NumPy Package

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

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

Documentation: www.numpy.org

NumPy Implementation

PyTorch

```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l

n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

TensorFlow

```
%matplotlib inline
import math
import time
import numpy as np
import tensorflow as tf
from d2l import tensorflow as d2l

n = 10000
a = tf.ones(n)
b = tf.ones(n)
```

NumPy Implementation

```
class Timer:
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        self.tik = time.time()

    def stop(self):
        self.times.append(time.time() - self.tik)
        return self.times[-1]
```

NumPy Implementation

PyTorch

```
timer.start()
d = a + b
f'{timer.stop():.5f} sec'
```

```
>'0.00026 sec'
```

TensorFlow

```
timer.start()
d = a + b
f'{timer.stop():.5f} sec'
```

```
>'0.00020 sec'
```

Class Dataset

Class that implements the datasets. Very simple initial version considering data array as `numpy` array.

```
import numpy as np
import matplotlib.pyplot as plt

class Dataset:

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

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

    def getXy (self):
            return self.X, self.Y
```

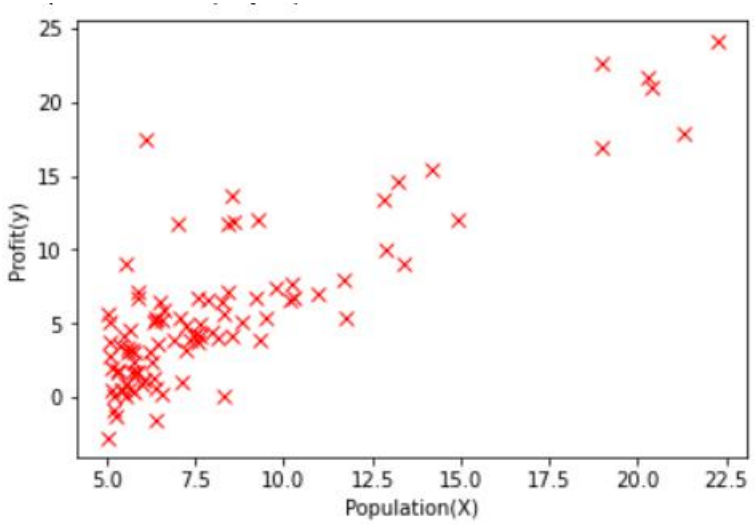
Class Dataset

Testing the class and visualizing the dataset

```
def plotData_2vars(X, Y, xlabel, ylabel):
    plt.plot(X, Y, 'rx', markersize=7)
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.show()

test_dataset("lr-example1.data")
```

```
def test_dataset(ds_path):
    d = Dataset(ds_path)
    X,y = d.getXy()
    print(f"Shape Atributos: {X.shape}")
    print(f"Shape Labels: {X.shape}")
    plotData_2vars(X,y,"Population(X)", "Profit(y)")
```

```
> Shape Atributos: (97, 1)
  Shape Labels: (97, 1)
```



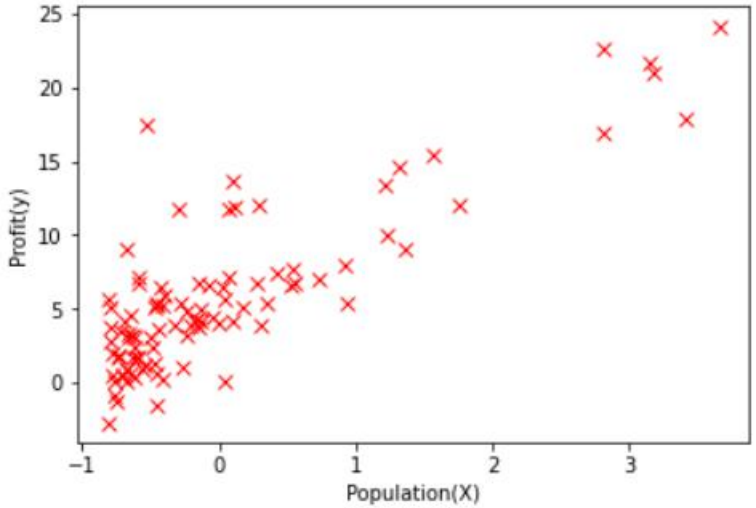
Normalizing the Data

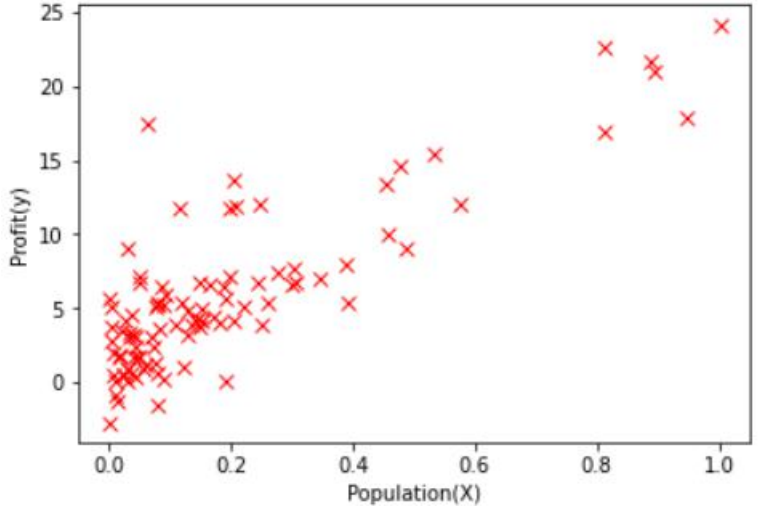
```
def standardize(X):
    Xmed = np.mean(X, axis = 0)
    Xsigma = np.std(X, axis = 0)
    return (X-Xmed)/Xsigma

