Implementation and testing of the Logistic Multiclass Regression

1 Introduction

For this project we will build a Logistic Regression model for multi-class classification. Logistic Regression is a discriminative probabilistic statistical classification model which can be very useful when dealing with relatively small dataset and excels at classifying data whose complexity is not very high. In order for it to work, we first train the model with a subset of the data we want to classify. We already got the algebraic definition for the binary classifier logistic regression model. We will work out from there and eventually we will obtain the definition for the multi-class model.
There is an error function which we will be trying to minimize. We will do so by iteratively changing the parameters of this function in different methods. At the end, we will be able to quantify the results so we can see each method's performance and accuracy.

2 Deriving the algebraic formulas

2.1 Defining the formulas for the multiclass LR

The binary Logistic Regression model can be considered as a linear model of classification. The Logistic model function aka "the sigmoid function" for this model is given by:

$$g(x) = \frac{1}{1 + e^{-z}}$$

From this binary classifier model function we can derive the similar function for the multiclass. In this case the function is called Softmax function:

$$P(X = i|x_i, W, b) = \frac{e^{W_i X + b_i}}{\sum_j e^{W_j X + b_j}}$$

Given an unknown vector x, the prediction is obtained from the following formula:

$$\hat{y} = argmax_i P(Y = 1|x, W, b)$$

If we extend this formula for the softmax function we obtain the following:

$$\hat{y} = argmax_i \left(\frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}\right)$$

Our goal is to maximize the probability of data x being predicted correctly. For this we can only modify the so called weights of the above formula. In our case they consist on the vector *W* and vector *b*. The better the weights are chosen, the more accurate will be our model.

The estimation technique we will be using is called the Maximum Likelihood. This method estimates the parameters so that the likelihood of training data is maximized.

$$L(\theta = W, b, D) = argmax \prod_{i=1}^{N} P(Y = y_i|x_i, W, b)$$

2.2 Calculating the gradient of the MLE

If we'd try to calculate the above Maximum Likelihood function it's obvious we would be getting arbitrarily small weights. This is a problem assuming we will be implementing this model in a computer which rounds floating point numbers at a certain point.

That's why we will calculate the log of this function instead, the so called Log Likelihood.

$$L(\theta, D) = argmax \sum_{i=1}^{N} \log P(Y = y_i | x_i, W, b)$$

We can convert this function into a minimization problem by negating its result. After this we'd obtain the following function:

$$L(\theta, D) = -argmin \sum_{i=1}^{N} \log P(Y = y_i | x_i, W, b)$$

Now that we've got the maximum likelihood function, our next step consists on implementing the gradient based methods for minimization. For a single data our function is defined as following:

$$\begin{aligned} L(\theta, D) &= -\log P(Y = y_i | x_i, W, b) \\ &= -log \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}} \\ &= -log\ e^{W_i x + b_i} - log \sum_j e^{W_j x + b_j} \\ &= -W_i x + b_i - log \sum_j e^{W_j x + b_j} \\ &= -W_i x + b_i + log \frac{1}{\sum_j e^{W_j x + b_j}} \end{aligned}$$

Now we can easily find the gradient of this function with respect to the weights. In other words, we will try to find the partial derivative of the formula with respect to the weights W, which will tell us how much does the MLE change for each 1 unit change in the weights. Note that we're deriving the formula for one single data.

$$\begin{aligned} \frac{\partial}{\partial W_i}\left(W_i^T X_i - log \sum_k^K e^{W_k^T X}\right) &= \frac{\partial}{\partial W_i}\left(W_i^T X_i\right) - \frac{\partial}{\partial W_i}\left(log \sum_k^K e^{W_k^T X_i}\right) \\ &= X_i - X_i \frac{e^{W_i^T X}}{\sum_k^K e^{W_i^T X}} \end{aligned}$$

Now we have to calculate the partial for all other Wj for which j≠i:

$$\frac{\partial}{\partial W_j}\left(W_i^T X_i - log\sum_k^K e^{W_k^T X}\right) = \frac{\partial}{\partial W_j}\left(W_i^T X_i\right) - \frac{\partial}{\partial W_j}\left(log\sum_k^K e^{W_k^T X_i}\right)$$

$$= -X_j \frac{e^{W_j^T X}}{\sum_k^K e^{W_k^T X}}$$

From the derivation formulas obtained above can be observed that they are similar but the partial for all other Wj has one less term. This happens because any change on Wj doesn't affect the term $W_i^T X$ at all, that's why the derivative of this term is 0.

