HW1A_Classification_NBA

March 14, 2020

1 Classification of NBA players role

1.1 IMPORTANT: make sure to rerun all the code from the beginning to obtain the results for the final version of your notebook, since this is the way we will do it before evaluting your notebook! Only the notebook file will be submitted, do not change the data filenames, file content or paths we'll use the provided ones.

Student name: Alessandro Valente ID Number: 1234429

1.1.1 Dataset description

We will be working with a dataset of NBA basketball players data (you can get from https://www.kaggle.com/jacobbaruch/nba-player-of-the-week the full dataset).

The provided data is a subset of the full dataset containing the players that have the role of Center and of Point Guard. For each player the dataset contains the height, weight and age.

From Wikipedia:

The **Center** (C), also known as the five, or the big man, is one of the five positions in a regular basketball game. The center is normally the tallest player on the team, and often has a great deal of strength and body mass as well. In the NBA, the center is usually 6′ 10" (2.08 m) or taller and usually weighs 240 lbs (109 kg) or more.

Point Guards (PG, a.k.a. as "play maker") are expected to run the team's offense by controlling the ball and making sure that it gets to the right players at the right time. In the NBA, point guards are usually about 6′ 3" (1.93 m) or shorter, and average about 6′ 2" (1.88 m). Having above-average size (height, muscle) is considered advantageous, although size is secondary to situational awareness, speed, quickness, and ball handling skills. Shorter players tend to be better dribblers since they are closer to the floor, and thus have better control of the ball while dribbling.

As it is clear from the description, the height and weight of the player are good hints to predict their role and in this lab we will exploit this features to estimate the role.

1.1.2 Three features (regressors) are considered for this dataset:

1) Height in cm

2) Weight in kg

3) Age in years

We first import all the packages that are needed.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import csv

        import random
        import numpy as np
        import scipy as sp
        import sklearn as sl
        from scipy import stats
        from sklearn import datasets
        from sklearn import linear_model
```

2 Perceptron

We will implement the perceptron algorithm and use it to learn a halfspace.

TO DO Set the random seed (you can use your ID (matricola) or any other number!).

```
In [2]: IDnumber = 1234429#YOUR_ID , try also to change the seed to see the impact of random i
        np.random.seed(IDnumber)
```

Load the dataset and then split in training set and test set (the training set is typically larger, you can use a 75% tranining 25% test split) after applying a random permutation to the datset.

A) Load dataset and perform permutation

```
In [3]: #load the dataset
        filename = 'data/NBA.csv' # do not change the file name or content
        NBA = csv.reader(open(filename, newline=''), delimiter=',')

        header = next(NBA) #skip first line
        print(header)

        dataset = list(NBA)
        for i in range(len(dataset)):
            dataset[i] = [int(x) for x in dataset[i]]

        dataset = np.asarray(dataset)

        X = dataset[:,1:4] #columns 1,2,3 contain the features
        Y = dataset[:,0]  # column 0: labels

        Y = Y*2-1  # set labels to -1, 1 as required by perceptron implementation

        m = dataset.shape[0]
        permutation = np.random.permutation(m) # random permurtation

        X = X[permutation]
        Y = Y[permutation]
        #print (X, Y)
```

```
['Position', 'Height cm', 'kg', 'Age']
```

We are going to classify class "1" (Center) vs class "-1" (Point Guard)

B) **TO DO** Divide the data into training set and test set (1/4 of the data in the first set, 3/4 in the second one)

```
In [4]: #Divide in training and test: make sure that your training set
        #contains at least 10 elements from class 1 and at least 10 elements
        #from class -1! If it does not, modify the code so to apply more random
        #permutations (or the same permutation multiple times) until this happens.
        #IMPORTANT: do not change the random seed.

