Logistic Regression Project Report

Abstract:

Breast cancer is the uncontrolled growth of breast cells. It represents the second cause of cancer death in women worldwide. It is important for patients to understand their disease and know what to expect in the future so that they can make decisions about treatment, rehabilitation, financial aid decisions and personal matters. This paper presents an approach for diagnosing breast cancer based on a set of input variables that describe some characteristics of tumor images. The proposed approach builds a binary logistic model that classifies between malignant and benign cases. The approach is applied to the Wisconsin Diagnostic Breast Cancer (WDBC) dataset. Experimental results show that the regression model that is statistically significant includes only the area, texture, concavity and symmetry features of a tumor. In addition to the simplicity of the used model, the reduced set of features gives performance measures that outperform similar approaches. Accordingly, the presented approach can be used for feature selection and reduction of the breast cancer data.

Introduction:

Cancer is a disease in which cells in the body grow out of control. Except for skin cancer, breast cancer is the most common cancer in women in the United States. Deaths from breast cancer have declined over time, but remain the second leading cause of cancer death among women overall and the leading cause of cancer death among Hispanic women.
Each year in the United States, about 245,000 cases of breast cancer are diagnosed in women and about 2,200 in men. About 41,000 women and 460 men in the U.S. die each year from breast cancer. Over the last decade, the rate of getting breast cancer has not changed for women overall, but the rate has increased for black women and Asian and Pacific Islander women. Black women have a higher rate of death from breast cancer than white women.

DataSet:

For this project we implemented a logistic regression function that takes information from the Wisconsin Diagnostic Breast Cancer to determine if a tumor is malignant or benign.

The dataset contains 569 instances with 32 attributes (ID, diagnosis (B/M), 30 real-valued input features). Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. Our job was to take this data-set parse it, and using the features given, try to come up with an efficient logistic regression code to predict whether a tumor is malignant or benign.

Parsing the Dataset:

In order to parse the dataset efficiently, we convert the dataset into a pandas data-frame. Using the following line of code:

```
df = pandas.read_csv('wdbc.dataset', names = x)
```

where x = list(range(32)).

The pandas library automatically parses the dataset, from a csv file, is uses the vector x as a naming convention. I chose a numbering of the columns from 0-31, so the dataframe can essentially be taught like a matrix. Since Logistic Regression works with labeling the binary class as either zero or one.
I parse the dataset to code “M” as a 1 and “B” as a 0.
I did this suing a python dictionary.

```
transform = {"M": 1, "B": 0}
```

I stored all the labels into a vector called Y by using list comprehension:

```
Y = np.array([transform[i] for i in df[1]])
```

Here df[1] is where all the labels are concurrentlystored.

In the X vector the first column consisted of all the Patient IDs. I decided this information wasn't going to help us predict whether the patient has cancer or not. So I did not include that column in the X vector dataset, instead I stored all the variables into an Xarray.Using lis comprehension, I was able to stored the normalized values of each column using a singleline
X = np.array([np.array((df[i]-min(df[i]))/(max(df[i])-min(df[i]))) for i in range(2,32)])
Here I took each column df[i] subtracted the minimum then divided by the range.
Then I used the list comprehension to store all of the columns into a single array.
indexing = np.array(range(len(X[0])))
np.random.shuffle(indexing)
Here I shuffled the indexes of an array
X = X[:,indexing]
Y = Y[indexing]
Then used numpy indexing, to index the numpy array in order to give X and Y this newrandom orderingThe reason indexing was used, so the same shuffling can be applied to both X and Y. **a =int((len(X[0])*0.8))**
b =int((len(X[0])*0.1))
This is was used to code that 80% was testing, while 10% was for validation and the rest (whichis10%)is used for testing.
X_train = X[:,:a]
X_val = X[:,a:a+b]
X_test = X[:,a+b:]
Y_train = Y[:a]
Y_val = Y[a:a+b]
Y_test = Y[a+b:]
This is to give the three way split to each testin, training and validation.

