***NumPy***

Includes data structures and functions to facilitate the development of algebraic and numerical methods in python

Object ***array(ndarray)***: allows to define and manipulate multidimensional vectors, matrices, and arrays with a wide range of functions available

Documentation: *https://numpy.org/doc/stable/*

May be useful (cheat sheet):

https://sebastianraschka.com/blog/2014/matrix_cheatsheet_table.html

Class *Dataset*

PYTHON IMPLEMENTATION

dataset.py

Class that implements datasets

Very simple initial version considering data array as numpy array

```
import numpy as np

 def __init__(self, filename = None, X = None, Y = None):
        if filename is not None:
            self.readDataset(filename)
       ( … )

 def readDataset(self, filename, sep = ","):
            data = np.genfromtxt(filename, delimiter=sep)
            self.X = data[:,0:-1]
            self.Y = data[:,-1]
```

PYTHON IMPLEMENTATION

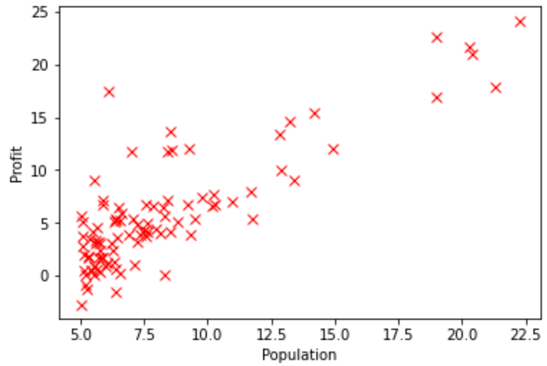
Class *Dataset*

Data standardization

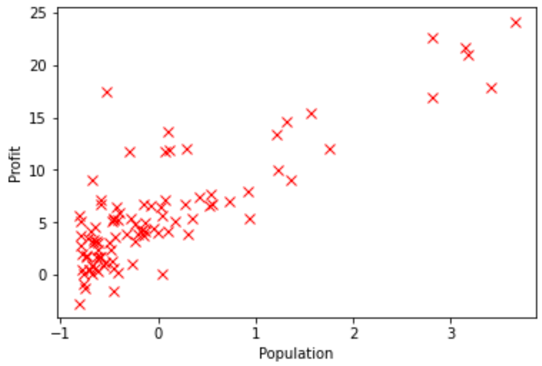
```
def standardize(self):
        self.mu = np.mean(self.X, axis = 0)
        self.Xst = self.X - self.mu
        self.sigma = np.std(self.X, axis = 0)
        self.Xst = self.Xst / self.sigma
```

Visualization

```
def plotData2vars(self, xlab, ylab, standardized = False):
        if standardized:
            plt.plot(self.Xst, self.Y, 'rx', markersize=7)
        else:
            plt.plot(self.X, self.Y, 'rx', markersize=7)
        plt.ylabel(ylab)
        plt.xlabel(xlab)
        plt.show()
```



Function ***test***:
Original data



Function ***testStandardized***:
Standardised data

PYTHON IMPLEMENTATION

linear_regression_v1.py

Class *LinearRegression*

Class that implements the linear regression models

```
import numpy as np

class LinearRegression:

    def __init__(self, dataset, standardize = False):
        if standardize:
            dataset.standardize()
            self.X = np.hstack ((np.ones([dataset.nrows(),1]), dataset.Xst ))
            self.standardized = True
        else:
            self.X = np.hstack ((np.ones([dataset.nrows(),1]), dataset.X ))
            self.standardized = False
        self.y = dataset.Y
        self.theta = self.theta = np.zeros(self.X.shape[1])
        self.data = dataset
```

Class Variables:

- *X*, *y* – dataset
- *theta* – Model Parameters
- *standardized* – flag that indicates whether data has been standardized

PYTHON IMPLEMENTATION

Linear regression

predict – Predict the value for a new instance

```
def predict(self, instance):
        x = np.empty([self.X.shape[1]])
        x[0] = 1
        x[1:] = np.array(instance[:self.X.shape[1]-1])

        if self.standardized:
            x[1:] = (x[1:] - self.data.mu) / self.data.sigma
        return np.dot(self.theta, x)
```

costFunction – calculates the value of the cost function (for all dataset examples)

```
def costFunction(self):
    m = self.X.shape[0]
    predictions = np.dot(self.X, self.theta)
    sqe = (predictions - self.y) ** 2
    res = np.sum(sqe) / (2*m)
    return res
```