3 Building the model

For this project we will be using Python programming language to implement our model. By doing so we will take advantage of the fast matrix operation libraries python has to offer.

Our first job is to implement the core part of a LR model, implementing the Softmax function.

Depending on the input data, generally we don't want to calculate a huge number that's why we initially subtract the maximum value of vector X from each of its cells. This is a method used to regularize and normalize the data.

```
def softmax(x):
    e = np.exp(x - np.max(x))   # prevent
overflow      if e.ndim == 1:

     else:return e / np.sum(e, axis=0)


        return np.divide(e ,
np.array([np.sum(e, axis=1)]).T)   # n_dim
```

Our next step consists on implementing the cost function. In our program we called it negative log likelihood.

```
def negative_log_likelihood():
                    sigmoid_activation = softmax(np.dot(x, W) + b)
     =cross_entropy

                                cross_entropy
np.mean(np.sum(y*np.log(sigmoid_activation)+(1-y)*  np.log(1-
sigmoid_activation),axis=1))
    return
```

The third most important part of our model is the implementation of the gradient descent.
Below we can see the function implementation of the partial derivatives we calculated in section two of this paper:

```
def     L2_reg=.):
(lr=.1:rain_descen t
                    prob_y_given_x =
                    W) + b)     d_y = y
                    -


      y_given_x     s. added regularization
                                -
      += lr * np.do
                          lr *
                              L2_reg*np.square(W)
      += lr * np.me         axis=0)
```

Note that besides implementing the gradient descent formula we also added a regularization term which will get useful in the tests we'll be conducting.
Another thing to notice is the modification of the b weights, also known as the intercept of the function. In order to do so, we derive the b from the simplest form of the function where y=b+mx and substitute the y, m and x accordingly in our model. Note that we use the mean of all errors for each x to calculate b.

4 Setting up the tests

Now that we implemented our model, our next goal is to set up the test files we are going to use in order to measure the performance, convergence rate and accuracy of our model.
For this project we will be using three popular datasets, namely: Iris dataset, handwritten digit dataset, Cifar-10 dataset.
The first one consists on a set of 3 class data. Each row has 4 columns or parameters which represent the Sepal Length, Sepal Width, Petal Length and Petal Width. There are 150 data in total. We will import this dataset from the 'scikit' library.
The next dataset we will be using is the handwritten digits dataset. It consists of 1797 rows of data, which are classified into one out of ten possible classes. Each datapoint is a 8x8 image of a digit.
The last dataset we will use is the CIFAR-10 dataset. It consists of 10.000 test images each one being classified as one of 10 classes. Each datapoint consists of a 32x32 color image.

Our program makes use of a function called getLabelVector. For every datapoint, It generates the label vector. It gets as an input the total number of classes and the class of the datapoint we are building the label vector for. For the two image datasets we will not be doing any other preprocessing other than adding the label vectors.

```
def getLabelVector(n_classes=2, cl=2):     tmp = np.zeros([n_classes])     tmp[cl-1] = 1;     return tmp
```

We will initially implement the simplest version of minimization known as the steepest descent. In order to quickly achieve the minima we will continuously modify the 'learning rate' coefficient.
Initially the weights are randomly initialized to a floating point number ranging from 0 to 1.
For each learning-iteration we will modify the weights accordingly. If the new cost is higher than the previous one, we will undo the change of the weights and we will halve the learning rate.
In case the cost becomes lower we will increase the learning rate by 10%. We also implemented a counter which makes sure we quit the learning loop if the cost doesn't change for a certain number of iterations in a row, meaning the model converged and no further changes can be made to obtain better results.
Another technique we will be adding to the training method is the regularization. We will implement the L2 regularization method also known as ridge regression. The basic idea behind regularization is to penalize the higher weights. This method has shown to be very useful at preventing over-fitting in the statistical models such as Linear Regression. At the end of the section 3 we can see the implementation of ridge regularization method for our model.

We will check our results by making use of another learning method called the Quasi-Newton method. For this we will import the function fmin_bfgs() from the library scipy.optimize.
The implementation code is the following:

```
optimize.fmin_bfgs(neg_log_lh_optimization_func, x0=_b_W)
```

The first argument is our cost or lost function and the second parameter is the set of weights given to the function.