Architecture:

```
Epochs = 1000
```

Epochs is the number of iterations we are going to run our gradient descent algorithm to figure out our weights for the logistic regression algorithm

```
m = X_train.shape[1]
```

This gives the size of the training samples. Since the rows of X gives the features, the columns give each training sample.

```
def sigmoid(z):
        if z > 0:
                return np.reciprocal(1 + np.exp(-z))
        return 1-np.reciprocal(1 + np.exp(z))
```

This is the sigmoid function. If our value is positive we follow the strict definition. If our value is negative we must rewrite the sigmoid function to avoid evaluating big posisitive exponents. For example if x is -10000, our algorithm needs to compute e^10000 which is astronomically large.
So how do we rewrite the sigmoid function? Using a bit algebra we get this 1/(1+e^-x) = e^x/(1+e^x) = 1-1/(1+e^x).

```
w = np.random.randn(X_train.shape[0], 1)*0.01
```

Here we randomize our initial weight vector. We use the guassian normal function to our weights. The shape of our weight function is given by the first component of X_train.

```
win = copy.deepcopy(w)
```

A deep copy of the intial weights of w are performed

```
b = 0
```

Intialize our b value

```
maxtup = (0,0)
```

This will be later used to find learning rate that gives maximum F-score.

```
np.seterr(divide = 'ignore')
```

This is just place there to avoid error message when running large learning rates.

```
alpha = list(range(-5,5))
alpha = [10**(i) for i in alpha]
```

This is the range of learning rates that we are trying to apply. It ranges from 10^-5 to 10^5. We look through alpha and try different learning rates to see which one gives the best F-score.

```
for learningrate in alpha:
        for epoch in range(epochs):
                z = np.dot(w.T, X_train) + b
                p = np.array([sigmoid(i) for i in z[0]])
                cost = -np.sum(np.multiply(np.log(p), Y_train) + np.multiply((1 - Y_train), np.log(1 - p)))/m
                dz = p-Y_train
                dw = (1 / m) * np.dot(X_train, dz.T)
                db = (1 / m) * np.sum(dz)
                w = w - learningrate * dw
                b = b - learningrate * db
```

This code was borough from the medium article in Sri hari's notes. We loop through each learning rate.
We loop through it by the number of epochs to run it epoch times. Z calculated the dot product of the weight and the X_train.

P actually calculated the sigmoid of the z vector, here we use list comprehension in order to perform sigmoid on each value for the z vector. The cost function is given by the cost function of logistic regression formula.

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

rHere the cost function is just implemeted on numpy.
The gradient of this function is given by

Want $\min_\theta J(\theta)$:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

$\}$

dz computes the inner size
then we dot product it X_train to the xj component multiplication and addiction. We then multiply that by 1/m in order to get the final results. We do something smiliar to the bias.

z = np.dot(w.T, X_val) + b
Y_testval = np.round(np.array([sigmoid(i) for i in z[0]]))
After it is trained we used the weights to evaluate the function on X_val. Then we applied the sigmoid and round the results 0 if below 0.5, 1 otherwise. This gives us what the actual regression function spits out.
TP = np.dot(Y_testval,Y_val.T)
Here we dot both the guess from our function and actual results. So only way we can get a one if both are positive (True Positives), the dot product adds them all up for us!!! Neat right? Who's the kong of python!
TN = np.count_nonzero((Y_testval+Y_val)==0)
Now here's a slick way of seeing our true negatives. So what we do is we add the two vectors and if equals 0 then we return a 1. Notice only equals 0 if both are zero.
FP = np.count_nonzero((2*Y_testval+Y_val)==2)
FN = np.count_nonzero((Y_testval+2*Y_val)==2)
This scales up the vector which we want ones by 2 and counts how many 2s we have after adding.
1*2+0 = 2 only combination that gives two for 2*x+y where x and y are binary values.

```
Precision = TP/(TP+FP) Recall
= TP/(TP+FN)
 F1 = 2/(1/Recall+1/Precision)
```
Self explanatory given by the formulas online.

```
 if F1> maxtup[0]:
              maxtup = (F1,learningrate)
```
Finds max F1 score and corresponding learning rate.

```
 learningrate = maxtup[1]
```
we can use the learningrate given the max.

```
w = win
b = 0
 acctrack = []
```
reinitializing everything!

```
 for epoch in range(epochs):
              z = np.dot(w.T, X_train) + b
              p = np.array([sigmoid(i) for i in z[0]])
              cost = -np.sum(np.multiply(np.log(p), Y_train) + np.multiply((1 - Y_train), np.log(1
 - p)))/m
              dz = p-Y_train
              dw = (1 / m) * np.dot(X_train, dz.T)
              db = (1 / m) * np.sum(dz)
              w = w - learningrate * dw
              b = b - learningrate * db
              TP_test = np.dot(np.round(p),Y_train.T)
              TN_test = np.count_nonzero((np.round(p)+Y_train)==0)
              acctrack.append((TP_test+TN_test)/(len(Y_train)))
```
Redoing everything accept keeping track of accuracy at every step and using the largest learning rate.