PYTHON IMPLEMENTATION

Linear regression

buildModel – Creates the model using the analytical method

```
def buildModel(self, dataset):
    from numpy.linalg import inv
    self.theta = inv(self.X.T.dot(self.X)).dot(self.X.T).dot(self.y)
```

gradientDescent – Creates the model using gradient descent

```
def gradientDescent (self, iterations = 1000, alpha = 0.001):
    m = self.X.shape[0]
    n = self.X.shape[1]
    self.theta = np.zeros(n)
    for its in range(iterations):
        J = self.costFunction()
        if its%100 == 0: print(J)
        delta = self.X.T.dot(self.X.dot(self.theta) - self.y)
        self.theta -= (alpha /m * delta )
```

PYTHON IMPLEMENTATION

Linear regression (No regularization)

Test the implementation:

Example with 2 variables

```
def test_2var():
    ds= Dataset("lr-example1.data")
    lrmodel = LinearRegression(ds)
...
```

Example with 3 variables

```
def test_multivar():
    ds= Dataset("lr-example2.data")
    lrmodel = LinearRegression(ds)

...
```

Implementation – logistic regression (exercise)

Based on the LinearRegression class, the goal will be to implement a class for logistic regression!

A file will be provided with some methods implemented and others to complete! ***logistic_regression_incomp.py***

Class *LogisticRegression*

PYTHON IMPLEMENTATION

logistic_regression_incomp.py

Class that implements the logistic regression models

```
import numpy as np
from dataset import Dataset

class LogisticRegression:
    def __init__(self, dataset, standardize = False):
        if standardize:
            dataset.standardize()
            self.X = np.hstack ((np.ones([dataset.nrows(),1]), dataset.Xst ))
            self.standardized = True
        else:
            self.X = np.hstack ((np.ones([dataset.nrows(),1]), dataset.X ))
            self.standardized = False
        …
```

Implementation - logistic regression (exercise)

Create methods for:

- def ***probability***(self, instance) – predict the probability value for a new instance
- def ***predict*** (self, instance) – predict the (binary) value for a new instance
- def ***costFunction*** (self) – calculates the value of the cost function (for all dataset examples)
- def ***gradientDescent*** (self, dataset, alpha = 0.01, iters = 10000) – creates the model using gradient descent

