READ ME

The code I included is about a little project I describe now.

I downloaded a data set from www.kaggle.com which contains ten real-valued features for each carcinogenic cell nucleus of the set. The set is formed by 568 examples, and the goal is to estimate if a carcinogenic cell is benign or malign based on the features given in this data set.

Next, I expose the different files that form my mini-project. It's advisable to open them in the order I list them.

File number 1 – BreastCancerWIDataPlay

In this first file I play a little bit with the data to find the best way of pre-processing it. I also plot the different features distributions to check their behavior.

I comment every single step of the analysis, so you know what I'm doing and why.

File number 2 – BreastCancerWIData

This second file includes all different functions I need to pre-process the data, based on the previous analysis of the file number 1.

I comment every single function as well as their inputs and outputs in the file too.

def randomShuffle(X,y)

Given the matrix data set **X** and its vector of labels **y**, it randomly shuffles them, keeping their correspondences.

def normInput(X)

The function normalizes the input matrix data X. It's necessary for a fast gradient descent algorithm.

def splitData(X,y,trainProp)

Given the matrix data set **X**, its vector of labels **y** and the proportion selected for training, it splits the input data pairs (**X**,**y**) into training, validation and test data sets.

def preProcessing(data,trainProp)

It pre-processes the data by dropping the null features and changing the feature values M and B from diagnosis to 0 and 1.

It creates the matrix data set **X** as well as its vector of labels **y**.

It also normalizes the matrix data **X**. It shuffles **X** and **y**.

Finally, it splits the data into training, validation and test data.

File number 3 – LogReg

This third file includes all different functions I need to compute the logistic regression algorithm to train my model and be able to predict the labels for the test and validation data.

I comment every single function as well as their inputs and outputs in the file too.

def randInit(dim)

Given the dimension of the weights, it randomly initializes them and the bias b.

def lineal(X, w, b)

Given the weights **w**, the bias b and the matrix data **X**, it computes the linear part of the function of logistic regression.

def sigmoid(z)

It calculates the sigmoid function of a given parameter z.

def costfunctionLogReg(A,X,Y,w,regu,lambd)

It computes the cost function of the logistic regression considering the regularization term.

def gradientsLogReg(A,X,Y,w,regu,lambd)

The function calculates the gradient of the cost function to update the values of the weights **w** and bias b. It also considers the regularization term.

def gradientDescentLogReg(X,y,learningRate,numIterations,regu,lambd)

The function applies the optimization method of gradient descent with a logistic regression model. The outputs are the final results for the weights **w** and bias b, as well as a list of the cost values along the iteration (it can be used to check the model behavior).

def classificatorLogReg(X,w,b)

Given the matrix validation/test data **X** and the learn parameters **w** and b, the function gives us the predicted labels.

def evalModel(y_predicted,y_gt)

Given the ground truth labels y for the validation and test data as well as the predicted labels, it calculates true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) in order to compute the precision, recall, f1 and accuracy of the method.

File number 4 – LDAQDA

This fourth file includes all different functions I need to compute the linear discriminant analysis and quadratic discriminant analysis algorithms to train my model and be able to predict the labels for the test and validation data.

I comment every single function as well as their inputs and outputs in the file too.

def preEstimation(X,y,k)

For a class k, the function creates a matrix data **X** and its labels **y**, formed only by the data of that class.

def meanEstim(X)

Given the matrix data **X** formed by the data of certain class k, the function estimates its mean vector.

def covarEstim(X)

Given the matrix data **X** formed by the data of certain class k, the function estimates its covariance matrix.

def bernouEstim(y,m)

It computes the probability of the label **y** being equal to the class k according to a Bernoulli distribution.

def ldaEstimation(X,y,K)

It estimates the mean vector, the covariance matrix and the probability of the label **y** being equal to the class k according to a Bernoulli distribution, for every single class k.

The output is a list which contains as positions as number of classes K. Moreover, each position contains a dictionary with the mean, covariance and Bernoulli probability for each class k.

def qdaEstimation(X,y,K)

It estimates the mean vector, the covariance matrix and the probability of the label **y** being equal to the class k according to a Bernoulli distribution, for every single class k.

The output is a list which contains as positions as number of classes K. Moreover, each position contains a dictionary with the mean, covariance and Bernoulli probability for each class k.

def classificatorLDA(X,estimations,K)

The function classifies the test/validation examples, finding the predicted labels.

def classificatorQDA(X,estimations,K)

The function classifies the test/validation examples, finding the predicted labels.

def evalModel(y_predicted,y_gt)

Given the ground truth labels y for the validation and test data as well as the predicted labels, it calculates true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) in order to compute the precision, recall, f1 and accuracy of the method.

File number 5 – BreastCancerWI

This fifth and last file includes the final analysis in order to find the best model to predict benign or malign breast cancers based on some features of the carcinogenic cell nucleus.