def normalize(X):
    Xmax = np.max(X, axis = 0)
    Xmin = np.min(X, axis = 0)
    return (X-Xmin)/(Xmax-Xmin)



test_standardize()
test_normalize()
```

```
def test_standardize():
    d = Dataset("lr-example1.data")
    X,y = d.getXy()
    Xs= standardize(X)
    print(f"Shape Atributos: {Xs.shape}")
    plotData_2vars(Xs,y,"Population(X)", "Profit(y)")

def test_normalize():
    d = Dataset("lr-example1.data")
    X,y = d.getXy()
    Xn= normalize(X)
    print(f"Shape Atributos: {Xn.shape}")
    plotData_2vars(Xn,y,"Population(X)", "Profit(y)")
```

Normalizing the Data

```
> Shape Atributos: (97, 1)
```

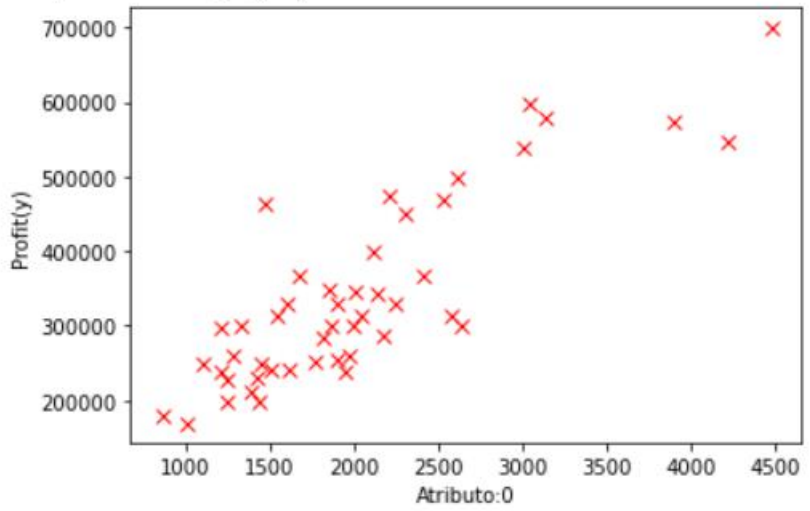




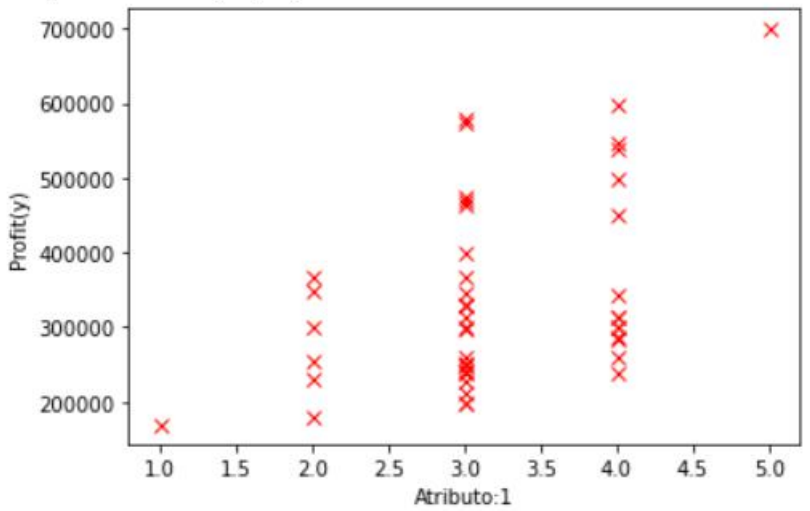
Testing With Several Attributes

