LogisticRegression-breastCancerData

March 7, 2022

Implement logistic regression from scratch using python and numpy to a binary classification problem.
three types of problems: classification(catagorical), regression(numerical), and cluster(groups).
Logistic regression, while it has a regression in its name is an algorithm for solving classification problems not regression problems.

Dataset= breast cancer dataset : This is a binary classification problem i.e., each data point in the training data belong to one of two classes.

[110]:
```
#1
#import all the important libraries
# organize imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
```

[111]:
```
#2
# ignore all warnings
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

[112]:
```
#3
# load scikit-learn's breast cancer dataset
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()

print(data.keys())

#formate example(placeholder in {}): txt1 = "My name is: {fname}, I'm {age}".
 ↪format(fname = "John", age = 36)
print("No.of.data points (rows) : {}".format(len(data.data)))
print("No.of.features (columns) : {}".format(len(data.feature_names)))
print("No.of.classes            : {}".format(len(data.target_names)))
print("Class names              : {}".format(list(data.target_names)))
```

```python
# view the datatype of each column
df = pd.DataFrame(data.data)
print(df.dtypes)
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename'])
No.of.data points (rows) : 569
No.of.features (columns) : 30
No.of.classes            : 2
Class names              : ['malignant', 'benign']
0     float64
1     float64
2     float64
3     float64
4     float64
5     float64
6     float64
7     float64
8     float64
9     float64
10    float64
11    float64
12    float64
13    float64
14    float64
15    float64
16    float64
17    float64
18    float64
19    float64
20    float64
21    float64
22    float64
23    float64
24    float64
25    float64
26    float64
27    float64
28    float64
29    float64
dtype: object
```

[113]:
```python
print(data.DESCR)
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------
```

```
**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)

        The mean, standard error, and "worst" or largest (mean of the three
        worst/largest values) of these features were computed for each image,
        resulting in 30 features.  For instance, field 0 is Mean Radius, field
        10 is Radius SE, field 20 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign

    :Summary Statistics:

    ===================================== ====== ======
                                           Min    Max
    ===================================== ====== ======
    radius (mean):                        6.981  28.11
    texture (mean):                       9.71   39.28
    perimeter (mean):                     43.79  188.5
    area (mean):                          143.5  2501.0
    smoothness (mean):                    0.053  0.163
    compactness (mean):                   0.019  0.345
    concavity (mean):                     0.0    0.427
    concave points (mean):                0.0    0.201
    symmetry (mean):                      0.106  0.304
    fractal dimension (mean):             0.05   0.097
    radius (standard error):              0.112  2.873
    texture (standard error):             0.36   4.885
    perimeter (standard error):           0.757  21.98
    area (standard error):                6.802  542.2
    smoothness (standard error):          0.002  0.031
    compactness (standard error):         0.002  0.135
    concavity (standard error):           0.0    0.396
    concave points (standard error):      0.0    0.053
    symmetry (standard error):            0.008  0.079
    fractal dimension (standard error):   0.001  0.03
    radius (worst):                       7.93   36.04
    texture (worst):                      12.02  49.54
    perimeter (worst):                    50.41  251.2
    area (worst):                         185.2  4254.0
    smoothness (worst):                   0.071  0.223
    compactness (worst):                  0.027  1.058
    concavity (worst):                    0.0    1.252
    concave points (worst):               0.0    0.291
    symmetry (worst):                     0.156  0.664
    fractal dimension (worst):            0.055  0.208
    ===================================== ====== ======

    :Missing Attribute Values: None

    :Class Distribution: 212 - Malignant, 357 - Benign

    :Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

    :Donor: Nick Street

    :Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].
```

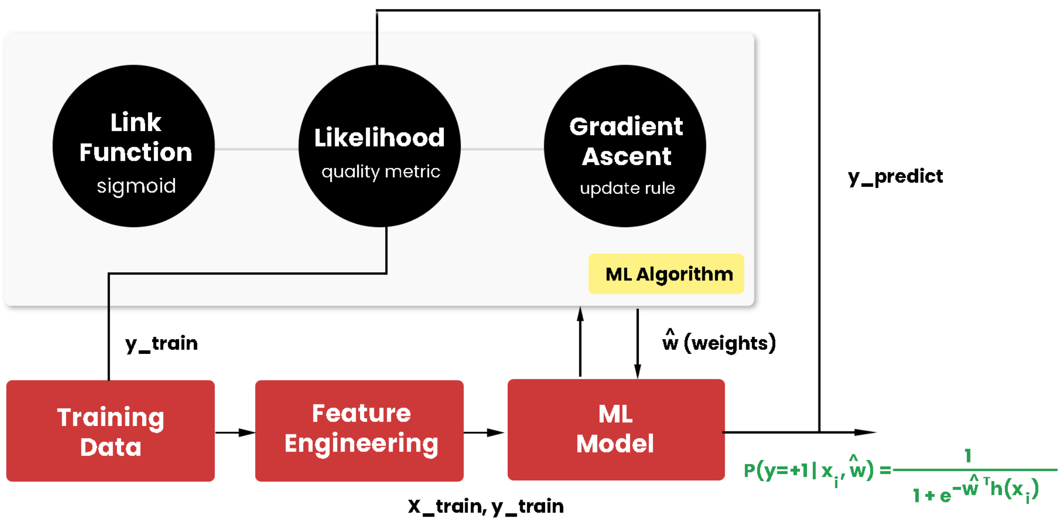
```
This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

