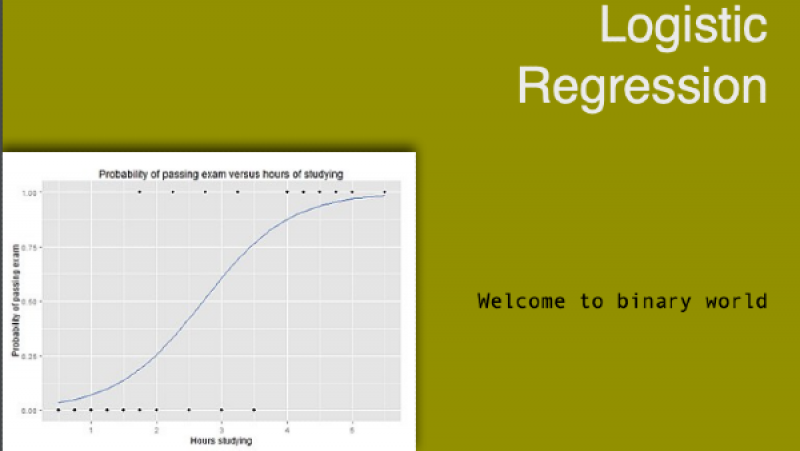
**Logistic Regression With Python**



* **338ACTIVITIES**
* **20SPOTCOINS**

Sep 13, 2019

**The objective of this study is to build logistic regression model, that can predict whether a subject has diabetes or not, given the values of certain diagnostic measurements.**

**Logistic Regression**

The following is the dataset we will be using to learn logistic regression in python.

**Pima-Indians-Diabetes**

preg plas pres skin test mass pedi age class

0 6 148 72 35 0 33.6 0.627 50 1

1 1 85 66 29 0 26.6 0.351 31 0

2 8 183 64 0 0 23.3 0.672 32 1

3 1 89 66 23 94 28.1 0.167 21 0

4 0 137 40 35 168 43.1 2.288 33 1

5 5 116 74 0 0 25.6 0.201 30 0

6 3 78 50 32 88 31.0 0.248 26 1

7 10 115 0 0 0 35.3 0.134 29 0

8 2 197 70 45 543 30.5 0.158 53 1

9 8 125 96 0 0 0.0 0.232 54 1

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. All patients here are females at least 21 years old from Pima Indian heritage. Please download the dataset from the link: <https://www.kaggle.com/uciml/pima-indians-diabetes-database>

The dataset consists of 9 columns and 768 observations. Columns are namely preg, plas, pres, skin, test, mass, pedi, age and class. First 8 columns are input variables or independent variables or features. The last column, that is the class variable is a dependent variable, which is 1 if a subject has diabetes, 0 otherwise.

1. Number of times the subject was pregnant,
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test,
3. Diastolic blood pressure (mm Hg),
4. Triceps skin fold thickness (mm),
5. 2-Hour serum insulin (mu U/ml),
6. Body mass index (weight in kg/(height in m)^2),
7. Diabetes pedigree function,
8. Age (years),
9. Class variable (0 or 1)

The dataset contains information for 500 non diabetic subjects and 268 diabetic patients.

**Let's restate the objective. Our objective is to build a logistic regression model, that will predict the class variable (that means whether a subject has Diabetes or not) of the subject based on the observation in 8 features.**

We will be using sklearn.linear\_model.**LogisticRegression** from scikit-learn for this model.

In [1]:

from sklearn.linear\_model import LogisticRegression

Logistic regression, despite its name, is a linear model for classification rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

We are now ready to build and train the logistics regression model. Let us first load the dataset. Variable X contains 8 features and variable Y contains class information, as explained earlier.

In [2]:

from pandas import read\_csv

filename = '/common/AI & ML/data/pima-indians-diabetes.csv'

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = read\_csv(filename, names=names)

array = dataframe.values

X = array[:,0:8]

Y = array[:,8]

The following section will split the dataset randomly into two groups, training dataset and test dataset. We will use 70% data as training data and remaining 30% as test data.