```
def test_dataset_multi(ds_path,coluna=0):
    d = Dataset(ds_path)
    X,y = d.getXy()
    print(f"Shape Atributos: {X.shape}")
    print(f"Shape Labels: {X.shape}")
    plotData_2vars(X[:,coluna],y,f"Atributo:{coluna}", "Profit(y)")

test_dataset_multi("lr-example2.data",0)
test_dataset_multi("lr-example2.data",1)
```

```
> Shape Atributos: (47, 2)
  Shape Labels: (47, 2)
```



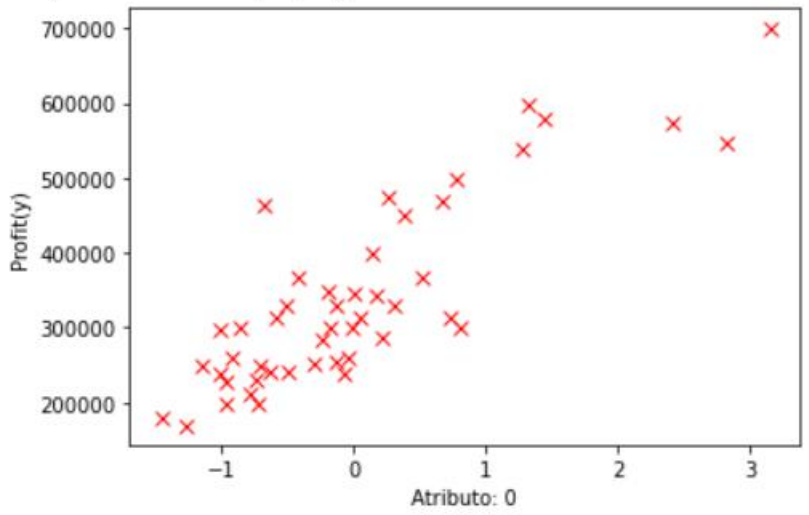

```
Shape Atributos: (47, 2)
Shape Labels: (47, 2)
```

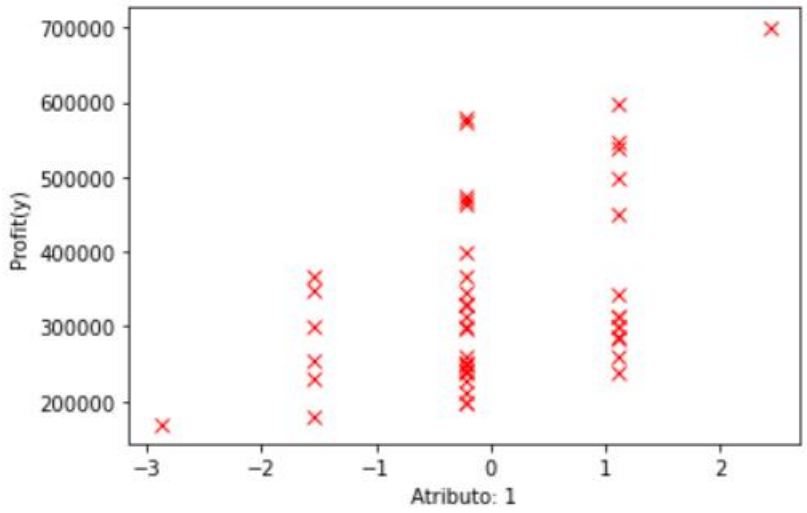


Testing With Several Attributes

```
def test_standardize():
    print(f"############## Atributos Standardizados ###################")
    d = Dataset("lr-example2.data")
    X,y = d.getXy()
    Xs= standardize(X)
    print(f"Shape Atributos: {Xs.shape}")
    plotData_2vars(Xs[:,0],y,"Atributo: 0", "Profit(y)")
    plotData_2vars(Xs[:,1],y,"Atributo: 1", "Profit(y)")
test_standardize()
```