.. topic:: References

   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction
     for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages 861-870,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and
     prognosis via linear programming. Operations Research, 43(4), pages
570-577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning
techniques
     to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77
(1994)
     163-171.
```

Target and feature variables: Target: The target is whatever the output of the input variables. It could be the individual classes that the input variables maybe mapped to in case of a classification problem or the output value range in a regression problem. If the training set is considered then the target is the training output values that will be considered. Targets are often manually labeled in a dataset, but there are ways to automate this process. Why are Target Variables Important? Without a labeled target, supervised machine learning algorithms would be unable to map available data to outcomes, just as a child would be incapable of figuring out that cats are called "cats" without having been told so at least a few times. It is important to have a well-defined target since the only thing an algorithm does is learn a function that maps relationships between input data and the target. The model's outcomes will be meaningless if your target doesn't make sense. Feature: Features are individual independent variables that act as the input in your system. Prediction models use features to make predictions. New features can also be obtained from old features using a method known as 'feature engineering'. More simply, you can consider one column of your data set to be one feature. Sometimes these are also called attributes. And the number of features are called dimensions.

Supervised Learning: train a model (set of algorithms) with the training data X_train along with its class names(labeled/target data) y_train and then use the trained model to predict the class y_test of unseen data point X_test(evaluate the model).

[114]:
```python
# load and show an image with Pillow
from PIL import Image
image = Image.open(r'C:\Users\nomaniqbal\Downloads\supervised.png')
image
```

[114]:



Logistic regression follows a beautiful procedure to learn from data. To learn means:
Weight: We define a weight value (parameter) for each feature (column) in the dataset.
Linear Predictor (score): We compute weighted sum for each data point in the dataset.
Link Function: We use a link function to transform that weighted sum to the probability range [0,1].
Log-Likelihood: We use the log-likelihood function as the quality metric to evaluate the prediction of the model i.e., how well the model has predicted y_predict when compared with ground truth y_train.
Gradient Ascent: We use gradient ascent algorithm to update the weights (parameters) by trying to maximize the likelihood.
Prediction: We take these learned weights and make predictions when new data point is given to the model.

Linear Predictor (score)
First, we define a weight value for each column (feature) in our dataset. As we have 30 features (columns) in the breast cancer dataset, we will have 30 weights [W1,W2...W30]. We compute the score (weighted sum) for each data point as follows.

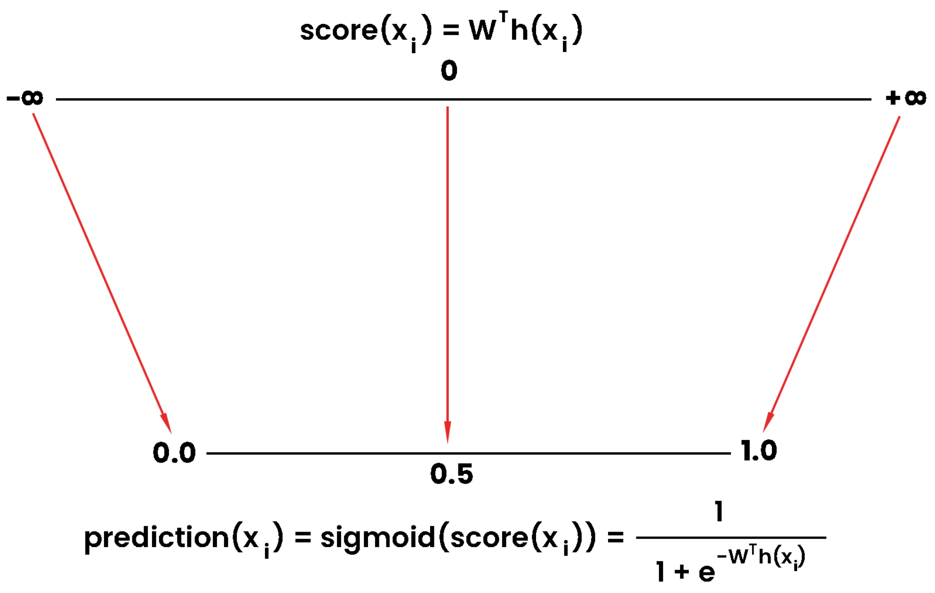
score=W0+(W1∗h1(xi))+(W2∗h2(xi))+...+(W30∗h30(xi))
=WT h(xi)

Notice we have W0 with no coefficient which is called the bias or intercept which must be learnt from the training data. As we have numeric values, the score for each data point might fall within a range [−∞,+∞]. Recall that our aim is to predict "given a new data point, tell me whether it's malignant (0) or benign (1)". This means, prediction from the ML model must be either 0 or 1. How are we going to achieve this? The answer is link function(sigmoid function).

Link Function If you give any input to a link function (say sigmoid), it transforms that input value to a range [0,1]. In our case, anything below 0.5 is assumed to be malignant-(of a disease) very virulent or infectious. (0), and anything above or equal to 0.5 is assumed to be benign-(of a disease) not harmful in effect. (1).

[115]:
```
#this code is just used for displaying the image
image = Image.open(r'C:\Users\nomaniqbal\Downloads\sigmoid.png')
image
```

[115]:



score(x_i) = W^T h(x_i)
0
-∞
+∞
0.0
0.5
1.0
prediction(x_i) = sigmoid(score(x_i)) = 1 / (1 + e^(-W^T h(x_i)))

[116]:
```
#5
#----------------------------------------------
# logistic regression without regularization
#----------------------------------------------

