Regularized Classification on Student Alcohol Dataset

March 14, 2020

1 Regularized Classification on Student Alcohol Dataset

We are going to use a dataset from Kaggle (https://www.kaggle.com/uciml/student-alcohol-consumption)

1.0.1 Dataset description

> The data were obtained in a survey of students from the portuguese language courses in a secondary school. It contains a lot of interesting social, gender and study information about students.
>
> Have a look at the information about the dataset at the webpage: https://www.kaggle.com/uciml/student-alcohol-consumption
>
> In this context, we ask you to estimate which students are the most prone to alcohol consumption given some social and educational information.

2 TO DO: put your Surname, Name and ID number ("numero di matricola")

Student Name: Valente Alessandro
ID Number: 1234429

```
In [1]: import numpy as np

        # The seed will be used as seed for splitting the data into training and test.
        # You can place your ID or also try different seeds to see the impact of the random su
        # and of the random components in the algorithm on the results
        IDnumber = 1234429
        np.random.seed(IDnumber)

In [2]: # let's load library for plotting
        %matplotlib inline
        import matplotlib.pyplot as plt
```

2.1 Data Preprocessing

Load the data from a .csv file. In this notebook we use the pandas (Python Data Analysis Library) package, since it provides useful functions to clean the data. In particular, it allows us to remove samples with missing data, as we do below. We also plot some descriptions of columns, check the pandas documentation for 'describe()' if you want to know more.

```
In [3]: # let's load pands and numpy
        import pandas as pd
        import numpy as np

        # this time we use pandas to load and clean the dataset

        # read the data from the csv file
        df = pd.read_csv("data/student-data.csv", sep=',')

        # let's see some statistics about the data
        df.describe()
```

```
Out[3]:        drink_alcohol         age        Medu        Fedu  traveltime  \
        count     649.000000  649.000000  649.000000  649.000000  649.000000
        mean        0.454545   16.744222    2.514638    2.306626    1.568567
        std         0.498314    1.218138    1.134552    1.099931    0.748660
        min         0.000000   15.000000    0.000000    0.000000    1.000000
        25%         0.000000   16.000000    2.000000    1.000000    1.000000
        50%         0.000000   17.000000    2.000000    2.000000    1.000000
        75%         1.000000   18.000000    4.000000    3.000000    2.000000
        max         1.000000   22.000000    4.000000    4.000000    4.000000

                studytime      famrel    freetime       goout      health    absences  \
        count  649.000000  649.000000  649.000000  649.000000  649.000000  649.000000
        mean     1.930663    3.930663    3.180277    3.184900    3.536210    3.659476
        std      0.829510    0.955717    1.051093    1.175766    1.446259    4.640759
        min      1.000000    1.000000    1.000000    1.000000    1.000000    0.000000
        25%      1.000000    4.000000    3.000000    2.000000    2.000000    0.000000
        50%      2.000000    4.000000    3.000000    3.000000    4.000000    2.000000
        75%      2.000000    5.000000    4.000000    4.000000    5.000000    6.000000
        max      4.000000    5.000000    5.000000    5.000000    5.000000   32.000000

                    Marks
        count  649.000000
        mean    11.906009
        std      3.230656
        min      0.000000
        25%     10.000000
        50%     12.000000
        75%     14.000000
        max     19.000000
```