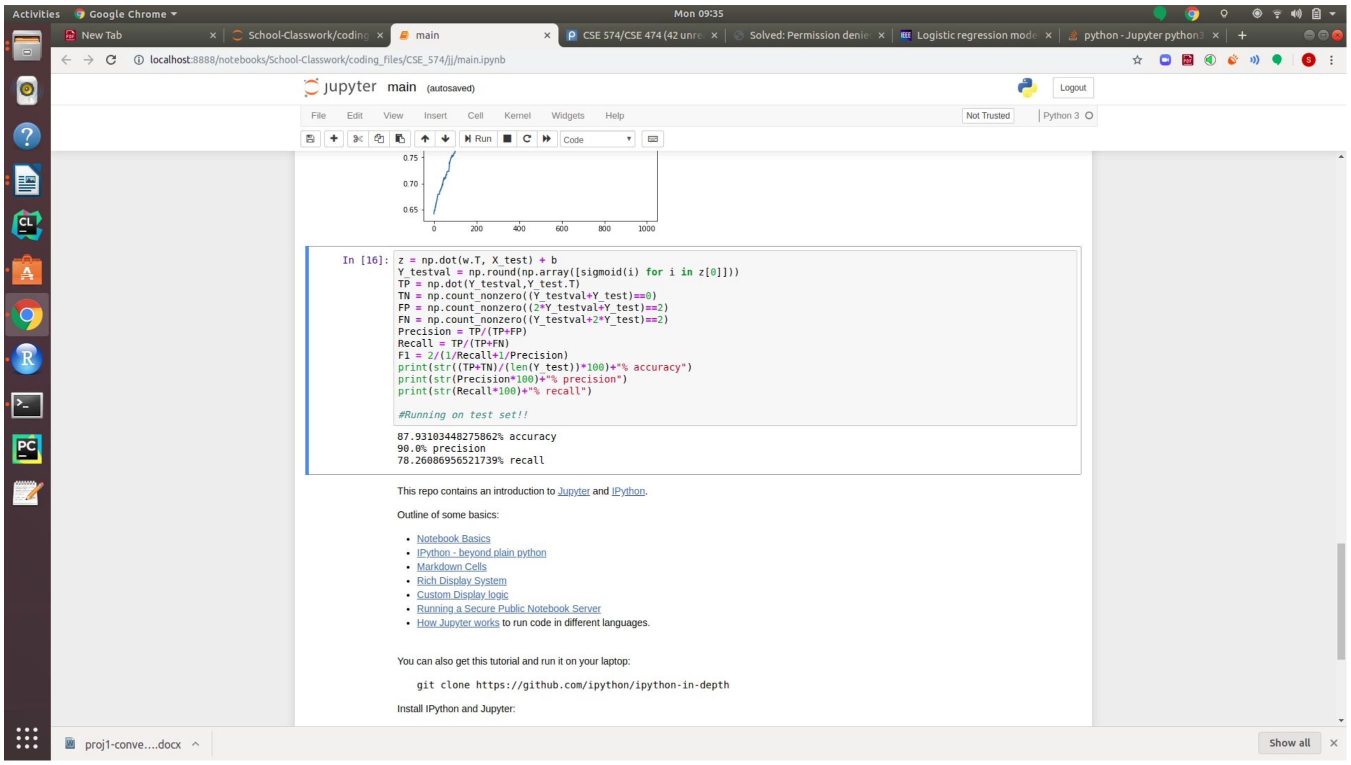
```
plt.plot(list(range(epochs)),acctrack)

plt.show()
```
Final plot of accuracy.