#to find the sigmoid value for a given input score.
def sigmoid(score):
    return (1 / (1 + np.exp(-score)))

def predict_probability(features, weights):
    score = np.dot(features, weights)
    return sigmoid(score)
#features, weights and score correspond to the matrices shown in the image␣
 ↪below.
```

[117]:
```
#this code is just used for displaying the image
image = Image.open(r'C:\Users\nomaniqbal\Downloads\expain.png')
image
```

[117]:

$$[features] = \begin{bmatrix} h(x_1)^T \\ h(x_2)^T \\ . \\ . \\ . \\ h(x_{569})^T \end{bmatrix} = \begin{bmatrix} h_0(x_1) & h_1(x_1) & . \; . \; . & h_{30}(x_1) \\ h_0(x_2) & h_1(x_2) & . \; . \; . & h_{30}(x_2) \\ . & . & . \; . \; . & . \\ . & . & . \; . \; . & . \\ . & . & . \; . \; . & . \\ h_0(x_{569}) & h_1(x_{569}) & . \; . \; . & h_{30}(x_{569}) \end{bmatrix}$$

$$\boxed{[score] = [features]\mathbf{w} = \begin{bmatrix} h(x_1)^T \\ h(x_2)^T \\ . \\ . \\ . \\ h(x_{569})^T \end{bmatrix} \mathbf{w} = \begin{bmatrix} h(x_1)^T\mathbf{w} \\ h(x_2)^T\mathbf{w} \\ . \\ . \\ . \\ h(x_{569})^T\mathbf{w} \end{bmatrix} = \begin{bmatrix} \mathbf{w}^T h(x_1) \\ \mathbf{w}^T h(x_2) \\ . \\ . \\ . \\ \mathbf{w}^T h(x_{569}) \end{bmatrix}}$$

But wait! how will the output value of this link function be the same as the ground truth value for a particular data point? It can't be as we are randomizing the weights for the features which will throw out some random value as the prediction. The whole point in learning algorithm is to adjust these weights based on the training data to arrive at a sweet spot that makes the ML model have low bias and low variance. Training the classifier = Learning the weight coefficients (with low bias and low variance). How do we adjust these weights? We need to define a quality metric that compares the output prediction of the ML model with the original ground truth class value. After evaluating the quality metric, we use gradient ascent algorithm to update the weights in a way that the quality metric reaches a global optimum value. Interesting isn't it?

Compute Likelihood How do we measure "how well the classifier fits the training data"? Using likelihood. We need to choose weight coefficients w. now, We define the below function to compute log-likelihood. Notice that we sum over all the training examples.

[118]:
```
#7

def compute_log_likelihood(features, labels, weights):
    indicators = (labels==+1)
    scores     = np.dot(features, weights)
    ll         = np.sum((np.transpose(np.array([indicators]))-1)*scores - np.
 ↪log(1. + np.exp(-scores)))
    return ll

def l2_compute_log_likelihood(features, labels, weights, l2_penalty):
    indicators = (labels==+1)
    scores      = np.dot(features, weights)
    ll          = np.sum((np.transpose(np.array([indicators]))-1)*scores - np.
 ↪log(1. + np.exp(-scores))) - (l2_penalty * np.sum(weights[1:]**2))
    return ll
```

Compute Derivative Once we have the log-likelihood equation, we can compute its derivative with respect to a single weight coefficient. We find the derivative of log-likelihood with respect to each of the weight coefficient w which in turn depends on its feature column. Notice that we sum over all the training examples, and the derivative that we return is a single number.

[119]:
```
#6
def feature_derivative(errors, feature):
    derivative = np.dot(np.transpose(errors), feature)
    return derivative
def l2_feature_derivative(errors, feature, weight, l2_penalty,␣
 ↪feature_is_constant):
    derivative = np.dot(np.transpose(errors), feature)
    if not feature_is_constant:
        derivative -= 2 * l2_penalty * weight
    return derivative
```

Gradient Ascent Now, we have all the ingredients to perform gradient ascent. Think of gradient ascent similar to hill-climbing. To reach the top of the hill (which is the global maximum), we choose a parameter called learning-rate. This defines the step-size that we need to take each iteration before we update the weight coefficients.

[120]:
```
#8
# logistic regression without L2 regularization
```

Split the dataset
To test our classifier's performance, we will split the original dataset into training and testing.

[121]:
```
#4
from sklearn.model_selection import train_test_split
# split the dataset into training and testing
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,␣
 ↪test_size=0.20, random_state=9)

#TRAIN VALUES ARE USED FOR MODEL TRAINING WHILE TEST VALUES ARE USED FOR MODEL␣
 ↪EVALUATION
#features/training data/predictors
print("X_train : " + str(X_train.shape))