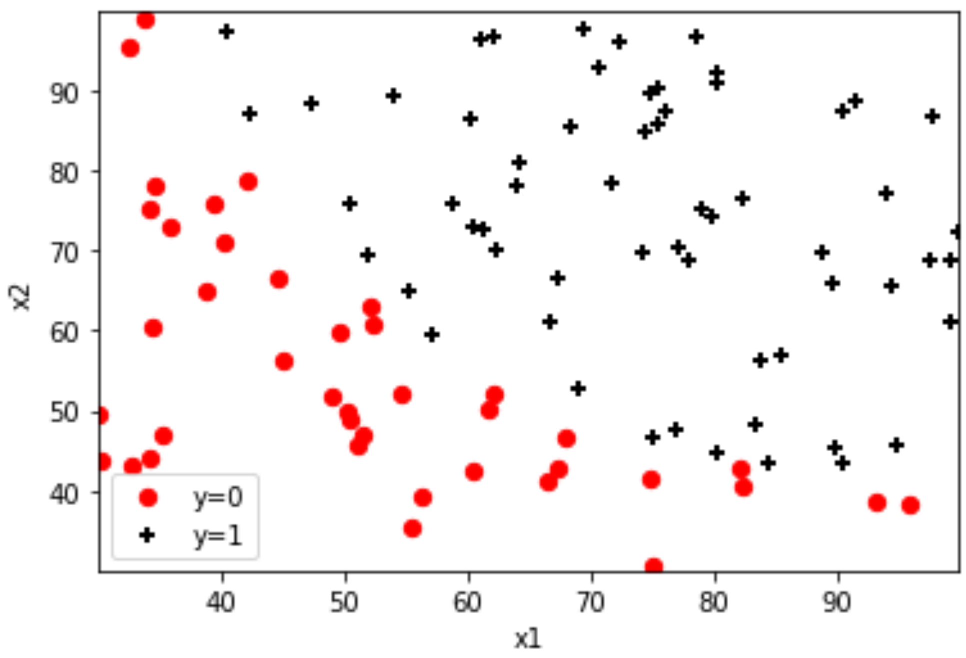
PYTHON IMPLEMENTATION

Implementation – logistic regression

Test the implementation with example log-ex1.data (2 variables):

```
def plotBinaryData(self):
	negatives = self.X[self.Y == 0]
	positives = self.X[self.Y == 1]
	plt.xlabel("x1")
	plt.ylabel("x2")
	…


def testBinary():
	ds= Dataset("log-ex1.data")
	ds.plotBinaryData()
```



Class **Dataset**

Implementation – logistic regression

Test the implementation with example log-ex1.data (2 variables):

```
def test():
    ds= Dataset("log-ex1.data")
    logmodel = LogisticRegression(ds)

    print ("Initial cost: ", logmodel.costFunction())

    logmodel.gradientDescent(ds, 0.002, 200000)
    logmodel.plotModel()
    print ("Final cost:", logmodel.costFunction())

    ex = np.array([45,65])
    print ("Prob. example:", logmodel.probability(ex))
    print ("Pred. example:", logmodel.predict(ex))
```

PYTHON IMPLEMENTATION

Class *LogisticRegression*

Gradient descent with more sophisticated optimization methods

```
def optim_model(self):
    from scipy import optimize
    n = self.X.shape[1]
    options = {'full_output': True, 'maxiter': 400}
    initial_theta = np.zeros(n)
    self.theta, _, _, _, _ =
      optimize.fmin(lambda theta: self.costFunction(theta), initial_theta, **options)
```

Adapt the test code to use this function in place of the gradient descent and compare results

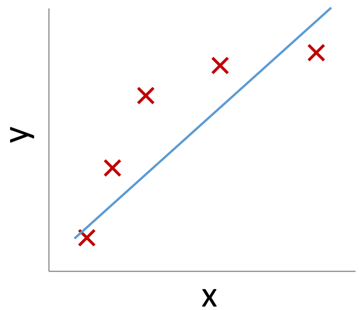
Overfitting and regularization

Overfitting

Regularization methods in linear and logistic regression

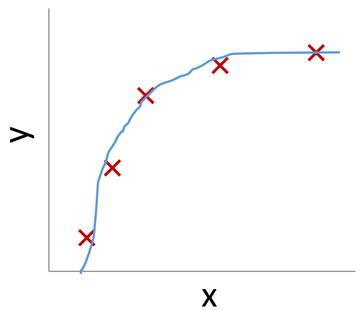
Overfitting in functional models

If we have a high number of features, the model can fit "too well" the training data and lose generalization



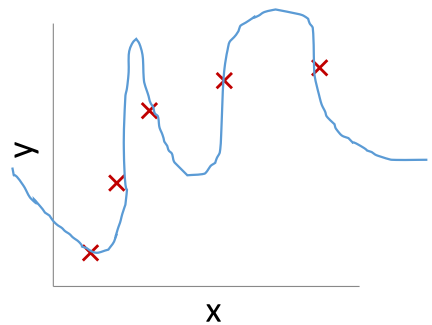
$$\theta_0 + \theta_1 x$$

Underfitting:
Insufficient model complexity



$$\theta_0 + \theta_1 x + \theta_2 x^2$$

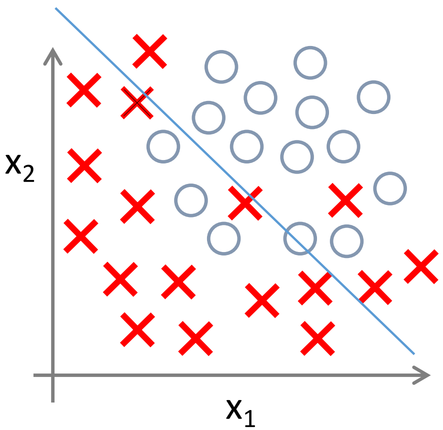
“Adequate” model complexity



$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

Overfitting:
Excessive complexity

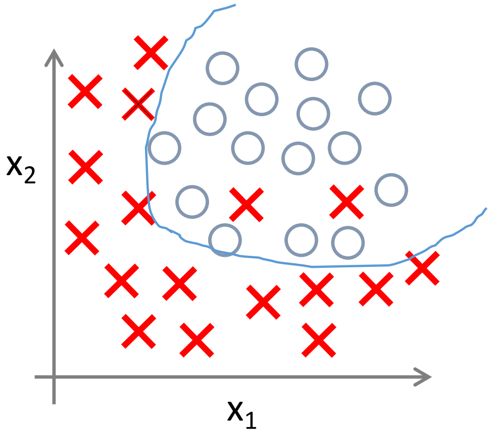
Overfitting in logistic regression: an example