Firstly, we import the different libraries we will need and read the data and pre-process it according to the analysis we did before.

```
data = pd.read_csv('breast-cancer-wisconsin-data/data.csv')
X_train, y_train, X_valid, y_valid, X_test, y_test = preProcessing(data,trainProp=.5)
```

First analysis can be to use a logistic regression model with no regularization, a learning rate of 0'009 and 1000 iterations. Then, we plot the learning curve. The behavior we expected.

```
w, b, costs = gradientDescentLogReg(X_train,y_train,0.009,1000,regu=0,lambd=0.5)
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations')
plt.title("Learning rate = 0.009")
plt.show()
```

Once we have the estimated parameters **w** and b, we predict the **y** labels of the training set and calculate its accuracy. If the accuracy is too low, our model has a high bias issue.

```
y_predict_train = classificatorLogReg(X_train,w,b)
print("Accuracy for the training set is {}%".format(np.round(100-
np.mean(np.abs(y_predict_train-y_train))*100,decimals=2)))
```

Next step, we train our model for 3 different learning rate values and evaluate the results using the validation data set. The test set cannot be used for it because we would be overfitting in the test set.

```
learningRate = [0.001,0.01,0.1]

for i in learningRate:
    w, b, costs = gradientDescentLogReg(X_train,y_train,i,1000,regu=0,lambd=0)
    plt.plot(costs,label=i)

    print("\n\nLearning Rate: {}".format(i))

    y_predict_train = classificatorLogReg(X_train,w,b)
    precision, recall, f1, accuracy = evalModel(y_predict_train,y_train)
    print("Train Accuracy: {}%".format(accuracy*100))
    y_predict_valid = classificatorLogReg(X_valid,w,b)
    precision, recall, f1, accuracy = evalModel(y_predict_valid,y_valid)
    print("Validation Accuracy: {}%".format(accuracy*100))

plt.ylabel('cost')
plt.xlabel('iterations')
legend = plt.legend(loc='best',shadow=True)
plt.show()
```

It looks that the best learning rate is equal to 0'1.

Now we train our model for 3 different lambda values and evaluate the results using the validation data set. We consider a L1 regularization. Again, the test set cannot be used for it because we would be overfitting in the test set.

```
lam = [0.1,0.5,0.9]

for j in lam:
    w, b, costs = gradientDescentLogReg(X_train,y_train,0.1,1000,1,j)
    plt.plot(costs,label=j)

    print("\n\nLambda value: {}".format(j))

    y_predict_train = classificatorLogReg(X_train,w,b)
    precision, recall, f1, accuracy = evalModel(y_predict_train,y_train)
    print("Train Accuracy: {}%".format(accuracy*100))
    y_predict_valid = classificatorLogReg(X_valid,w,b)
    precision, recall, f1, accuracy = evalModel(y_predict_valid,y_valid)
    print("Validation Accuracy: {}%".format(accuracy*100))

plt.ylabel('cost')
plt.xlabel('iterations')
legend = plt.legend(loc='best',shadow=True)
plt.show()
```

It looks that the regularization term doesn't affect the performance that much for this problem. Now we repeat the same analysis but considering a L2 regularization. Again, the test set cannot be used for it because we would be overfitting in the test set.

```
lam = [0.1,0.5,0.9]

for j in lam:
    w, b, costs = gradientDescentLogReg(X_train,y_train,0.1,1000,2,j)
    plt.plot(costs,label=j)

    print("\n\nLambda value: {}".format(j))

    y_predict_train = classificatorLogReg(X_train,w,b)
    precision, recall, f1, accuracy = evalModel(y_predict_train,y_train)
    print("Train Accuracy: {}%".format(accuracy*100))
    y_predict_valid = classificatorLogReg(X_valid,w,b)
    precision, recall, f1, accuracy = evalModel(y_predict_valid,y_valid)
    print("Validation Accuracy: {}%".format(accuracy*100))

plt.ylabel('cost')
plt.xlabel('iterations')
legend = plt.legend(loc='best',shadow=True)
plt.show()
```

Again, it looks that the regularization term doesn't affect the performance that much for this problem.

Now we use the LDA algorithm to model our data. We predict the y labels of the training set and calculate its accuracy. If the accuracy is too low, our model has a high bias issue.

```
estimations = ldaEstimation(X_train,y_train,K=2)
y_predict = classificatorLDA(X_train,estimations,K=2)
print("Accuracy for the training set is {}%".format(np.round(100-np.mean(np.abs(y_predict-y_train))*100,decimals=2)))
```

Once we have checked the model doesn't suffer from high bias, we use the validation data to evaluate the LDA model.

```
estimations = ldaEstimation(X_train,y_train,K=2)
y_predict = classificatorLDA(X_valid,estimations,K=2)
print("Accuracy for the training set is {}%".format(np.round(100-np.mean(np.abs(y_predict-y_valid))*100,decimals=2)))
```

Now we use the QDA algorithm to model our data. We predict the y labels of the training set and calculate its accuracy. If the accuracy is too low, our model has a high bias issue.

```
estimations = qdaEstimation(X_train,y_train,K=2)
y_predict = classificatorQDA(X_train,estimations,K=2)
print("Accuracy for the training set is {}%".format(np.round(100-np.mean(np.abs(y_predict-y_train))*100,decimals=2)))
```

Once we have checked the model doesn't suffer from high bias, we use the validation data to evaluate the QDA model.

```
estimations = qdaEstimation(X_train,y_train,K=2)
y_predict = classificatorQDA(X_valid,estimations,K=2)
print("Accuracy for the training set is {}%".format(np.round(100-np.mean(np.abs(y_predict-y_valid))*100,decimals=2)))
```

For the evaluation with the test set, we use the algorithm and hyper-parameters with which we have obtained better results in the validation set.

In other words, we will use a logistic regression model with learning rate equals to 0'1 and no regularization term at all.

```
w, b, costs = gradientDescentLogReg(X_train,y_train,0.1,1000,regu=0,lambd=0)
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations')
plt.title("Learning rate = 0.1, No Regularization")
plt.show()
```

```
y_predict = classificatorLogReg(X_test,w,b)
print("Accuracy for the test set is {} %".format(np.round(100-np.mean(np.abs(y_predict-y_test))*100,decimals=2)))
```