To make sure the gradient is calculated correctly, we will make use of another function of the scipy.optimize library. The function we will implement is the check_grad() function and it basically uses the difference approximation method to check the gradient function. Below we can see the implementation in our program:

```
optimize.check_grad(func=neg_log_lh_optimization_func,
    grad=train_descent_optimization_func, x0=_b_W)
```

We will be giving three parameters to this function namely the cost function, the gradient descent function and some initial weights. This method returns a value which represents the error of our gradient. A small value means the gradient is being calculated correctly.
The last learning method we will be implementing is the cross-validation method. Cross validation helps reducing the over-fitting and has proven to produce much better results than the conventional idea of breaking a dataset into parts namely the train set and the test set.
When using cross-validation we divide the dataset into 'k' equal parts. We loop 'k' times and for each time we let the train set to be one of the 'k' sets while we keep training over the rest of the dataset.

Lastly we will implement PCA function from the scikit library. Basically the idea behind the PCA is to reduce the dimensions of the data before we feed it to our model. We will be implementing this method to the largest dataset we are using, the CIFAR-10.

5 Measuring and profiling the results

In this section we will display and analyze the results we obtained from performing several tests over our model. The goal of these tests is to observe the overall accuracy of the model given different testing datasets and different parameter configurations. We will also analyze the convergence rate of the loss function along the overall accuracy.
Prior to go ahead and apply the model we make sure whether our gradient is calculated correctly. We do this validation by comparing our function with a finite difference approximation. Practically we did this validation and we obtained a very small value called "error" of about $2.026xe^{-12}$. This means our model calculates the

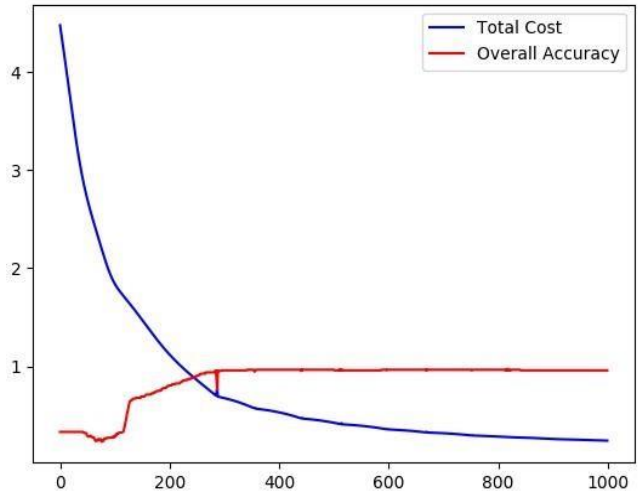
gradient correctly, thus we are confident now to go ahead and test it on our datasets.

5.1 Training the model with Steepest Descent

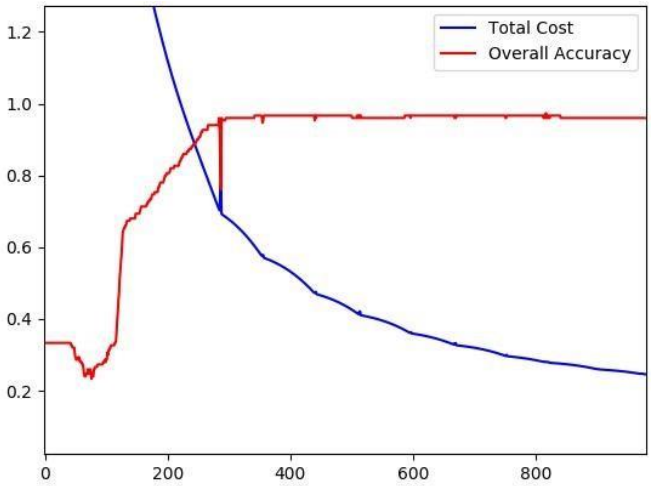
We'll be testing our steepest descent algorithm using the "Iris" and "Handwritten digits" datasets.
The steepest descent (a.k.a. gradient descent) consists on modifying the weights of the loss function based on the value of the partial derivative of the loss function with respect to the weights themselves. Our goal is to minimize the negativeloglikelihood function which we refer to as cost function, in the most efficient and accurate way.
The steepest descent algorithm is trying to reach an ideally global maximum of the cost function. It does so by making use of the partial derivative of that function at a certain point.
Initially the weights are initialized either at a certain arbitrarily value say 0, or randomly initialized.
Then it updates the weights of the function based on the derivative.
Note that instead of applying the "full" derivative value to our weights, we only apply a part of it. We can define this "part" by making use of the Learning Rate variable which is one of the most important parts of a steepest descent algorithm.
After updating the weights we calculate the cost function.
If the value returned by this function is lower than the one before, this means we did better with the modified weights and we can assume that the direction of movement that the derivative gave to us is right, that's why we decide to increment the original Learning rate by a factor of two percents (i.e multiplying it by a factor of 1.02).
If the cost value obtained after the updated weights is higher than the previous cost value, we can say that our step wasn't a good one, that's why we undo the last step by replacing the weights with the weights before the last update. Beside that we choose to decrement the learning rate by 50% (i.e. multiplying it by a factor of 0.5). In this way we will make sure the next "step" of our gradient descending algorithm will be smaller.
In our algorithm we also included a "convergence counter" which basically counts the number of epochs or iterations that the change of the cost is 0. In other words even though we may set the algorithm to iterate for say 1000 epochs, the steepest