#checking the results of machine learning for accuracy against the real world.
#Ground truth isn't true. It's an ideal expected result (according to the␣
 ↪people in charge).
```

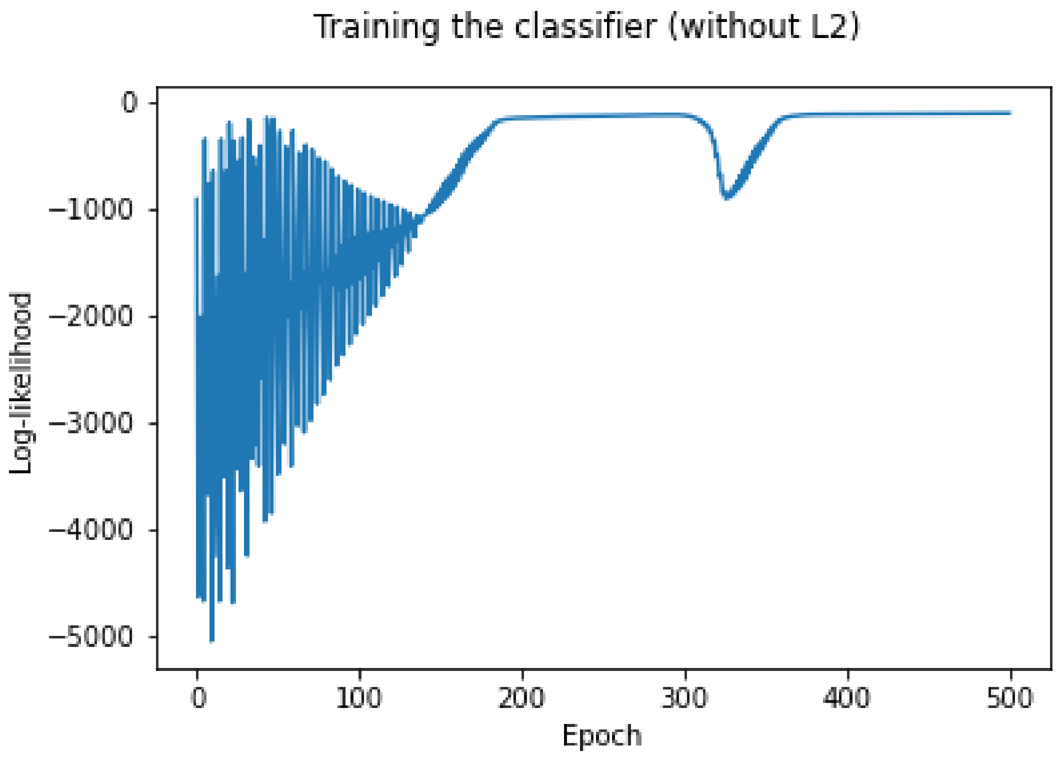
```
#For example, a set of images might be painstakingly hand-labeled as cat or␣
↪not-cat according
#to the opinions of whoever was in charge of the project and
#those cat/not-cat labels will be called "ground truth" for the project.
print("y_train : " + str(y_train.shape))

print("X_test : " + str(X_test.shape))
print("y_test : " + str(y_test.shape))
```

```
X_train : (455, 30)
y_train : (455,)
X_test : (114, 30)
y_test : (114,)
```

Train the classifier
As we already learnt, training the classifier means learning the weight coefficients. To train the classifier, we

Add intercept or bias to the feature matrix. Initialize the weight coefficients to zeros. Handpick the hyper-parameters learning rate and epochs. Use logistic_regression() function that we have just built and pass in the ingredients.

[122]:
```
#9
# hyper-parameters
learning_rate = 1e-7
epochs        = 500

# perform logistic regression
learned_weights = logistic_regression(X_train, y_train, learning_rate, epochs)
```



Test the classifier
To make predictions using the trained classifier, we use X_test data (testing data), learned_weights and predict_probability() function.

To find the accuracy between ground truth class values y_test and logistic regression predicted class values predictions, we use scikit-learn's accuracy_score() function.

[123]:
```
#10

from sklearn.metrics import accuracy_score
# make predictions using learned weights on testing data
bias_train     = np.ones((X_train.shape[0], 1))
bias_test      = np.ones((X_test.shape[0], 1))
features_train = np.hstack((bias_train, X_train))
features_test  = np.hstack((bias_test, X_test))
```

Reduce Overfitting
(when the model performs well with the new data but dont perform well with the new data)
Overfitting is a mandatory problem that we need to solve when it comes to machine learning. After training, we have the learned weight coefficients which must not overfit the training dataset.

When the decision boundary traced by the learned weight coefficients fits the training data extremely well, we have this overfitting problem. Often, overfitting is associated with very large estimated weight coefficients. This leads to overconfident predictions which is not very good for a real-world classifier.

To solve this, we need to measure the magnitude of weight coefficients. There are two approaches to measure it.

L1 norm: Sum of absolute value

w 1=|w0|+|w1|+|w2|...+|wN|

L2 norm: Sum of squares

w 22=w20+w21+w22...+w2N

L2 Regularization
We will use L2 norm (sum of squares) to reduce overshooting weight coefficients. It turns out that, instead of using likelihood function alone as the quality metric, what if we subtract w 22 from it, where is a hyper-parameter to control bias-variance tradeoff due to this regularization.

So, our new quality metric with regularization to combat overconfidence problem would be Large : High bias, low variance. Small : Low bias, high variance. Recall to perform gradient ascent, we need to know the derivative of quality metric to update the weight coefficients. Thus, the new derivative equation would be

l(w)/ wj−2 wj Let's understand the regularization impact on penalizing weight coefficients.

If wj>0, then −2 wj<0, thus it decreases wj>0 resulting in wj closer to 0. If wj<0, then −2 wj>0, thus it increases wj>0 resulting in wj closer to 0

When it comes to code, we need to update feature_derivative() function, compute_log_likelihood() function and logistic_regression() function with whatever we have learnt so far about L2 regularization as shown below.

[124]:
```
#-------------------------------------------------------
# Name      : Gogul Ilango
# Purpose   : Implement Logistic Regression from scratch
#             using python and numpy
# Variants  : 1. LR without L2 regularization
#             2. LR with L2 regularization
# Libraries : numpy, scikit-learn
#-------------------------------------------------------