```
> ############## Atributos Standardizados ##################
Shape Atributos: (47, 2)
```

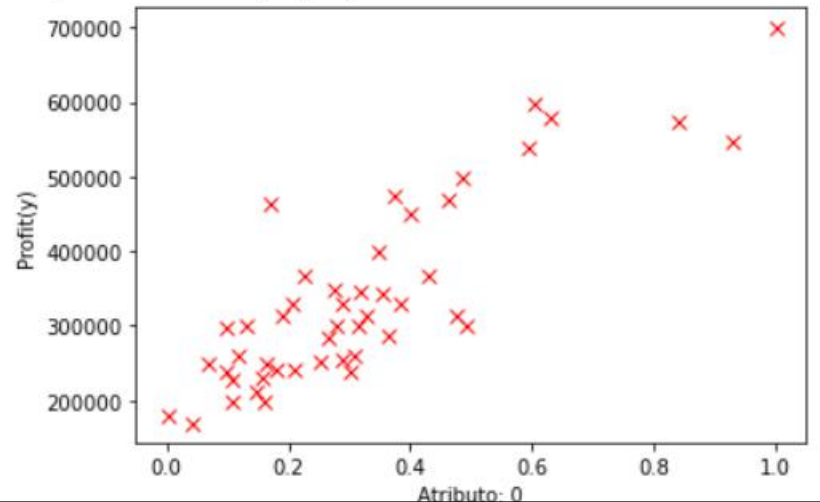


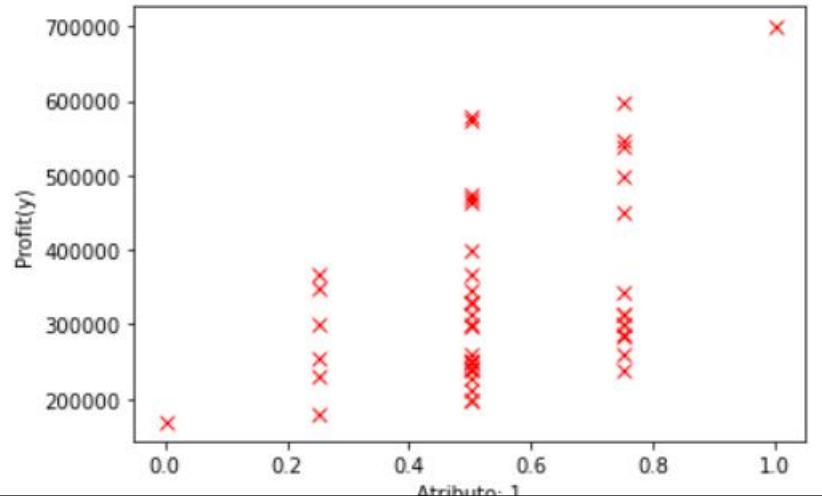


Testing With Several Attributes

```
def test_normalize():
    print(f"############## Atributos Normalizados ###################")
    d = Dataset("lr-example2.data")
    X,y = d.getXy()
    Xn= normalize(X)
    print(f"Shape Atributos: {Xn.shape}")
    plotData_2vars(Xn[:,0],y,"Atributo: 0", "Profit(y)")
    plotData_2vars(Xn[:,1],y,"Atributo: 1", "Profit(y)")

test_normalize()
```

```
> ############## Atributos Normalizados ##################
  Shape Atributos: (47, 2)
```





Class `LinearRegression`

Class that implements the linear regression models.

Class variables:

- `X, y` – dataset
- `theta` – model parameters
- `regularization` – flag that defines whether to use regularization
- `lamda` – regularization parameter
- `normalized` – flag that indicates whether data has been standardized

Class `LinearRegression`

Methods:

- `def predict(self, instance)` – predict the value for a new instance
- `def costFunction(self)` – calculates cost function value (for all dataset examples)
- `def buildModel(self, dataset)` – builds the model using the analytic method
- `def gradientDescent(self, iterations = 1000, alpha = 0.001)` – creates the model using gradient descent

Class `LinearRegression`

Calculating the error

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

Class LinearRegression

```
class LinearRegression:


    def __init__(self, dataset, standardization = False, regularization = False, lamda = 1):
        self.X, self.y = dataset.getXy()
        self.standardization = standardization
        self.regularization = regularization
        self.lamda = lamda
        self.X = np.hstack ((np.ones([self.X.shape[0],1]), self.X )
        self.theta = np.zeros(self.X.shape[1])
        if self.standardization:
            self.X[:,1:] = standardize(self.X[:,1:])
```

Class LinearRegression

```
class LinearRegression:

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

    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
        theta = inv(self.X.T.dot(self.X) + matl).dot(self.X.T).dot(self.y)
        return theta
```

Class LinearRegression