In [3]:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=1234)

Let's create an instance of LogisticRegression:

In [4]:

model = LogisticRegression()

**Description of parameters**

* penalty : str, ‘l1’, ‘l2’, ‘elasticnet’ or ‘none’, optional (default=’l2’), Used to specify the norm used in the penalization. The ‘newton-cg’, ‘sag’ and ‘lbfgs’ solvers support only l2 penalties. ‘elasticnet’ is only supported by the ‘saga’ solver. If ‘none’ (not supported by the liblinear solver), no regularization is applied.
* dual : bool, optional (default=False), Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n\_samples > n\_features.
* tol : float, optional (default=1e-4), Tolerance for stopping criteria.
* C : float, optional (default=1.0), Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.
* fit\_intercept : bool, optional (default=True), Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
* intercept\_scaling : float, optional (default=1), Useful only when the solver ‘liblinear’ is used and self.fit\_intercept is set to True. In this case, x becomes [x, self.intercept\_scaling], i.e. a “synthetic” feature with constant value equal to intercept\_scaling is appended to the instance vector. The intercept becomes intercept\_scaling \* synthetic\_feature\_weight.
* class\_weight : dict or ‘balanced’, optional (default=None), Weights associated with classes in the form {class\_label: weight}. If not given, all classes are supposed to have weight one. The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* np.bincount(y)).
* random\_state : int, RandomState instance or None, optional (default=None), The seed of the pseudo random number generator to use when shuffling the data. If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when solver == ‘sag’ or ‘liblinear’.
* solver : str, {‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’}, optional (default=’liblinear’). Algorithm to use in the optimization problem.
* For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and ‘saga’ are faster for large ones. For multiclass problems, only ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ handle multinomial loss; ‘liblinear’ is limited to one-versus-rest schemes. ‘newton-cg’, ‘lbfgs’, ‘sag’ and ‘saga’ handle L2 or no penalty ‘liblinear’ and ‘saga’ also handle L1 penalty ‘saga’ also supports ‘elasticnet’ penalty ‘liblinear’ does not handle no penalty
* max\_iter : int, optional (default=100), Maximum number of iterations taken for the solvers to converge.
* multi\_class : str, {‘ovr’, ‘multinomial’, ‘auto’}, optional (default=’ovr’), If the option chosen is ‘ovr’, then a binary problem is fit for each label. For ‘multinomial’ the loss minimised is the multinomial loss fit across the entire probability distribution, even when the data is binary. ‘multinomial’ is unavailable when solver=’liblinear’. ‘auto’ selects ‘ovr’ if the data is binary, or if solver=’liblinear’, and otherwise selects ‘multinomial’.
* verbose : int, optional (default=0), For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.
* warm\_start : bool, optional (default=False), When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See the Glossary.
* n\_jobs : int or None, optional (default=None) Number of CPU cores used when parallelizing over classes if multi\_class=’ovr’”. This parameter is ignored when the solver is set to ‘liblinear’ regardless of whether ‘multi\_class’ is specified or not. None means 1 unless in a joblib.parallel\_backend context. -1 means using all processors.
* l1\_ratio : float or None, optional (default=None), The Elastic-Net mixing parameter, with 0 <= l1\_ratio <= 1. Only used if penalty='elasticnet'. Setting l1\_ratio=0 is equivalent to using penalty='l2', while setting l1\_ratio=1 is equivalent to using penalty='l1'. For 0 < l1\_ratio <1, the penalty is a combination of L1 and L2.

Now, let's build the model with training data.

In [5]:

model.fit(X\_train, Y\_train)

Out[5]:

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='warn',

n\_jobs=None, penalty='l2', random\_state=None, solver='warn',

tol=0.0001, verbose=0, warm\_start=False)

**We have now trained our model. Next, we need to test our model with the test data. The following section will test how efficient (or accurate) our model is.**

**What are we going to test?** We are going to test that for a given subject in the test data whether the predicted value of class variable (that is, whether a subject has diabetes or not) matches with the actual value of the class variable in the test data. The following piece of code shows how the class variable is predicted for a given set of test data.

In [6]:

import pandas as pd

import numpy as np

import random as rnd

rnd.seed(123458)

X\_new = X[rnd.randrange(X.shape[0])]

X\_new = X\_new.reshape(1,8)

YHat = model.predict(X\_new)

df = pd.DataFrame(X\_new, columns = names[:-1])

df["predicted"] = YHat

df.head(1)

Out[6]:

pregplaspresskintestmasspediagepredicted08.0196.076.029.0280.037.50.60557.01.0

The above table shows, the person is likely to be a diabetes patient.

The next step is to calculate the accuracy of the prediction. And we will do this using **accuracy\_score** function.

In [7]:

from sklearn.metrics import accuracy\_score

YHat = model.predict(X\_test)

# calculate accuracy

print (round(accuracy\_score(Y\_test, YHat)\*100,2))

75.76

We can see that, accuracy of our model is approximately 76%.

So for a new subject, we can predict her diabetes status with 76% accuracy, which is the accuracy of our model.