# organize imports
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# ignore all warnings
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

```
# load scikit-learn's breast cancer dataset
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()

print(data.keys())
print("No.of.data points (rows) : {}".format(len(data.data)))
print("No.of.features (columns) : {}".format(len(data.feature_names)))
print("No.of.classes            : {}".format(len(data.target_names)))
print("Class names              : {}".format(list(data.target_names)))

# view the datatype of each column
df = pd.DataFrame(data.data)
print(list(df.dtypes))

# split the dataset into training and testing
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,␣
 ↪test_size=0.20, random_state=9)

print("X_train : " + str(X_train.shape))
print("y_train : " + str(y_train.shape))
print("X_test : " + str(X_test.shape))
print("y_test : " + str(y_test.shape))
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename'])
No.of.data points (rows) : 569
No.of.features (columns) : 30
No.of.classes            : 2
Class names              : ['malignant', 'benign']
[dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64'), dtype('float64'), dtype('float64'),
dtype('float64'), dtype('float64')]
X_train : (455, 30)
y_train : (455,)
X_test : (114, 30)
y_test : (114,)
```

[125]:
```
#-----------------------------------------------
# logistic regression without regularization
#-----------------------------------------------
def sigmoid(score):
        return (1 / (1 + np.exp(-score)))

def predict_probability(features, weights):
        score = np.dot(features, weights)
        return sigmoid(score)

def feature_derivative(errors, feature):
        derivative = np.dot(np.transpose(errors), feature)
        return derivative

def l2_feature_derivative(errors, feature, weight, l2_penalty,␣
↪feature_is_constant):
        derivative = np.dot(np.transpose(errors), feature)
        if not feature_is_constant:
                derivative -= 2 * l2_penalty * weight
        return derivative

def compute_log_likelihood(features, labels, weights):
        indicators = (labels==+1)
        scores     = np.dot(features, weights)
        ll         = np.sum((np.transpose(np.array([indicators]))-1)*scores -␣
↪np.log(1. + np.exp(-scores)))
        return ll

def l2_compute_log_likelihood(features, labels, weights, l2_penalty):
        indicators = (labels==+1)
        scores     = np.dot(features, weights)
        ll         = np.sum((np.transpose(np.array([indicators]))-1)*scores -␣
↪np.log(1. + np.exp(-scores))) - (l2_penalty * np.sum(weights[1:]**2))
        return ll
```

[126]:
```
# logistic regression without L2 regularization
def logistic_regression(features, labels, lr, epochs):

        # add bias (intercept) with features matrix
        bias      = np.ones((features.shape[0], 1))
        features  = np.hstack((bias, features))

        # initialize the weight coefficients
        weights = np.zeros((features.shape[1], 1))

        logs = []

        # loop over epochs times
        for epoch in range(epochs):

                # predict probability for each row in the dataset
                predictions = predict_probability(features, weights)

                # calculate the indicator value
                indicators = (labels==+1)

                # calculate the errors
                errors = np.transpose(np.array([indicators])) - predictions

                # loop over each weight coefficient
                for j in range(len(weights)):

                        # calculate the derivative of jth weight cofficient
                        derivative = feature_derivative(errors, features[:,j])
                        weights[j] += lr * derivative

                # compute the log-likelihood
                ll = compute_log_likelihood(features, labels, weights)
                logs.append(ll)

        import matplotlib.pyplot as plt
        x = np.linspace(0, len(logs), len(logs))
        fig = plt.figure()
        plt.plot(x, logs)
        fig.suptitle('Training the classifier (without L2)')
        plt.xlabel('Epoch')
        plt.ylabel('Log-likelihood')
        fig.savefig('train_without_l2.jpg')
        plt.show()

        return weights
```