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

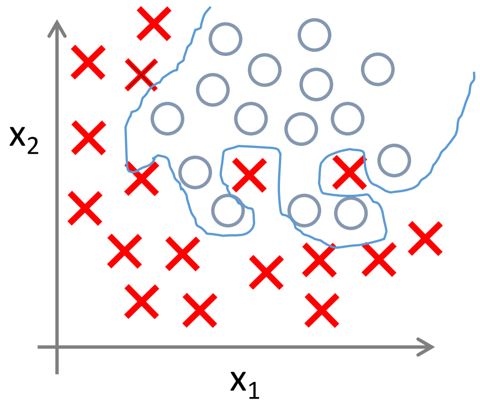
g = sigmoid function

Underfitting:
Insufficient model complexity



$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 \\ +\theta_3 x_1^2 + \theta_4 x_2^2 \\ +\theta_5 x_1 x_2)$$

"Adequate" model complexity



$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 \\ +\theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 \\ +\theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \ldots)$$

Overfitting:
Excessive complexity

Regularization: cost function

Idea: penalize high values of the parameters in the cost function

For linear regression:

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right]$$

Regularization parameter: higher values penalize parameter values more
If too high: risk of underfitting
If too low: risk of overfitting

Regularization for linear regression

Analytical method:

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix} \right)^{-1} X^T y$$

Regularization for linear regression

Gradient descent

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\theta_j := \theta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$(j = \cancel{0}, 1, 2, 3, \ldots, n)$$

Term imposed by regularization
Note that the expression between () is always < 1

Regularization in linear regression

Previous method named ***Ridge regression***: uses squared values (norm L2) in the penalty

Alternative: use sum of absolute values (norm L1) of the parameters – ***Lasso regression***

Elastic nets – use a combination of both L2 and L1 from Ridge and Lasso with two parameters

Regularization in logistic regression

Cost function

$$J(\theta) = \left[-\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log\left(h_\theta(x^{(i)}) + (1-y^{(i)})\log 1 - h_\theta(x^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Regularization term

Gradient

$$\frac{\partial}{\partial\theta_0}J(\theta) \qquad \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\frac{\partial}{\partial\theta_1}J(\theta) \qquad \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)} - \frac{\lambda}{m}\theta_1$$

(...)

PYTHON IMPLEMENTATION

Linear regression with regularization

linear_regression_v2.py

- ***analyticalWithReg*** – analytical method for building the model with regularization

```
def analyticalWithReg (self):
    from numpy.linalg import inv
    matl = np.zeros([self.X.shape[1], self.X.shape[1]])
    for i in range(1,self.X.shape[1]): matl[i,i] = self.lamda
    mattemp = inv(self.X.T.dot(self.X) + matl)
    self.theta = mattemp.dot(self.X.T).dot(self.y)
```

- método ***buildModel -*** adapted to use the previous one when the regularization flag is used

```
def buildModel (self, dataset):
    from numpy.linalg import inv
    if self.regularization:
        self.analyticalWithReg()
    else:
        self.theta =inv(self.X.T.dot(self.X)).dot(self.X.T).dot(self.y)
```

Linear regression with regularization

```
def gradientDescent (self, iterations = 1000, alpha = 0.001):
    m = self.X.shape[0]
    n = self.X.shape[1]
    self.theta = np.zeros(n)
    if self.regularization:
            lamdas = np.zeros([self.X.shape[1]])
            for i in range(1,self.X.shape[1]):
                lamdas[i] = self.lamda
    for its in range(iterations):
        J = self.costFunction()
        if its%100 == 0: print(J)
        delta = self.X.T.dot(self.X.dot(self.theta) - self.y)
        if self.regularization:
                self.theta -= (alpha/m * (lamdas+delta))
        else:
                self.theta -= (alpha /m * delta )
```