        #m_training needs to be the number of samples in the training set

        m_training = int(len(X)*3/4)# PLACE YOUR CODE

        #m_test needs to be the number of samples in the test set
        m_test = len(X)-m_training # PLACE YOUR CODE

        #X_training = instances for training set
        X_training = X[0:m_training,:]
        #Y_training = labels for the training set
        Y_training = Y[0:m_training]

        #X_test = instances for test set
        X_test =  X[m_training:,:]
        #Y_test = labels for the test set
        Y_test = Y[m_training:]

        print(Y_training) #to make sure that Y_training contains both 1 and -1
        print(m_test)

        print("Shape of training set: " + str(X_training.shape))
        print("Shape of test set: " + str(X_test.shape))
```

```
[-1 -1  1 -1  1  1 -1 -1  1  1 -1 -1 -1 -1  1 -1  1  1  1  1 -1 -1 -1 -1
 -1 -1  1 -1  1 -1  1 -1  1  1 -1  1  1  1  1 -1 -1  1  1  1  1  1 -1 -1
 -1 -1  1  1  1  1 -1 -1 -1  1  1  1 -1 -1  1  1  1  1 -1  1  1 -1 -1 -1
  1 -1 -1  1 -1 -1 -1  1 -1  1  1 -1  1  1  1 -1 -1 -1  1  1  1 -1  1  1
  1  1  1 -1  1 -1 -1  1 -1  1  1 -1 -1 -1 -1  1  1  1 -1 -1  1  1 -1  1
  1 -1  1 -1 -1 -1  1 -1  1 -1  1  1 -1  1  1  1  1  1  1  1  1  1  1 -1
 -1  1 -1  1  1  1  1  1 -1  1 -1  1  1  1 -1 -1 -1 -1 -1  1  1  1 -1  1
 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1  1  1 -1  1 -1 -1  1  1  1 -1 -1
  1 -1  1]
65
Shape of training set: (195, 3)
Shape of test set: (65, 3)
```

TO DO Now add a 1 in front of each sample so that we can use a vector in homogeneous coordinates to describe all the coefficients of the model. You can use the function *hstack* in *numpy*

```
In [5]: #add a 1 to each sample (homogeneous coordinates)
        X_training=np.c_[np.ones(int(m_training)),X_training]
        X_test=np.c_[np.ones(int(m_test)),X_test]

        print(X_training[0:10])
        print(X_test[0:10])
```

```
[[  1. 192.  76.  37.]
 [  1. 195.  94.  32.]
 [  1. 231. 139.  26.]
 [  1. 192.  76.  26.]
 [  1. 216. 112.  31.]
 [  1. 216. 146.  21.]
 [  1. 182.  76.  29.]
 [  1. 192.  90.  28.]
 [  1. 219. 117.  29.]
 [  1. 186. 103.  27.]]
[[  1. 213. 114.  25.]
 [  1. 216. 112.  29.]
 [  1. 195.  94.  27.]
 [  1. 213. 114.  32.]
 [  1. 182.  78.  25.]
 [  1. 216. 112.  30.]
 [  1. 195.  94.  29.]
 [  1. 186. 121.  25.]
 [  1. 195.  94.  25.]
 [  1. 185.  76.  28.]]
```

TO DO Now complete the function *perceptron*. Since the perceptron does not terminate if the data is not linearly separable, your implementation should return the desired output (see below) if it reached the termination condition seen in class or if a maximum number of iterations have already been run, where one iteration corresponds to one update of the perceptron weights. In case the termination is reached because the maximum number of iterations have been completed, the implementation should return **the best model** seen up to now.

The input parameters to pass are: - X: the matrix of input features, one row for each sample - Y: the vector of labels for the input features matrix X - *max_num_iterations*: the maximum number of iterations for running the perceptron

The output values are: - *best_w*: the vector with the coefficients of the best model - *best_error*: the *fraction* of missclassified samples for the best model

```
In [6]: w=np.zeros(len(X_training[0]))
        print(w)
```

```
[0. 0. 0. 0.]
```

```
In [7]: def percep_err (X, Y, w):
            return np.sum(np.multiply(Y, np.dot(X,w))<=0, axis=0)

        def perceptron(X, Y, max_num_iterations):
            w=best_w=np.zeros(len(X[0]))
            error=best_error=percep_err (X, Y, w)
            for t in range( max_num_iterations):
                r=random.randint(0,len(Y)) #start computing at random place in the array
                for i in range (r,len(Y)+r) :
                    if i>=len(Y):
                        i-=len(Y) #to avoid out of bound situation
                    if Y[i]*np.dot(w, X[i])<=0:
                        w=w+Y[i]*X[i]
                        error=percep_err(X, Y, w) #update
                        if best_error>=error: #check for best
                            best_w=w
                            best_error=error
                        break


          #  for i in range(len(X)):
           #     if Y[i]*np.dot(w, X[i])<=0:
           #         best_error+=1.0

            return best_w, best_error/len(Y)
```

Now we use the implementation above of the perceptron to learn a model from the training data using 100 iterations and print the error of the best model we have found.