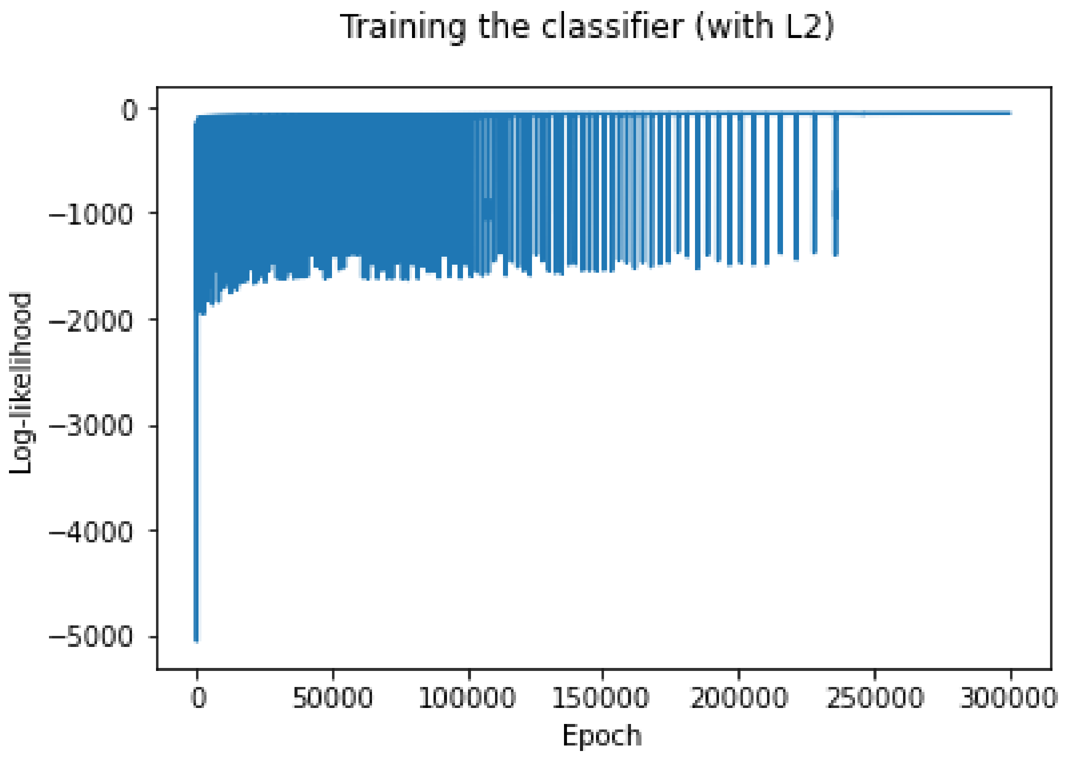
[127]:
```
# logistic regression with L2 regularization
def l2_logistic_regression(features, labels, lr, epochs, l2_penalty):

        # add bias (intercept) with features matrix
        bias      = np.ones((features.shape[0], 1))
        features  = np.hstack((bias, features))

        # initialize the weight coefficients
        weights = np.zeros((features.shape[1], 1))

        logs = []

        # loop over epochs times
        for epoch in range(epochs):

                # predict probability for each row in the dataset
                predictions = predict_probability(features, weights)

                # calculate the indicator value
                indicators = (labels==+1)

                # calculate the errors
                errors = np.transpose(np.array([indicators])) - predictions

                # loop over each weight coefficient
                for j in range(len(weights)):

                        isIntercept = (j==0)

                        # calculate the derivative of jth weight cofficient
                        derivative = l2_feature_derivative(errors, features[:
↪,j], weights[j], l2_penalty, isIntercept)
                        weights[j] += lr * derivative

                # compute the log-likelihood
                ll = l2_compute_log_likelihood(features, labels, weights,␣
↪l2_penalty)
                logs.append(ll)

        import matplotlib.pyplot as plt
        x = np.linspace(0, len(logs), len(logs))
        fig = plt.figure()
        plt.plot(x, logs)
        fig.suptitle('Training the classifier (with L2)')
        plt.xlabel('Epoch')
        plt.ylabel('Log-likelihood')
        fig.savefig('train_with_l2.jpg')
        plt.show()

        return weights
```

[128]:
```
# logistic regression without regularization
def lr_without_regularization():
        # hyper-parameters
        learning_rate = 1e-7
        epochs        = 500

        # perform logistic regression and get the learned weights
        learned_weights = logistic_regression(X_train, y_train, learning_rate,␣
↪epochs)

        # make predictions using learned weights on testing data
        bias_train     = np.ones((X_train.shape[0], 1))
        bias_test      = np.ones((X_test.shape[0], 1))
        features_train = np.hstack((bias_train, X_train))
        features_test  = np.hstack((bias_test, X_test))

        test_predictions  = (predict_probability(features_test,␣
↪learned_weights).flatten()>0.5)
        train_predictions = (predict_probability(features_train,␣
↪learned_weights).flatten()>0.5)
        print("Accuracy of our LR classifier on training data: {}".
↪format(accuracy_score(np.expand_dims(y_train, axis=1), train_predictions)))
        print("Accuracy of our LR classifier on testing data: {}".
↪format(accuracy_score(np.expand_dims(y_test, axis=1), test_predictions)))

        # using scikit-learn's logistic regression classifier
        model = LogisticRegression(random_state=9)
        model.fit(X_train, y_train)
        sk_test_predictions  = model.predict(X_test)
        sk_train_predictions = model.predict(X_train)
        print("Accuracy of scikit-learn's LR classifier on training data: {}".
↪format(accuracy_score(y_train, sk_train_predictions)))
        print("Accuracy of scikit-learn's LR classifier on testing data: {}".
↪format(accuracy_score(y_test, sk_test_predictions)))

        #visualize_weights(np.squeeze(learned_weights), 'weights_without_l2.
↪jpg')
```