```
class LinearRegression:


   def predictions (self): #previsões dos casos de teste
            return np.dot(self.X, self.theta)


    def predict(self, instance):
        x = np.empty([self.X.shape[1]])
        x[0] = 1 #na 1ª posição coloca um 1
        x[1:] = np.array(instance[:self.X.shape[1]-1])
        print(instance)
        print(x)
        if self.standardization:
            #x[1:] = (x[1:] - self.mu) / self.sigma
            x[1:] = standardize(x[1:])
        return np.dot(self.theta, x)


    def costFunction(self):
        J = error_sqe(self.predictions(),self.y)
        return J
```

Class LinearRegression

```
class LinearRegression:


    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):
            #predictions = np.dot(self.X, self.theta)
            J = error_sqe(self.predictions(),self.y)
            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 )
```

Class LinearRegression

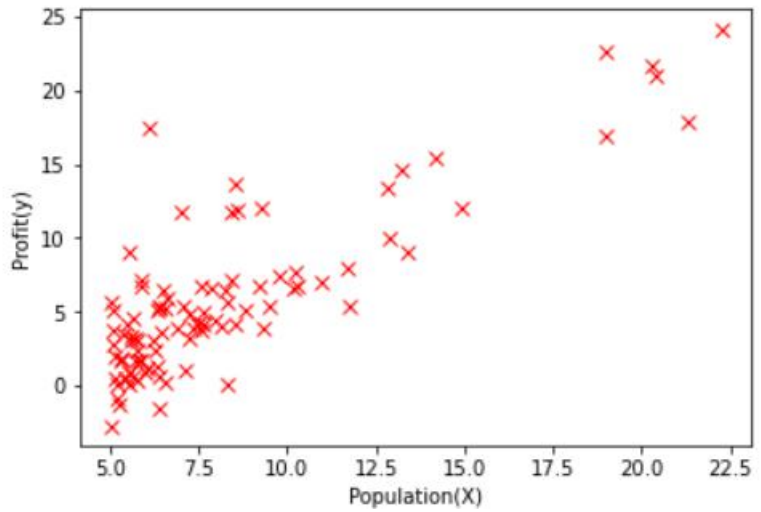
```
class LinearRegression:


    def getXy (self):
            return self.X[:,1], self.y
    def getX (self):
            return self.X


    def get_theta (self):
            return self.theta
```

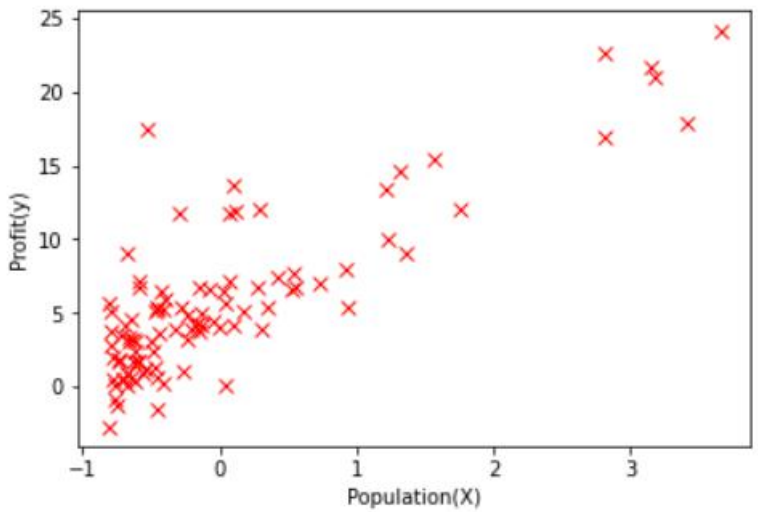
Class LinearRegression

Testing the class

```
def test_2var_1(regul = False):
    ds= Dataset("lr-example1.data")
    if regul:
        lrmodel = LinearRegression(ds, True, True, 100.0)
    else:
        lrmodel = LinearRegression(ds)
    X,y = lrmodel.getXy()
    plotData_2vars(X,y,"Population(X)", "Profit(y)")
    print("Cost function value for theta with zeros:")
    print(lrmodel.costFunction())