```
In [8]: #now run the perceptron for 100 iterations
        w_found, error = perceptron(X_training,Y_training, 100)
        print("Training Error of perpceptron (100 iterations): " + str(error))
```

```
Training Error of perpceptron (100 iterations): 0.12307692307692308
```

TO DO use the best model *w_found* to predict the labels for the test dataset and print the fraction of missclassified samples in the test set (the test error that is an estimate of the true loss).

```
In [9]: #now use the w_found to make predictions on test dataset

        num_errors = percep_err(X_test, Y_test, w_found)
        true_loss_estimate = num_errors/m_test  # error rate on the test set
        #NOTE: you can avoid using num_errors if you prefer, as long as true_loss_estimate is
        print("Test Error of perpceptron (100 iterations): " + str(true_loss_estimate))
```

```
Test Error of perpceptron (100 iterations): 0.07692307692307693
```

TO DO [Answer the following] What about the difference betweeen the training error and the test error in terms of fraction of missclassified samples)? Explain what you observe. [Write the answer in this cell]

We can observe that the error on the test and training set are similar, we can then deduce that the model is able to represent the data efficiently also beacuse of the value obtained.

TO DO Copy the code from the last 2 cells above in the cell below and repeat the training with 5000 iterations. Then print the error in the training set and the estimate of the true loss obtained from the test set.

```
In [10]: #now run the perceptron for 5000 iterations here!


         w_found, error = perceptron(X_training,Y_training, 5000)
         print("Training Error of perpceptron (5000 iterations): " + str(error))
         num_errors = percep_err(X_test, Y_test, w_found)
         true_loss_estimate = num_errors/m_test  # error rate on the test set
         print("Test Error of perpceptron (5000 iterations): " + str(true_loss_estimate))
```

```
Training Error of perpceptron (5000 iterations): 0.08205128205128205
Test Error of perpceptron (5000 iterations): 0.06153846153846154
```

TO DO [Answer the following] What about the difference betweeen the training error and the test error in terms of fraction of missclassified samples) when running for a larger number of iterations ? Explain what you observe and compare with the previous case. [Write the answer in this cell]

The error on the training set is reduced doing 5000 iterations meaning that the model paramenter estimation improves as expected doing more iterations, besides the error on the test set is equal in both cases meaning that the improvement obtained in the 5000 iterations case is only a light improvement and so also the paramenter estimation with 100 iteration can be considered a good estimation of the model parameters.

3 Logistic Regression

Now we use logistic regression, as implemented in Scikit-learn, to predict labels. We will also plot the decision region of logistic regression.

We first load the dataset again.

```
In [11]: filename = 'data/NBA.csv'
         NBA = csv.reader(open(filename, newline=''), delimiter=',')

         header = next(NBA)
         print(header)

         dataset = list(NBA)
         for i in range(len(dataset)):
             dataset[i] = [int(x) for x in dataset[i]]

         dataset = np.asarray(dataset)

         X = dataset[:,1:]
         Y = dataset[:,0]

         Y = Y*2-1  # set labels to {-1, 1} as required by perceptron implementation

         m = dataset.shape[0]
         permutation = np.random.permutation(m)

         X = X[permutation]
         Y = Y[permutation]
```

```
['Position', 'Height cm', 'kg', 'Age']
```

TO DO As for the previous part, divide the data into training and test (75%-25%) and add a 1 as first component to each sample.