Now we create data matrices: many of the features are categorical, so we need to encode them with ***indicator matrices*** (i.e., using the so called one-hot encoding). That is, if a feature can take ℓ different values $v_1, \ldots, v_\ell$, we create ℓ indicator (0-1) features $I_1, \ldots, I_\ell$, such that $I_j = 1$ if and only if the value of the feature is v_j. This can be done in Python by first encode a feature with integers with LabelEncoder() and then obtain the indicator variables with OneHotEncoder().

```
In [4]: #df.values contains the data, both the values of instances and the value of the label
        Data = df.values
        # the matrix including the categorical data is given by columns from the second one
        X_categorical = Data[:,1:]
        # the target value (class) is in the first column
        Y = Data[:,0]

        print(list(df))

        # get the number d of features of each sample
        d = X_categorical.shape[1]

        # get the number m of samples
        m = X_categorical.shape[0]

        #let's see what the number of samples is
        print("Number of samples: {}".format(m))

        #now encode categorical variables using integers and one-hot-encoder

        from sklearn.preprocessing import LabelEncoder, OneHotEncoder
        label_encoder = LabelEncoder()
        onehot_encoder = OneHotEncoder(categories='auto')

        # encode the first column of the data matrix into indicator variables

        X_tmp = label_encoder.fit_transform(X_categorical[:,1])
        X_tmp = X_tmp.reshape(X_tmp.shape[0],1)
        X = onehot_encoder.fit_transform(X_tmp[:,0].reshape(-1,1)).toarray()
        print("Categorical feature:", df.columns[1], "   Number of categories:", X[1,:].shape)

        # repeat for the other categorical input variables
        index_categorical = [1,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

        for i in range(1,19):
            if i in index_categorical:
                X_tmp = label_encoder.fit_transform(X_categorical[:,i])
                X_tmp = X_tmp.reshape(X_tmp.shape[0],1)
                X_tmp = onehot_encoder.fit_transform(X_tmp[:,0].reshape(-1,1)).toarray()
                X = np.hstack((X,X_tmp))
                print("Categorical feature:", df.columns[i+1], "   Number of categories:", X_tr
            else:
                X_tmp = X_categorical[:,i]
                X_tmp = X_tmp.reshape(X_tmp.shape[0],1)
                X = np.hstack((X,X_tmp))
                print("Valued feature:", df.columns[i+1])

        print("Shape of X:", X.shape)
        print("Sample element from X:", X[20,:])
```

```
['drink_alcohol', 'school', 'sex', 'age', 'address', 'famsize', 'Pstatus', 'Medu', 'Fedu', 'Mj
Number of samples: 649
Categorical feature: school    Number of categories: (2,)
Categorical feature: sex    Number of categories: (2,)
Valued feature: age
Categorical feature: address    Number of categories: (2,)
Categorical feature: famsize    Number of categories: (2,)
Categorical feature: Pstatus    Number of categories: (2,)
Categorical feature: Medu    Number of categories: (5,)
Categorical feature: Fedu    Number of categories: (5,)
Categorical feature: Mjob    Number of categories: (5,)
Categorical feature: Fjob    Number of categories: (5,)
Categorical feature: guardian    Number of categories: (3,)
Categorical feature: traveltime    Number of categories: (4,)
Categorical feature: studytime    Number of categories: (4,)
Categorical feature: famrel    Number of categories: (5,)
Categorical feature: freetime    Number of categories: (5,)
Categorical feature: goout    Number of categories: (5,)
Categorical feature: health    Number of categories: (5,)
Valued feature: absences
Valued feature: Marks
Shape of X: (649, 64)
Sample element from X: [0.0 1.0 0.0 1.0 15 0.0 1.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
 0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0
 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 1.0 0.0
 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0 14]
```

2.2 Data Preprocessing

The class labels are already 0-1, so we can use them directly.

```
In [5]: # properly encode the target labels
        Y = label_encoder.fit_transform(Y)
        K = max(Y) + 1 # number of classes

        print("Number of classes: "+str(K))
```

```
Number of classes: 2
```

Given m total data points, keep $m\_training = 100$ data points as data for ***training and validation*** and $m\_test = m - m\_training$ as test data. Splitting is random, using as seed your ID number. Make sure that the training set contains at least 10 instances from each class.If it does not, modify the code so to apply a random permutation (or the same permutation multiple times) to the samples until this happens.

```
In [6]: # Split data into training and validation data

        # load a package which is useful for the training-test splitting
        # from sklearn.cross_validation import train_test_split
        from sklearn.model_selection import train_test_split

        # number of samples
        m = np.shape(X)[0]

        #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.

        permutation = np.random.permutation(m)
        X = X[permutation]
        Y = Y[permutation]

        m_training = 100 #  # use 100 samples for training + validation...
        m_test = m-m_training # and the rest for testing

        # test_size is the proportion of samples in the test set
        X_training, X_test, Y_training, Y_test = train_test_split(X, Y, test_size =float(m_tes

        print(Y_training)

        m_training = X_training.shape[0]
        m_test = X_test.shape[0]

        #let's see what the fraction of ones in the entire dataset is
        print(float(sum(Y_training)+sum(Y_test))/float(m_training+m_test))
```

```
[1 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 1 1 0 0 0 1 1 1 1 0 1 0 1
 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 0
 0 1 0 0 0 1 0 0 0 1 1 0 0 1 1 1 0 0 1 1 0 0 0 0 0 1]
0.45454545454545453
```