descent algorithm may converge before that number. We don't want to keep iterating if the algorithm isn't improving the cost function anymore.

By applying these simple rules we managed to obtain very impressive results.
When we applied the model to the iris dataset we obtained the following results:

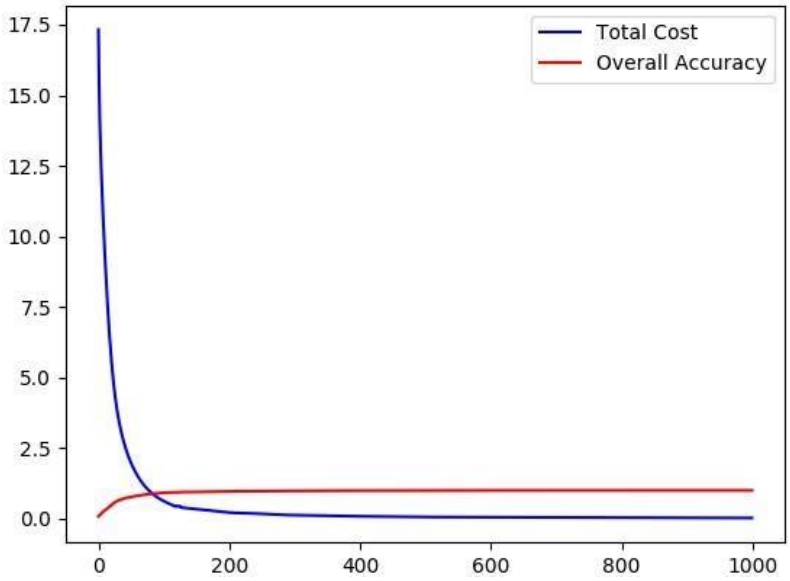


The graph in the left shows how the total cost and the accuracy changed after each "learning" iteration. There were 1000 epochs in total. At the end of all learning epochs the model managed to successfully classify 96.666…% of the data.
Because the initial weights are initially randomly assigned this graph may change for each time we run but we noticed that in almost every case the model successfully converges to this accuracy rate.
From the graph we can observe the relation between the total cost and the prediction rate. As the cost becomes smaller the accuracy increases.
In order to see this relationship better we zoomed in to the last graph and obtained the following view:
We can see from the graph that the overall accuracy, represented by the red line, eventually converges to ~96% after ~300 epochs.



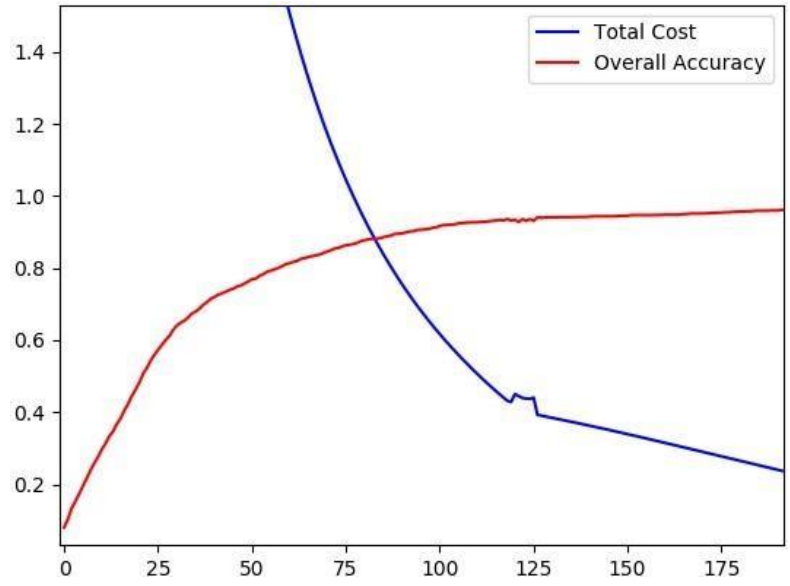
There is a small spike of the cost function at ~240 epochs. Seems like the algorithm made a "wrong" step which gets canceled right away. Smaller spikes of this type may be seen throughout the whole blue line.
Obviously these spikes are reflected in the accuracy line as well.
We managed to run the model for 5.000.000 epochs for the Iris dataset and we were able to obtain an accuracy rate of ~98.2%. Even though the number of epochs got arbitrarily large, the accuracy rate didn't change much. This is why we are discouraged to run the model for a high number of epochs.