test_2var_1(False)
test_2var_1(True)
```

Class `LinearRegression`






```
Cost function value for theta with zeros:
32.072733877455676
Cost function value for theta with zeros:
32.072733877455676
```

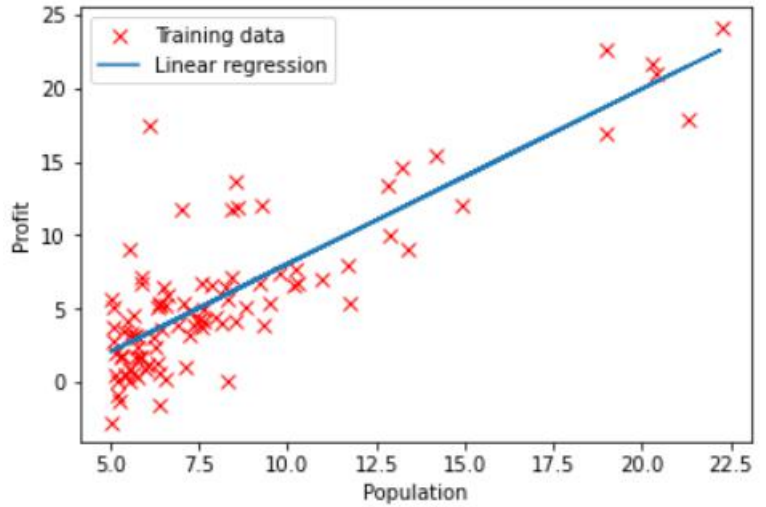
Class LinearRegression

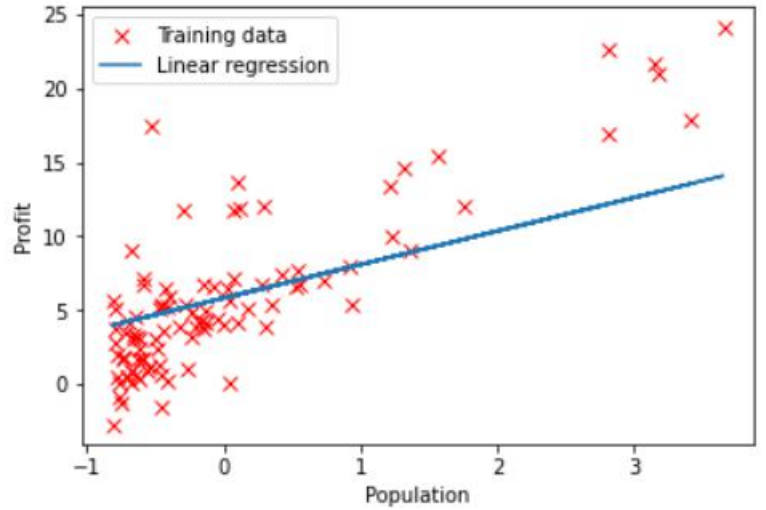
Testing the class

```
def test_2var_3(regul = False):
    ds= Dataset("lr-example1.data")
    if regul:
        lrmodel = LinearRegression(ds, True, True, 100.0)
    else:
        lrmodel = LinearRegression(ds)
    print(f"Model with analytical solution. Regularização = {regul}")
    lrmodel.buildModel()
    print(f"Cost function value for theta from analytical solution. Regularização = {regul}")
    print(lrmodel.costFunction())
    X,y = lrmodel.getXy()
    print(f"Casos. Regularização = {regul}")
    print(f"Shape: {X.shape}")
    print(X)
    print(f"Labels. Regularização = {regul}")
    print(f"Shape: {y.shape}")
    print(y)
    print(f"Coeficientes da solução analitica. Regularização = {regul}")
    theta = lrmodel.get_theta ()
    print(f"Shape: {theta.shape}")
    print(theta)
    print(f"Previsão para os casos de Treino usando o modelo. Regularização = {regul}")
    pred = lrmodel.predictions()
    print(pred.shape)
    print(pred)
    plotDataAndModel(X,y,pred, "Population", "Profit")

test_2var_3(False)
test_2var_3(True)
```

Class LinearRegression

```
> Model with analytical solution. Regularização = False
Cost function value for theta from analytical solution. Regularização = False
4.476971375975179
Casos. Regularização = False
Shape: (97,)
...
Labels. Regularização = False
Shape: (97,)
...Coeficientes da solução analitica. Regularização = False
Shape: (2,)
[-3.89578088  1.19303364]
Previsão para os casos de Treino usando o modelo. Regularização = False
(97,)
...
Model with analytical solution. Regularização = True
Cost function value for theta from analytical solution. Regularização = True
7.194901089799717
Casos. Regularização = True
Shape: (97,)
...
Labels. Regularização = True
Shape: (97,)
...
Coeficientes da solução analitica. Regularização = True
Shape: (2,)
[5.83913505 2.26154817]
Previsão para os casos de Treino usando o modelo. Regularização = True
(97,)
...
```





Class `LinearRegression`

Testing the class

```
def test_2var_4(regul = False):
...

test_2var_4(False)  #sem regularização
test_2var_4(True)   #com regularização: standardization = True, regularization = True, lamda = 1

def test_2var_5(x, regul = False):
...