Method **gradientDescent** adapted to use regularization when the regularization flag is used

Test your code with the previous examples**: flag regul = True**

```
def test_2var(regul = False):
    ds= Dataset("lr-example1.data")
    if regul:
        lrmodel = LinearRegression(ds, True, True, 100.0)
    else:
        lrmodel = LinearRegression(ds)

....
```

Logistic regression with regularization

PYTHON IMPLEMENTATION

logistic_regression_v2.py

Cost Function with regularization

```
def costFunctionReg (self, theta = None, lamda = 1):
    if theta is None: theta= self.theta
    m = self.X.shape[0]
    p = sigmoid ( np.dot(self.X, theta) )
    cost = (-self.y * np.log(p) - (1-self.y) * np.log(1-p) )
    reg = np.dot(theta[1:], theta[1:]) * lamda / (2*m)
    return (np.sum(cost) / m) + reg
```

More sophisticated methods of optimization

```
def optim_model_reg(self, lamda):
    from scipy import optimize
    n = self.X.shape[1]
    initial_theta = np.zeros(n)
    result = optimize.minimize(lambda theta: self.costFunctionReg(theta, lamda),
                initial_theta, method='BFGS',  options={"maxiter":500, "disp":True} )
    self.theta = result.x
```

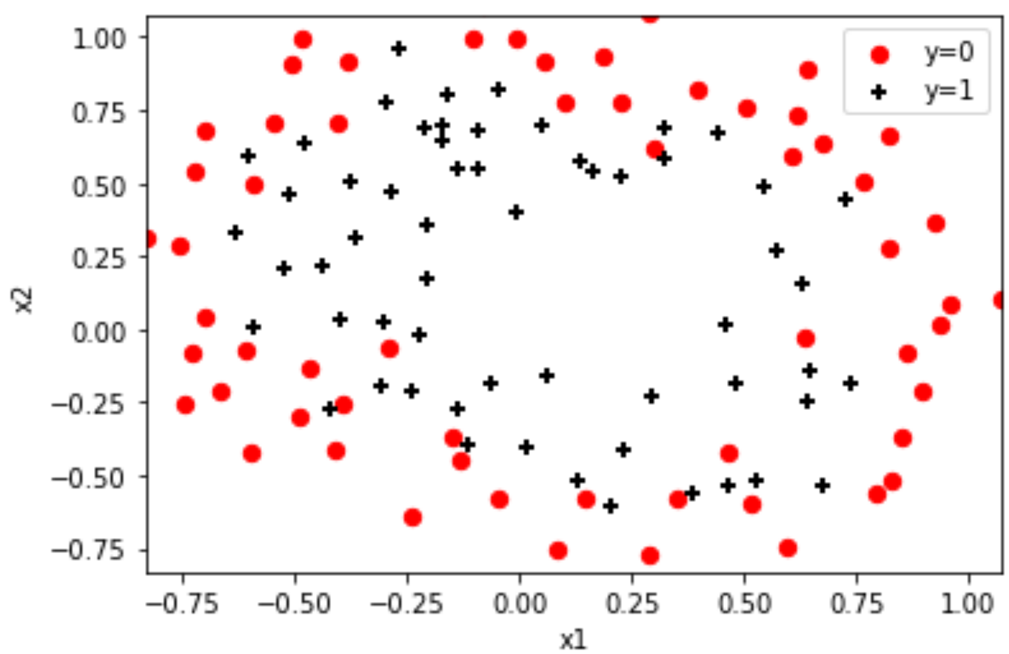
PYTHON IMPLEMENTATION

Logistic regression with regularization - example

Dataset: *log-ex2*:

```
def plotBinaryData(self):
        negatives = self.X[self.Y == 0]
        positives = self.X[self.Y == 1]
        plt.xlabel("x1")
        plt.ylabel("x2")
        ...


def testBinary2():
        ds= Dataset("log-ex2.data")
        ds.plotBinaryData()
```