After the pleasing results we obtained from the Iris dataset, we run our model over the handwritten digits dataset. Below we can observe the "cost & accuracy" graph we obtained:



Since each data point consists of 64 parameters where each one of them represents a certain pixel of the digit image, obviously the cost function is initially very high.
The model quickly recovers from that by lowering the costs "drastically" in less than 50 epochs.
We can see that the accuracy level converges pretty quickly as well.

In order to better see the convergence of the two lines we zoomed in the graph and obtained the following:

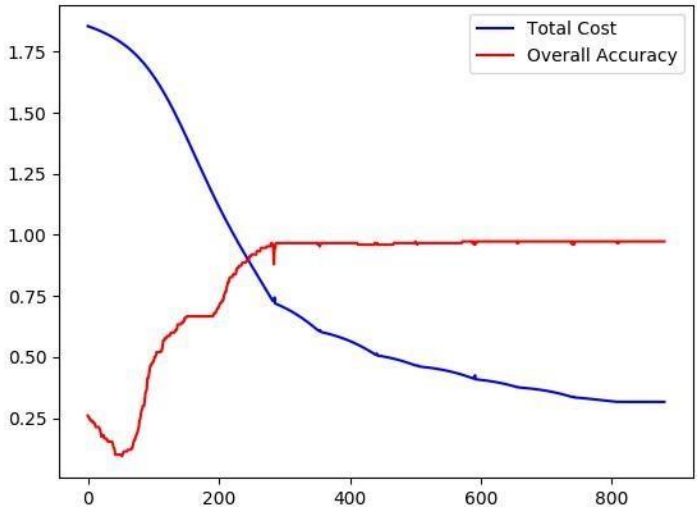


In this version of the graph we can see that the cost function is smoother for the handwritten digits dataset than the one for the iris dataset. Even the accuracy rate seems to converge quicker (in less epochs) for this dataset.
We got such an impressive overall accuracy of 100%. It means that our model managed to accurately classify 100% of the data we trained it on.

Training the model with Regularized Weights

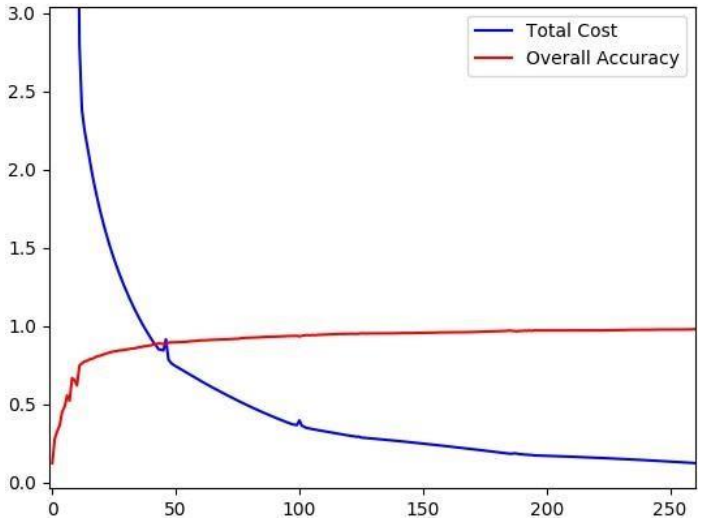
The regularization method we will be using as stated in the section four is the ridge regularization. By penalizing the higher weights the model is supposed to learn faster.