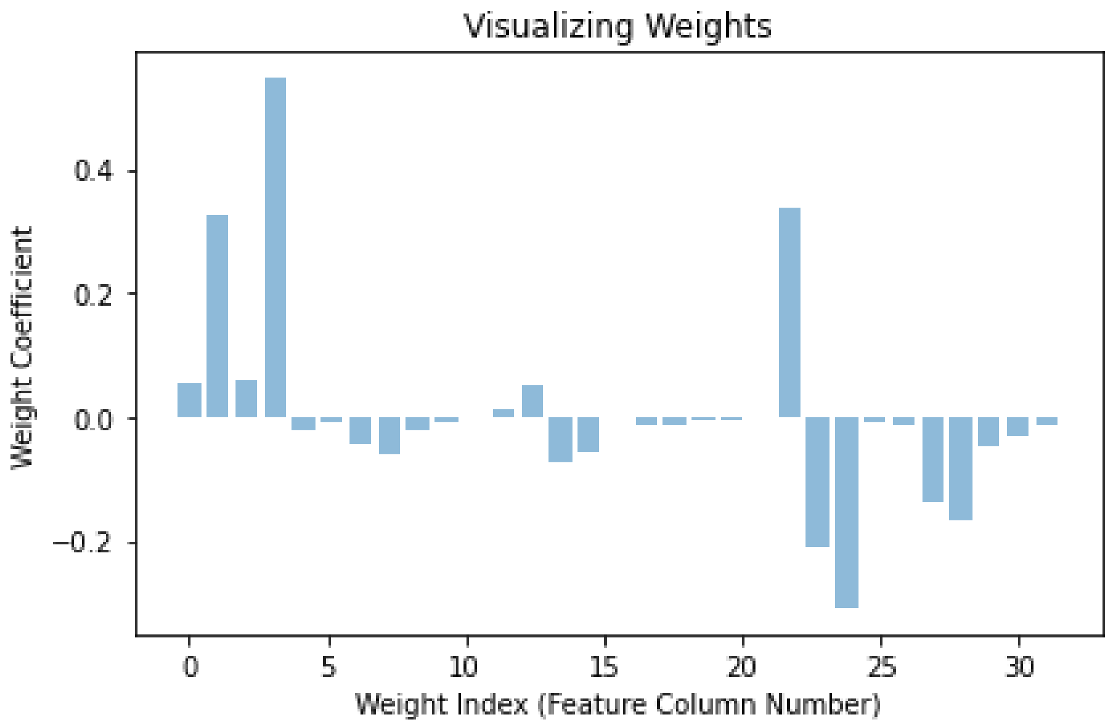
[129]:
```
# logistic regression with regularization
def lr_with_regularization():
        # hyper-parameters
        learning_rate = 1e-7
        epochs        = 300000
        l2_penalty    = 0.001

        # perform logistic regression and get the learned weights
        learned_weights = l2_logistic_regression(X_train, y_train,␣
↪learning_rate, epochs, l2_penalty)

        # make predictions using learned weights on testing data
        bias_train     = np.ones((X_train.shape[0], 1))
        bias_test      = np.ones((X_test.shape[0], 1))
        features_train = np.hstack((bias_train, X_train))
        features_test  = np.hstack((bias_test, X_test))

        test_predictions  = (predict_probability(features_test,␣
↪learned_weights).flatten()>0.5)
```

```
        train_predictions = (predict_probability(features_train,␣
↪learned_weights).flatten()>0.5)
        print("Accuracy of our LR classifier on training data: {}".
↪format(accuracy_score(np.expand_dims(y_train, axis=1), train_predictions)))
        print("Accuracy of our LR classifier on testing data: {}".
↪format(accuracy_score(np.expand_dims(y_test, axis=1), test_predictions)))

        # using scikit-learn's logistic regression classifier
        model = LogisticRegression(random_state=9)
        model.fit(X_train, y_train)
        sk_test_predictions  = model.predict(X_test)
        sk_train_predictions = model.predict(X_train)
        print("Accuracy of scikit-learn's LR classifier on training data: {}".
↪format(accuracy_score(y_train, sk_train_predictions)))
        print("Accuracy of scikit-learn's LR classifier on testing data: {}".
↪format(accuracy_score(y_test, sk_test_predictions)))

        visualize_weights(np.squeeze(learned_weights), 'weights_with_l2.jpg')
```

[130]:
```
# visualize weight coefficients
def visualize_weights(weights, title):
        import matplotlib.pyplot as plt
        x = np.linspace(0, len(weights), len(weights))

        fig = plt.figure()
        plt.bar(x, weights, align='center', alpha=0.5)
        plt.xlabel("Weight Index (Feature Column Number)")
        plt.ylabel("Weight Coefficient")
        plt.title('Visualizing Weights')
        plt.tight_layout()
        fig.savefig(title)

        plt.show()

lr_with_regularization()
```



```
Accuracy of our LR classifier on training data: 0.9406593406593406
Accuracy of our LR classifier on testing data: 0.9385964912280702
Accuracy of scikit-learn's LR classifier on training data: 0.9516483516483516
Accuracy of scikit-learn's LR classifier on testing data: 0.9473684210526315
```



1 A MORE EASIER IMPLEMENTATION:

Introducing the Breast Cancer Dataset

Now that we've built up the tools to build a Logistic Regression model for a BC dataset, we'll introduce a more easier way to deal with the dataset.

In the breast cancer dataset, each datapoint has measurements from an image of a breast mass and whether or not it's cancerous. The goal will be to use these measurements to predict if the mass is cancerous.

This dataset is built right into scikit-learn so we won't need to read in a csv. The object returned (which we stored in the cancer_data variable) is an object similar to a Python dictionary. We can see the available keys with the keys method. now as we learned, We'll start by looking at DESCR, which gives a detailed description of the dataset.

We can see there are 30 features, 569 datapoints, and target is either Malignant (cancerous) or Benign (not cancerous). For each of the datapoints we have measurements of the breast mass (radius, texture, perimeter, etc.). For each of the 10 measurements, multiple values were computed, so we have the mean, standard error and the worst value. This results in 10 * 3 or 30 total features. In the breast cancer dataset, there are several features that are calculated based on other columns. The process of figuring out what additional features to calculate is feature engineering.