test_2var_5(7, False)  #sem regularização
test_2var_5(7, True)   #com regularização: standardization = True, regularization = True, lamda = 1

def test_multivar():
...
test_multivar()
```

Class `LogisticRegression`

Methods:

- `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 cost function value (for all dataset examples)
- `def gradientDescent(self, dataset, alpha = 0.01, iters = 10000)` – creates the model using gradient descent

Class LogisticRegression

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

class LogisticRegression:

    def __init__(self, dataset, normalize = False, regularization = False, lamda = 1):
        self.X, self.y = dataset.getXy()
        self.X = np.hstack ( (np.ones([self.X.shape[0],1]), self.X ) )
        self.theta = np.zeros(self.X.shape[1])
        self.regularization = regularization
        self.lamda = lamda
        if normalize:
            self.normalize()
        else:
            self.normalized = False
```

Class LogisticRegression

```
class LogisticRegression:


    def normalize(self):
        self.mu = np.mean(self.X[:,1:], axis = 0)
        self.X[:,1:] = self.X[:,1:] - self.mu
        self.sigma = np.std(self.X[:,1:], axis = 0)
        if np.all(self.sigma!= 0): self.X[:,1:] = self.X[:,1:] / self.sigma
        self.normalized = True

    def printCoefs(self):
        print(self.theta)

    def predict (self, instance):
        p = self.probability(instance)
        if p >= 0.5: res = 1
        else: res = 0
        return res
```

Class LogisticRegression

```
class LogisticRegression:


    def probability(self, instance):
        x = np.empty([self.X.shape[1]])
        x[0] = 1
        x[1:] = np.array(instance[:self.X.shape[1]-1])
        # ...
        if self.normalized:
            if np.all(self.sigma!= 0):
                 x[1:] = (x[1:] - self.mu) / self.sigma
            else: x[1:] = (x[1:] - self.mu)
        return sigmoid ( np.dot(self.theta, x))
```

Class LogisticRegression

```
class LogisticRegression:


    def costFunction(self, theta = None):
        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) )
        return cost


    def buildModel(self, dataset):
        if self.regularization:
            self.optim_model()
        else:
            self.optim_model_reg(self.lamda)
```

```
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
```

Class LogisticRegression

```
class LogisticRegression:


    def gradientDescent(self, dataset, alpha = 0.01, iters = 10000):
        m = self.X.shape[0]
        n = self.X.shape[1]
        self.theta = np.zeros(n)
        for its in range(iters):
            J = self.costFunction()
            if its%1000 == 0: print(J)
            ...
            delta = self.X.T.dot(sigmoid(self.X.dot(self.theta)) - self.y)
            self.theta -= (alpha /m * delta )
```

Class LogisticRegression

```
class LogisticRegression:


def optim_model(self):
        from scipy import optimize
        n = self.X.shape[1]
        options = {'full_output': True, 'maxiter': 500}
        initial_theta = np.zeros(n)
        self.theta, _, _, _, _ = optimize.fmin(lambda theta: self.costFunction(theta), initial_theta,
**options)


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

Class LogisticRegression

```
class LogisticRegression:


    def mapX(self):
        self.origX = self.X.copy()
        mapX = mapFeature(self.X[:,1], self.X[:,2], 6)
        self.X = np.hstack((np.ones([self.X.shape[0],1]), mapX) )
        self.theta = np.zeros(self.X.shape[1])


    def mapFeature (X1, X2, degrees = 6):
        out = np.ones( (np.shape(X1)[0], 1) )
        for i in range(1, degrees+1):
            for j in range(0, i+1):
                term1 = X1 ** (i-j)
                term2 = X2 ** (j)
                term = (term1 * term2).reshape( np.shape(term1)[0], 1 )
                out = np.hstack(( out, term ))
        return out
```

Class LogisticRegression

```
class LogisticRegression:


    def plotData(self):
        import matplotlib.pyplot as plt
        negatives = self.X[self.y == 0]
        positives = self.X[self.y == 1]
        plt.xlabel("x1")
        plt.ylabel("x2")
        plt.xlim([self.X[:,1].min(), self.X[:,1].max()])
        plt.ylim([self.X[:,1].min(), self.X[:,1].max()])
        plt.scatter( negatives[:,1], negatives[:,2], c='r', marker='o', linewidths=1, s=40, label='y=0' )
        plt.scatter( positives[:,1], positives[:,2], c='k', marker='+', linewidths=2, s=40, label='y=1' )
        plt.legend()
        plt.show()
```

Class LogisticRegression