In practice, in about 10% of the times we run our model on different sets, we ended up with arbitrarily large weights. When calculating the softmax function we would end up calculating the exponential of a large number. In python this results in assigning the value 'nan' as a value. This makes the model fail.
By making use of the regularization of the weights we overcame this issue. Another positive aspect we observed after making use of the regularization is the smaller number of iterations needed for the model to converge.
Below we can see the results obtained by applying a coefficient of 0.0001 to the "regularizer" for the iris dataset:



From the results we can see that both the cost and the overall accuracy converged faster. Since the last iterations had no impact over the cost, the model quit the learning loop earlier at about 900 epochs. Another interesting aspect we observed is the accuracy rate. Even though fewer epochs were used, we managed to get an overall accuracy rate of 97.333%.

We applied the regularized learning method to the handwritten digits dataset as well and we obtained the following results:



In the handwritten digits dataset we used a regularization rate of 0.00005.

The graph is similar to the one we obtained in the previous sub-section 5.1. The difference consists on the fast converging rate. At about 50 epochs the model acquired a ~95% accuracy. The overall accuracy is about the same, around 99.9994%.

Training the model with 2nd order optimization (Quasi-Newton method)

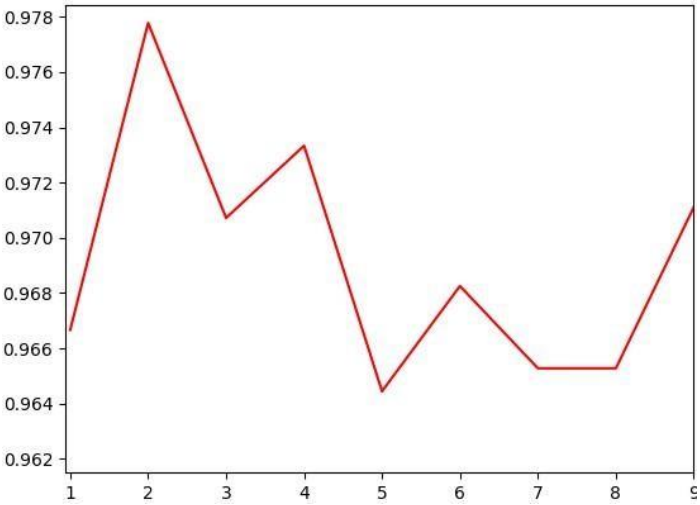
The 2nd order optimization method consists on another more accurate way of gradient descent learning approach. Using this method requires much more resources to compute. Anyways, using this method we achieved optimal convergence rate in much less iterations.
For the Iris dataset the method required 11 iterations to achieve an accuracy of 97%.
The output of this function for the Iris dataset was the following:

```
Current function value: 0.363623         Iterations: 10
         Function evaluations: 187
         Gradient evaluations: 11
 [ -5.00549064   1.08575341   5.65110727  10.2257016   -4.16866126   4.9543357
-31.46111516   6.28784179  26.11245876 -12.69618551
   1.58770113  13.04863667  13.11826821   1.14724059 -2.87749914]
0.966666666667
```

Note that the values in the brackets are the weights at which we get the accuracy highlighted in yellow.
The method tends to converge at a different number of iterations depending on the values initially assigned to the weights. Anyways the results tend to be the same.

Training the model with Cross-Validation

Cross validation consists on dividing the data in k parts. We train the model on k-1 parts and test its accuracy at the other part left.
Since the iris dataset is class-sorted we wouldn't like our model to get tested on some classes it never seen. This is why we randomly shuffle the data beforehand.
After that we iterate k times through the data performing the needed actions.
We train the model using the method we tested on the section 5.1.
After running the function for different values of 'k' within the range 1 to 9, for the Iris dataset we obtained the following graph:



The test was run for three times and each accuracy rate represents the average accuracy for each value k in the x-axis. From this graph we can say that the best accuracy rate is obtained at k=2. At this point we achieved an accuracy of ~97.8%

This is understandable since the iris dataset is quite small.

Even though the variance from one 'k' to another seem to be huge, if we focus on the y axis we can see that the differences from one 'k' to another lay among 1%.

Applying this algorithm to the handwritten digits dataset resulted in a slight decrease in accuracy performance rates. In this case the cross-validation underperformed comparing to the other learning methods we used.

