MACHINE LEARNING PROJECT REPORT

1. Uploading the CSV files: In all cases, I uploaded the csv files using files library of google colab. After that I use pandas to read and store the data into a numpy array. Subsequently the target variable and input features are separated from the data.

2. Normalization: Wherever required, the standardization of data is done by z-score normalization where we subtract each data from the combined mean and divide it by its standard deviation.

3. Regression: The regression class is used for both linear and polynomial regression since the underlying process of the change of weights and bias is the same in both the cases.

- We compute the cost function by the standard formula of (1/2m) *sum of (wx+b-y) ^2. After using transpose of certain matrices to fit the dot product properly the cost function gave the required result.
- After this follows the standard gradient descent method where derivative of cost function with respect to weights and bias is calculated and the same is adjusted accordingly. With each iteration we store the cost function so that we can finally plot the decrease in cost function with increasing iteration and determine where the value saturates.
- After w and b are trained they are put into the test data and the value predicted is stored in a new csv file.
- At first, my implementation used loops and hence was too slow to predict the data. After suitable vectorization in the derivative part the result is obtained in just 1 second. The problem that was

occurring was that the dimensions were not properly matched and hence I had to figure out where to use transposes.

- The learning rate after testing turned out to be as high as 0.1 for really fast decrease in the cost function. Iteration was kept at 100 and the corresponding cost is about 4771. Further decrease in cost function of up to 4768 can be achieved but I felt the marginal benefit for the increased number of iterations and hence increased amount of time was unnecessary.

4. Polynomial Data details: The only difference here with respect to Linear Regression is that the 3 input features are multiplied to different degrees together so that a polynomial expression is generated with a certain degree. Subsequently the process is same as the one in linear with the new features being passed to the regression class.

- At first I defined degree as the power of each individual term (for example a degree 7 means terms up to (x^7)(y^7)(z^7)). This produced over inflated terms which not only exponentially increased the time required for execution but also made the cost too high.
- So I redefined degree as the sum of individual degree of each term (for example a degree 7 means terms up to (x^2)(y^2)(z^3)). This decreased computation time and gave smooth variations in cost function.
- After hit and trial, it was found that the hyperparameter degree is optimal for 5 because the cost decreases till 5 and increases thereafter.
- With a learning rate of 0.1 and iterations equal to 10000 the accuracy came in the order of 99.996%.

5. Classification: Logistic regression is carried out with the help of the classification class.

- At first, I did the whole algorithm with the sigmoid function as is commonly used in logistic regression. It involved creating ten different sets of weights and bias because the particular problem is of multi-class classification with digits from 0-9. So each set of w, b is tuned for each digit. This of course meant using loops.
- The accuracy in the beginning came to be round about 10%. Due to some reason every set of w, b was predicting the same digit. Then the idea came to me to use the argmax () function to calculate where which digit had the maximum probability for the given input features.
- All of this implementation produced an accuracy of about 83% for 1000 iterations. But the execution took around 30 minutes so I decided to use the SoftMax function which better fits multi-class classification because of its vectorized approach.
- The changes to be made were that I put the 10 sets of weights and bias into a vector. Normalization in case of sigmoid function was replaced by subtracting the max value from each data point in case of SoftMax function. This is an essential step because when 1 is added to any value less than e^(-40) it results in loss of precision and the function returns 1 which causes an error in logarithm because log(0) is undefined.
- The essential step in SoftMax is hot encoding of the target variable which is a standard procedure with not much deviation. We split each target variable into 10 parts and keep the position of the corresponding digit as 1 and others as 0.
- We again plot the accuracy with increasing iterations on 20% split of the train data.
- Learning rate optimization took some time because there were irregular decrease in accuracy in certain places but finally it was fixed at 2e-6.

- Finally, one more step to simplify the whole procedure was that I removed the part where we calculate the derivative of cost function with respect to the bias. To compensate for it, I added an extra input feature with value 1 in each data row. But it did not produce much difference in the result.

The change to SoftMax decreased the time required significantly from 30 minutes to around 5 minutes. The iterations was kept at 1000 which produced around 83% accuracy but since this value saturates at around 85% anyway, the marginal benefit is too less.

6. K-Nearest Neighbors: KNN was also performed on the same dataset as classification.

- Firstly, the Euclidean distance of each point with respect to all other points are calculated and stored in their respective array.
- The array is sorted and the first k elements are picked up from it. This signifies the k nearest points to that point. We also keep track of the index of each distance point.
- Subsequently the corresponding target value of the k nearest point is picked up with the help of the index and since it is a classification data their mode is calculated. This mode is assigned to that particular set of input features.
- To better predict the accuracy, I did not run the verification on the train data but on 60% of the predicted test data already produced in logistic regression. This is because the optimal value of k highly depends on the dataset and it will not be beneficial if we do the accuracy test on the train data.
- Finally, for one value of k the result is produced in about 17 minutes. To predict the optimal value of k, I ran a loop for k from 50 to 150 with an increment of 10. Although the accuracy does

not vary by much, k=50 turned out to be optimal in this range with an accuracy of 86.3%.

7. Neural Network: Neural Network is performed on the classification dataset. It is directly an n-layer implementation.

- At first, I initialized the parameters w and b with 1. I also decided to use Rectified Linear Unit (ReLU) activation for all the hidden layers and sigmoid activation for the last layer as this is the most common combination for image classification tasks.
- I wrote the functions for the sigmoid and ReLU activations and also the derivative of the functions which will be required later.
- Next, I moved on to the forward propagation which uses the standard WA+b formula for all neurons and this value is then passed to the corresponding activation function. For each layer, we also store two types of caches namely linear cache (which stores the value of A, W and b) and activation cache (which stores the value of Z). This value will be required to calculate the derivative later on in backward propagation. [Z=WA (prev.) +b and A(next)=f(Z) where f is either sigmoid or ReLU]
- Then I wrote a separate function which calculates the derivative of A, W and b according to the formula dw=(1/m) *dZ*A(prev.), db=(1/m) *sum(dZ) and dA(prev.) =W*dZ. Here dZ=dA*f'(Z). Point to be noted here is that the proper dimension matching of all terms is done by proper usage of transpose wherever required.
- I proceeded with the backward propagation which calculates the derivative of A, W and b in each layer and store that in a dictionary named as grad.
- The final step was to implement all of this together i.e., for each iteration, I forward propagate to get the cache of each layer and the predicted value of the target variable. I then pass this data

to the backward propagation to get the derivative in each layer. This is then used to modify the parameters W and b.

- Accuracy is then calculated on a 20% split of the train data and the graph of accuracy v/s number of iterations is plotted.
- As for the making of neurons in each layer, I decided to extract the number of input features and keep dividing it by 3 in each layer until I get 10 neurons left. I applied ReLU activation in these layers and finally added another layer with 10 neurons and sigmoid activation. Hot encoding of the target variable is also done similar to that of logistic regression.
- Initially on execution, my accuracy was coming out to be too low (around 15%) no matter how I tuned the value of learning rate and number of iterations. At first, I thought it must be due to not using feature scaling but the result did not change much even after using normalization or regularization.
- After printing out the values of W in each layer I saw that they were changing uniformly and did not differ from each other which indicated that my initialization method was incorrect. So, I implemented the He-et-al initialization method for W which first assigns a random value to the variable and then multiplies it with the square root of 2/ (no. of neurons in previous layer + no. of neurons in that layer).
- This initialization method along with some tuning of other hyperparameters like learning rate and number of iterations finally improved the accuracy to around 86% for 500 iterations, learning rate = 0.1 and lambda of regularization = 0.01, and the time required is about 15 minutes.