```
class LogisticRegression:


    def plotModel(self):
        import matplotlib.pyplot as plt
        from numpy import r_
        pos = (self.y == 1).nonzero()[:1]
        neg = (self.y == 0).nonzero()[:1]
        plt.plot(self.X[pos, 1].T, self.X[pos, 2].T, 'k+', markeredgewidth=2, markersize=7)
        plt.plot(self.X[neg, 1].T, self.X[neg, 2].T, 'ko', markerfacecolor='r', markersize=7)
        if self.X.shape[1] <= 3:
            plot_x = r_[self.X[:,2].min(),  self.X[:,2].max()]
            plot_y = (-1./self.theta[2]) * (self.theta[1]*plot_x + self.theta[0])
            plt.plot(plot_x, plot_y)
            plt.legend(['class 1', 'class 0', 'Decision Boundary'])
        plt.show()
```

Class LogisticRegression

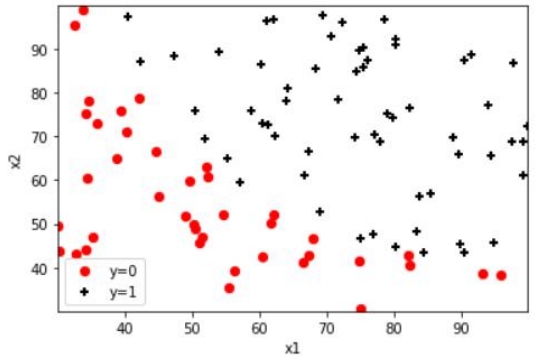
```
class LogisticRegression:


    def plotModel2(self):
        negatives = self.origX[self.y == 0]
        positives = self.origX[self.y == 1]
        plt.xlabel("x1"); plt.ylabel("x2")
        plt.xlim([self.origX[:,1].min(), self.origX[:,1].max()])
        plt.ylim([self.origX[:,1].min(), self.origX[:,1].max()])
        plt.scatter( negatives[:,1], negatives[:,2], c='r', marker='o', linewidths=1, s=40, label='y=0' )
        plt.scatter( positives[:,1], positives[:,2], c='k', marker='+', linewidths=2, s=40, label='y=1' )
        plt.legend()
        u = np.linspace( -1, 1.5, 50 )
        v = np.linspace( -1, 1.5, 50 )
        z = np.zeros( (len(u), len(v)) )
                                                for i in range(0, len(u)):
                                                          for j in range(0, len(v)):
                                                              x = np.empty([self.X.shape[1]])
                                                              x[0] = 1
                                                              mapped = mapFeature( np.array([u[i]]), np.array([v[j]]) )
                                                              x[1:] = mapped
                                                              z[i,j] = x.dot( self.theta )
                                                    z = z.transpose()
                                                    u, v = np.meshgrid( u, v )
                                                    plt.contour( u, v, z, [0.0, 0.001])
                                                    plt.show()
```

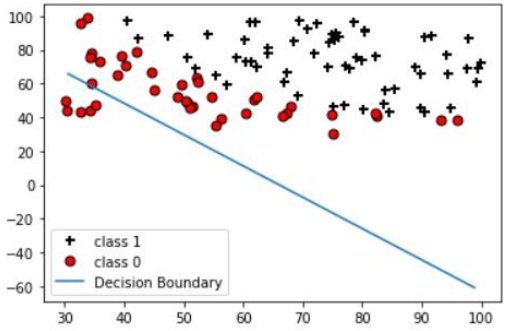
Class LogisticRegression

Testing the class with example `log-ex1.data` (2 variables)

```
def test():
    ds= Dataset("log-ex1.data")
    logmodel = LogisticRegression(ds)
    logmodel.plotData()
    print ("Initial cost: ", logmodel.costFunction())
    logmodel.gradientDescent(ds, 0.001, 20000)
    logmodel.plotModel()
    print ("Final cost:", logmodel.costFunction())
    ex = np.array([45,65])
    print ("Prob. example:", logmodel.probability(ex)
test()
```

```
> Initial cost:  [0.69314718 0.69314718 0.69314718 0.69314718 0.69314718

...
Final cost: [0.80142644 0.58235417 0.7850892
...

Prob. example: 0.5673933354864463
```





Extra Exercise

- Run version of logistic regression with regularization – `testreg()`
- Generate a graph that shows the discrimination generated by the algorithm
- Analyze various values of the regularization parameter (e.g. 0, 1, 10, 100, …)

Extra Exercise

- Run version of logistic regression with regularization – `testreg()`
- Generate a graph that shows the discrimination generated by the algorithm
- Analyze various values of the regularization parameter (e.g. 0, 1, 10, 100, …)