[131]:
```
import pandas as pd
from sklearn.datasets import load_breast_cancer
```

```python
cancer_data = load_breast_cancer()
print(cancer_data.keys())
print(cancer_data['DESCR'])
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename'])
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
        - concavity (severity of concave portions of the contour)
        - concave points (number of concave portions of the contour)
        - symmetry
        - fractal dimension ("coastline approximation" - 1)

        The mean, standard error, and "worst" or largest (mean of the three
        worst/largest values) of these features were computed for each image,
        resulting in 30 features.  For instance, field 0 is Mean Radius, field
        10 is Radius SE, field 20 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign

    :Summary Statistics:

    ===================================== ====== ======
                                           Min    Max
    ===================================== ====== ======
    radius (mean):                        6.981  28.11
    texture (mean):                       9.71   39.28
    perimeter (mean):                     43.79  188.5
    area (mean):                          143.5  2501.0
    smoothness (mean):                    0.053  0.163
    compactness (mean):                   0.019  0.345
    concavity (mean):                     0.0    0.427
    concave points (mean):                0.0    0.201
    symmetry (mean):                      0.106  0.304
    fractal dimension (mean):             0.05   0.097
    radius (standard error):              0.112  2.873
    texture (standard error):             0.36   4.885
    perimeter (standard error):           0.757  21.98
    area (standard error):                6.802  542.2
    smoothness (standard error):          0.002  0.031
    compactness (standard error):         0.002  0.135
    concavity (standard error):           0.0    0.396
    concave points (standard error):      0.0    0.053
    symmetry (standard error):            0.008  0.079
    fractal dimension (standard error):   0.001  0.03
    radius (worst):                       7.93   36.04
    texture (worst):                      12.02  49.54
    perimeter (worst):                    50.41  251.2
    area (worst):                         185.2  4254.0
    smoothness (worst):                   0.071  0.223
    compactness (worst):                  0.027  1.058
    concavity (worst):                    0.0    1.252
    concave points (worst):               0.0    0.291
    symmetry (worst):                     0.156  0.664
    fractal dimension (worst):            0.055  0.208
    ===================================== ====== ======

    :Missing Attribute Values: None

    :Class Distribution: 212 - Malignant, 357 - Benign

    :Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

    :Donor: Nick Street

    :Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
https://goo.gl/U2Uwz2

Features are computed from a digitized image of a fine needle
aspirate (FNA) of a breast mass.  They describe
characteristics of the cell nuclei present in the image.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

.. topic:: References

   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction
     for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages 861-870,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and
     prognosis via linear programming. Operations Research, 43(4), pages
570-577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning
techniques
     to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77
(1994)
     163-171.
```

Loading the Data into Pandas

Let's pull the feature and target data out of the cancer_data object. First, the feature data is stored with the 'data' key. When we look at it, we see that it's a numpy array with 569 rows and 30 columns. That's because we have 569 datapoints and 30 features. The following is a numpy array of the data. We use the shape to see that it is an array with 569 rows and 30 columns.

[132]:
```
cancer_data['data']
```

[132]:
```
array([[1.799e+01, 1.038e+01, 1.228e+02, …, 2.654e-01, 4.601e-01,
        1.189e-01],
       [2.057e+01, 1.777e+01, 1.329e+02, …, 1.860e-01, 2.750e-01,
        8.902e-02],
       [1.969e+01, 2.125e+01, 1.300e+02, …, 2.430e-01, 3.613e-01,
        8.758e-02],
       …,
       [1.660e+01, 2.808e+01, 1.083e+02, …, 1.418e-01, 2.218e-01,
        7.820e-02],
       [2.060e+01, 2.933e+01, 1.401e+02, …, 2.650e-01, 4.087e-01,
        1.240e-01],
       [7.760e+00, 2.454e+01, 4.792e+01, …, 0.000e+00, 2.871e-01,
        7.039e-02]])
```

Build a Logistic Regression Model

Now that we've taken a look at our data and gotten it into a comfortable format, we can build our feature matrix X and target array y so that we can build a Logistic Regression model. X = df[cancer_data.feature_names].values y = df['target'].values

Now we create a Logistic Regression object and use the fit method to build the model. model = LogisticRegression() model.fit(X, y)

When we run this code we get a Convergence Warning. This means that the model needs more time to find the optimal solution. One option is to increase the number of iterations. You can also switch to a different solver, which is what we will do. The solver is the algorithm that the model uses to find the equation of the line. You can see the possible solvers in the Logistic Regression documentation model = LogisticRegression(solver='liblinear') model.fit(X, y)

Let's see what the model predicts for the first datapoint in our dataset. Recall that the predict method takes a 2-dimensional array so we must put the datapoint in a list. model.predict([X[0]])

So the model predicts that the first datapoint is benign.

To see how well the model performs over the whole dataset, we use the score method to see the accuracy of the model. model.score(X, y)

[133]:
```python
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

cancer_data = load_breast_cancer()
df = pd.DataFrame(cancer_data['data'], columns=cancer_data['feature_names'])
df['target'] = cancer_data['target']

X = df[cancer_data.feature_names].values
y = df['target'].values

model = LogisticRegression(solver='liblinear')
model.fit(X, y)
print("prediction for datapoint 0:", model.predict([X[0]]))
print(model.score(X, y))
```

```
prediction for datapoint 0: [0]
0.9595782073813708
```

We see that the model gets 96% of the datapoints correct. With the tools we've developed, we can build a model for any classification dataset.

```
[ ]:
```