```
In [12]: #Divide in training and test: make sure that your training set
         #contains at least 10 elements from class 1 and at least 10 elements
         #from class -1! If it does not, modify the code so to apply more random
         #permutations (or the same permutation multiple times) until this happens.
         #IMPORTANT: do not change the random seed.

         m_training = int(len(X)*3/4)
         m_test = len(X)-m_training

         X_training = np.c_[np.ones(int(m_training)),X[0:m_training,:]]
         Y_training = Y[0:m_training]

         X_test = np.c_[np.ones(int(m_test)), X[m_training:,:]]
         Y_test = Y[m_training:]
```

To define a logistic regression model in Scikit-learn use the instruction
$linear\_model.LogisticRegression(C = 1e5)$
(C is a parameter related to *regularization*, a technique that we will see later in the course. Setting it to a high value is almost as ignoring regularization, so the instruction above corresponds to the logistic regression you have seen in class.)

To learn the model you need to use the $fit(...)$ instruction and to predict you need to use the $predict(...)$ function. See the Scikit-learn documentation for how to use it.

TO DO Define the logistic regression model, then learn the model using the training set and predict on the test set. Then print the fraction of samples missclassified in the training set and in the test set.

```
In [13]: def bad_classified (A, B):
             return np.sum((A-B)!=0, axis=0)
```

```
In [14]: #part on logistic regression for 2 classes
         logreg = linear_model.LogisticRegression(C=1e5)

         #learn from training set
         logreg.fit(X_training, Y_training)

         #predict on training set
         prediction_training = logreg.predict(X_training)

         #print the error rate = fraction of missclassified samples
         error_rate_training = bad_classified (prediction_training, Y_training)/len(Y_training

         print("Error rate on training set: "+str(error_rate_training))

         #predict on test set

         prediction_test = logreg.predict(X_test)

         #print the error rate = fraction of missclassified samples
         error_rate_test = bad_classified (prediction_test, Y_test) /len(Y_test)

         print("Error rate on test set: " + str(error_rate_test))
```

```
Error rate on training set: 0.07692307692307693
Error rate on test set: 0.06153846153846154


/home/alessandro/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:433: Fu
  FutureWarning)
```

TO DO Now pick two features and restrict the dataset to include only two features, whose indices are specified in the *feature* vector below. Then split into training and test. Which features are you going to select ?

```
In [15]: #to make the plot we need to reduce the data to 2D, so we choose two features
         features_list = ['height', 'weight', 'age']
         labels_list = ['Center', 'Point guard']

         index_feature1 = 0 # select a feature
         index_feature2 = 1# select a feature
         features = [index_feature1, index_feature2]

         feature_name0 = features_list[features[0]]
         feature_name1 = features_list[features[1]]

         X_reduced = X[:,features]

         #X_training = np.c_[np.ones(int(m_training)),X_reduced[0:m_training,:]]
```

```
        X_training = X_reduced[0:m_training,:]
        Y_training = Y[0:m_training]

        #X_test = np.c_[np.ones(int(m_test)), X_reduced[m_training:,:]]
        X_test = X_reduced[m_training:,:]
        Y_test = Y[m_training:]

        print(X_training[0:10])
```

```
[[216 112]
 [185  88]
 [185 126]
 [185  88]
 [186 119]
 [210 114]
 [210 117]
 [192  76]
 [195  94]
 [186 121]]
```

Now learn a model using the training data and measure the performances.

```
In [16]: # learning from training data
        logreg = linear_model.LogisticRegression(C=1e5)
        logreg.fit(X_training, Y_training)
        prediction_training = logreg.predict(X_training)
        error_rate_training = bad_classified (prediction_training, Y_training)/len(Y_training
        print("Error rate on training set: "+str(error_rate_training))

        #predict on test set
        prediction_test = logreg.predict(X_test)
        error_rate_test = bad_classified (prediction_test, Y_test) /len(Y_test)
        print("Error rate on test set: " + str(error_rate_test))
```