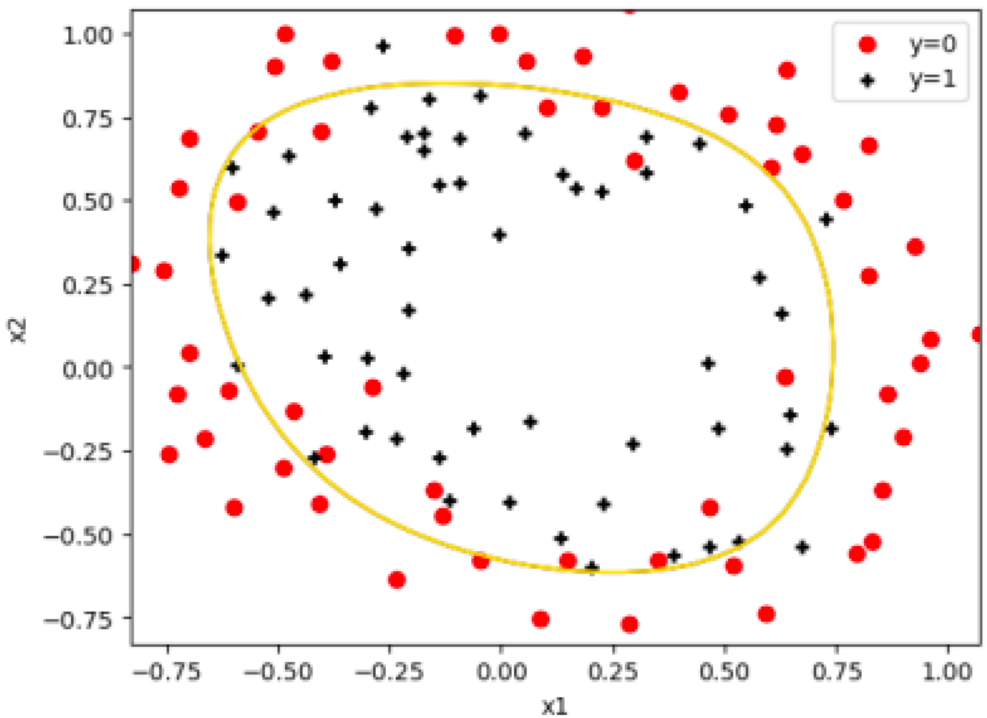


Logistic regression with regularization - example

PYTHON IMPLEMENTATION

Let's create helper attributes from the original variables x_1 and x_2 with all polynomial terms up to degree 6

$$\begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1x_2^5 \\ x_2^6 \end{bmatrix}$$



Analysis

- Run Logistic Regression version with regularization – ***testreg()***
- Check the graph showing the discrimination generated by the algorithm
- Analyze various values of the regularization parameter (e.g. 0, 1, 10, 100, …)

- Test different configurations of the Logistic Regression for the *hearts* dataset. Notice that you have the **holdout** function implemented.

Logistic regression – example with a larger dataset

PYTHON IMPLEMENTATION

Let us now test this code with a larger dataset – hearts

We will need to predict values for a test set

```
def predictMany(self, Xt):
    p = sigmoid ( np.dot(Xt, self.theta) )
    return np.where(p >= 0.5, 1, 0)

def accuracy(self, Xt, yt):
    preds = self.predictMany(Xt)
    errors = np.abs(preds-yt)
    return 1.0 - np.sum(errors)/yt.shape[0]
```

… and calculate error metrics (in this case **accuracy**)

This function wraps it all: creates training and test sets, trains the model in the training set and evaluates it in the test set

```
def holdout(self, p = 0.7):
    dataset = Dataset(None, X = self.X, Y = self.y)
    Xtr, ytr, Xts, yts = dataset.train_test_split(p)
    self.X = Xtr
    self.y = ytr
    self.buildModel()
    return self.accuracy(Xts, yts)
```

Analysis

- Test different configurations of the Logistic Regression for the *hearts* dataset.
- Use the **holdout** function implemented before to try diferente values of lambda and create a table to select the best one (to have more statistical significance repeat the holdout process for 5 or 10 x)

- Notice that lambda is a hyperparameter for regularized methods !

PYTHON IMPLEMENTATION

Implementing supervised ML pipelines in Python: scikit-learn

Available classes in scikit-learn include those implementing linear models, in the **linear_model** module for classification and regression including:

- Linear regression (with analytical method) – class LinearRegression
- Regularized linear regression (several methods) – classes Ridge, Lasso, ElasticNets
- Logistic regression (different optimization methods/ solvers + regularization) – class LogisticRegression

These classes follow the same interface all other supervised models

https://scikit-learn.org/stable/modules/linear_model.html