z = np.dot(w.T, X_test) + b
Y_testval = np.round(np.array([sigmoid(i) for i in z[0]]))
TP = np.dot(Y_testval,Y_test.T)
TN = np.count_nonzero((Y_testval+Y_test)==0)
FP = np.count_nonzero((2*Y_testval+Y_test)==2)
FN = np.count_nonzero((Y_testval+2*Y_test)==2)
Precision = TP/(TP+FP)
Recall = TP/(TP+FN)
F1 = 2/(1/Recall+1/Precision)
print(str((TP+TN)/(len(Y_test))*100)+"% accuracy")
print(str(Precision*100)+"% precision")
print(str(Recall*100)+"% recall")

Running on test set!!

Results:



```
In [16]: z = np.dot(w.T, X_test) + b
         Y_testval = np.round(np.array([sigmoid(i) for i in z[0]]))
         TP = np.dot(Y_testval,Y_test.T)
         TN = np.count_nonzero((Y_testval+Y_test)==0)
         FP = np.count_nonzero((2*Y_testval+Y_test)==2)
         FN = np.count_nonzero((Y_testval+2*Y_test)==2)
         Precision = TP/(TP+FP)
         Recall = TP/(TP+FN)
         F1 = 2/(1/Recall+1/Precision)
         print(str((TP+TN)/(len(Y_test))*100)+"% accuracy")
         print(str(Precision*100)+"% precision")
         print(str(Recall*100)+"% recall")

         #Running on test set!!

         87.93103448275862% accuracy
         90.0% precision
         78.26086956521739% recall
```

This repo contains an introduction to Jupyter and IPython.

Outline of some basics:

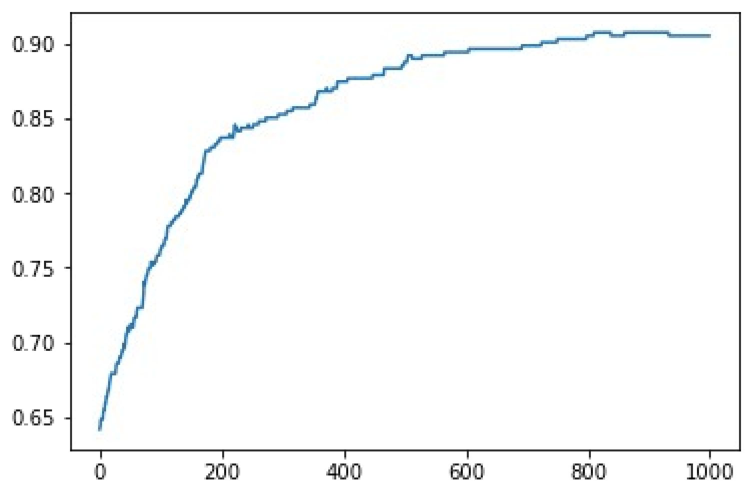
- Notebook Basics
- IPython - beyond plain python
- Markdown Cells
- Rich Display System
- Custom Display logic
- Running a Secure Public Notebook Server
- How Jupyter works to run code in different languages.

You can also get this tutorial and run it on your laptop:

```
git clone https://github.com/ipython/ipython-in-depth
```

Install IPython and Jupyter:

Here we see we get 88% accruacy, 90% Precision and 78% Recall



This graph shows us accuracy vs number of epochs (accuracy- y axis and number of epochs x-axis)
We can see that we could achieve similar results if we limited epochs to 800.

Conclusion:

Here, we can conclude that logistic regression is a good algorithm to detect whether a cancer is benign or malignant. Our logistic regression model can effectively discriminate between benign and malignant breast disease and can identify the most important features associated with breast cancer.