```
Error rate on training set: 0.07179487179487179
Error rate on test set: 0.06153846153846154
```

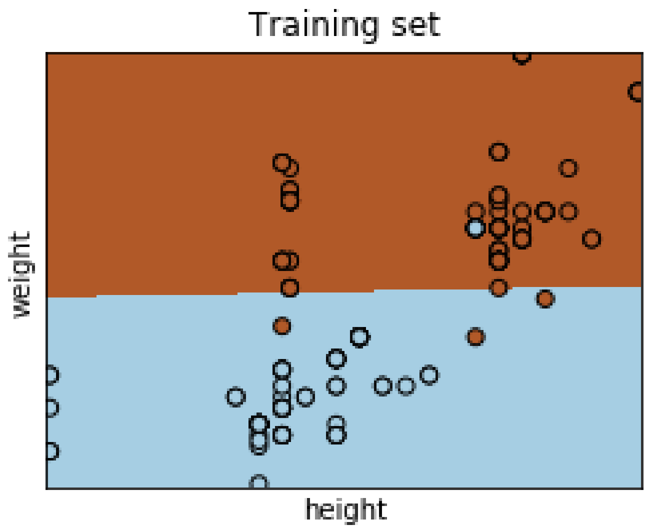
```
/home/alessandro/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:433: F
  FutureWarning)
```

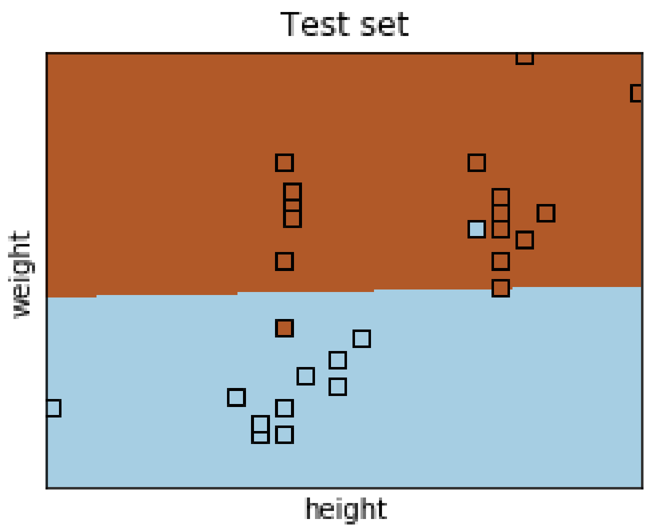
TO DO [Answer the following] Which features did you select and why ? Compare the perfromances with the ones of the case with all the 3 features and comment about the results. [Write the answer in this cell]

I selected weight and height as features for the reduced set, this choice was because the age is very less relevant in the role estimation of a player while the body structure (represented by height and weight) plays a crucial role, this hypothesis in confirmed by looking at the the error on the the reduced and full data set, since their value is identical we can deduce that the reduction of the data set excluding the age information does not harm the model capability of prediction, so the age information is not relevant for the model.

If everything is ok, the code below uses the model in *logreg* to plot the decision region for the two features chosen above, with colors denoting the predicted value. It also plots the points (with correct labels) in the training set. It makes a similar plot for the test set.

```
In [17]: # Plot the decision boundary. For that, we will assign a color to each
         # point in the mesh [x_min, x_max]x[y_min, y_max].

         # NOTICE: This visualization code has been developed for a "standard" solution of the
         # it could be necessary to make some fixes to adapt to your implementation

         h = .02  # step size in the mesh
         x_min, x_max = X_reduced[:, 0].min() - .5, X_reduced[:, 0].max() + .5
         y_min, y_max = X_reduced[:, 1].min() - .5, X_reduced[:, 1].max() + .5
         xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

         Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])

         # Put the result into a color plot
         Z = Z.reshape(xx.shape)

         plt.figure(1, figsize=(4, 3))
         plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

         # Plot also the training points
         plt.scatter(X_training[:, 0], X_training[:, 1], c=Y_training, edgecolors='k', cmap=pl
         plt.xlabel(feature_name0)
         plt.ylabel(feature_name1)

         plt.xlim(xx.min(), xx.max())
         plt.ylim(yy.min(), yy.max())
         plt.xticks(())
         plt.yticks(())
         plt.title('Training set')

         plt.show()

         # Put the result into a color plot
         Z = Z.reshape(xx.shape)
         plt.figure(1, figsize=(4, 3))
         plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

         # Plot also the test points
         plt.scatter(X_test[:, 0], X_test[:, 1], c=Y_test, edgecolors='k', cmap=plt.cm.Paired,
         plt.xlabel(feature_name0)
         plt.ylabel(feature_name1)
         plt.xlim(xx.min(), xx.max())
         plt.ylim(yy.min(), yy.max())
         plt.xticks(())
         plt.yticks(())
         plt.title('Test set')

         plt.show()
```





```
In [ ]:
```

```
In [ ]:
```