Standardize the data to have zero-mean and unit variance (columnwise):

```
In [7]: # Standardize the Features Matrix
        from sklearn import preprocessing
```

```
X = X.astype(np.float64) #standard scaler works with double precision data
X_training = X_training.astype(np.float64)
X_test = X_test.astype(np.float64)

#let's use the standard scaling; we degine the scaling for the entire dataset
scaler = preprocessing.StandardScaler().fit(X)

#let's apply the scaling to the training set

X_training = scaler.transform(X_training)
#let's apply the scaling to the test set

X_test = scaler.transform(X_test)
```

2.2.1 Perform Logistic Regression

We now perform logistic regression using the function provided by Scikit-learn.

Note: as provided by Scikit-learn, logistic regression is always implemented using regularization. However, the impact of regularization can be dampened to have almost no regularization by changing the parameter C, which is the inverse of λ. Therefore to have no regularization, which is $\lambda = 0$ for the model seen in class, we need C to have a large value. Here we fix $C = 100000000$.

[Note that the intercept is estimated in the model.]

For all our models we are going to use 10-fold cross validation to estimate the parameters (when needed) and/or estimate the validation error.

```
In [8]: from sklearn import linear_model

        # define a logistic regression model with very high C parameter -> low impact from reg
        # there are many solvers available to obtain the solution to the logistic regression p
        # one of them; 'cv' is the number of folds in cross-validation; we also specify l2 as
        # just to pick one; Cs contains the values of C to be tested and to pick from with val
        # are interested in only 1 value of C, and use cross-validation just to estimate the v
        # in a same way as other models

        reg = linear_model.LogisticRegressionCV(Cs=[100000000], solver='newton-cg',cv=10, pena

        #fit the model on training data
        reg.fit(X_training, Y_training)

        # the attribute 'Cs_' contains ALL the values of C evaluated in cross-validation;
        # let's print them
        print("Values of parameter C tried in 10-fold Cross-Validation: {}".format( reg.Cs_ ))

        # the attribute 'scores_' contains the accuracy obtained in each fold, for each value
        # of C tried; we now compute the average accuracy across the 10 folds

        CV_accuracies = np.divide(np.sum(reg.scores_[1],axis=0),10)
        # let's print the average accuracies obtained for the various values of C

        print("Accuracies obtained for the different values of C with 10-fold Cross-Validation

        # the attribute 'C_' contains the best value of C as identified by cross-validation;
        # let's print it

        print("Best value of parameter C according to 10-fold Cross-Validation: {}".format( re

        # let's store the best CV accuracy, and then print it
        print(type(reg.scores_))
        reg_best_CV_accuracy = max(reg.scores_[1])
        print("10-fold Cross-Validation accuracies obtained with the best value of parameter C
```

```
Values of parameter C tried in 10-fold Cross-Validation: [100000000]
Accuracies obtained for the different values of C with 10-fold Cross-Validation: [0.55212121]
Best value of parameter C according to 10-fold Cross-Validation: 100000000
<class 'dict'>
10-fold Cross-Validation accuracies obtained with the best value of parameter C: [0.8]
```

Note that the logistic regression function in Scikit-learn has many optional parameters. Read the documentation if you want to understand what they do!

2.3 TODO 1

2.3.1 Learn the best model from Logistic Regression on the entire training set and examine coefficients (by printing and plotting them)

Note that you can use simply *linear_model.LogisticRegression*(), that does not use cross-validation, without passing the best value of *C* (and then fit()).

```
In [9]: #Use a large C to disable regularization
        reg_full = linear_model.LogisticRegression(C=10000000)
        reg_full.fit(X_training, Y_training)


        # print the coefficients from the logistic regression model.
        print("Coefficients obtained using the entire training set: {}".format( reg_full.coef_

        # note that the intercept is not in coef_, it is in intercept_

        print("Intercept: {}".format( reg_full.intercept_ ))

        print("total number coef:  ",len(reg_full.coef_[0]) )
        print("below 1e0  coef:    ",len(reg_full.coef_[abs(reg_full.coef_)<1e0]))
        print("below 1e-1 coef:    ",len(reg_full.coef_[abs(reg_full.coef_)<1e-1]))
        print("below 1e-2 coef:    ",len(reg_full.coef_[abs(reg_full.coef_)<1e-2]))
        print("below 1e-3 coef:    ",len(reg_full.coef_[abs(reg_full.coef_)<1e-3]))
```