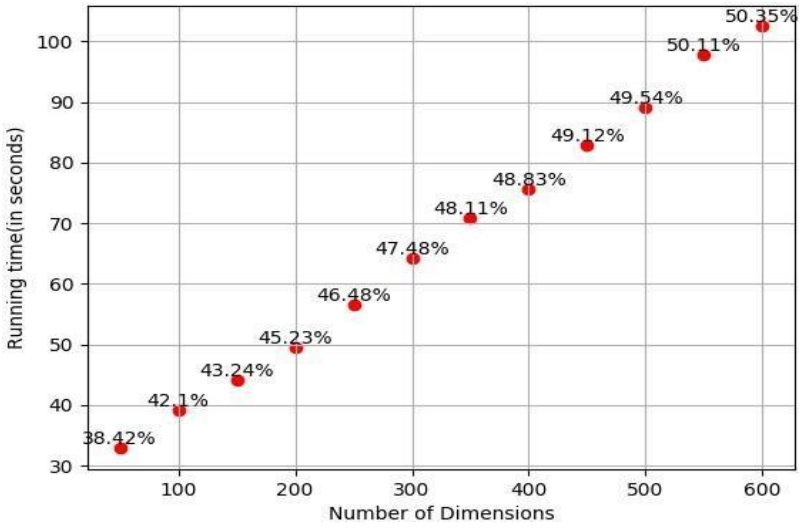
Implementing PCA

Principal component analysis (PCA) is a statistical procedure that converts a set of observations into lower dimensions. PCA is a very useful tool to use when dealing with a high dimensionality data. Previously in this paper we analyzed datasets with relatively small number of dimensions. The last dataset we will be testing is the CIFAR-10 dataset. The full dataset consists of 60.000 images classified in 10 different classes. Each image consists on a 32x32 RGB image. If we choose to directly feed these data to our classifier as it is, each data point would have 32*32*3 = 3.072 dimensions. This is quite a large number and it would take a lot of time for our model to process. We will be using only 1/6th of the whole dataset. This is where PCA comes handy. By using this library we are able to reduce the number of each data point to a fixed number we choose. Obviously the larger the number the better the accuracy would be.

To test this claim we run the model using PCA with different number of dimensions.

For each dimension, the maximum number of learning epochs was 1.000.

Beside the accuracy we also measured the running time for each configuration. Below we may see the results we obtained:



From the table we can clearly see how the running time increases as the number of dimensions increases. Anyways we can see that the increment is not proportionally. For an additional increase of 100 dimensions, the model needs approximately 10 seconds more to process the data.

Note that the time needed by the PCA to reduce the number of dimensions is not included. Another interesting aspect is the overall accuracy rate noted above each point in the graph. If we look closely we can see that the overall rate of accuracy is also improved as we increment the number of dimensions for the data. 50.35% accuracy was achieved by using 600 dimensions per data.

Running the data over the original dataset which consists of 3.072 dimensions per data point took 1.100 seconds. Beside the large running time the overall accuracy was worse too. After 1.000 epochs the model managed to accurately classify only 38.31% of the data. This is worse than any overall accuracy rate obtained by applying the PCA tool.

6 Conclusion

Throughout this project we algebraically defined the Multiclass Logistic Regression model. This model is similar the idea behind the regression model except the fact that it uses a different approach to classify the data. The softmax function this model incorporates transforms this model into a statistical model.

We covered the detailed implementation of this model on a computer, using Python programming language.

In order to test this model we implemented different experiments. We made sure the gradient calculation was correct by checking it with the 'check_grad()' function, part of the '*scipy*' library.

We tested different learning rates, number of epochs and different regularization coefficients trying to understand their significance and define the optimal configuration for our model.
We used two popular datasets to implement the experiments, namely: iris-dataset and the handwritten-digits-dataset.
For each of the datasets we plotted the overall accuracy rate and the total cost of the model (error).
Quasi-Newton method of optimization was also implemented for both the datasets and the results were plotted as well.
We tested the cross-validation method of learning. The results were quite interesting. We plotted the impact the number of divisions 'k' has over the accuracy. Using the Iris dataset, we obtained best result at k=2.
At the end we implemented the Principal Component Analysis tool a.k.a. PCA. We tested this feature on another big dataset called CIFAR-10 which consists of 60.000 of images. The results obtained from using the PCA against not using it at all were much better both in terms of overall accuracy and time efficiency. We plotted the correlation of the number of dimensions used, the overall accuracy rate and the running time the model took to run for each configuration. The best overall accuracy of 50.35% was achieved by making use of 600 dimensions from PCA.