ML Boot Camp

WOC 6.0

By: Hemant Sharma(23JE0400)



Project report

Introduction

This project aims to develop a basic, user-friendly machine learning library tailored for educational and prototyping purposes. The goal of our project is to create a fundamental machine learning library from the scratch, using only freely available libraries such as NumPy, Pandas, and Matplotlib

Dataset Overview

This project included a wide range of tasks in both the regression and classification areas, with datasets tailored to each. Two unique datasets were provided for regression tasks: one for linear regression and one for polynomial regression. The model confronted the problem of evaluating from 20 features in the linear regression dataset, with the goal of learning the underlying patterns and relationships in the data. The polynomial regression dataset, on the other hand, included three characteristics, forcing the model to capture more sophisticated nonlinear connections. Working with pixel-by-pixel representations of the image data gave an interesting twist to the categorization job. These images comprised numbers ranging from 0 to 9, and the goal was to train the model to recognize the presented numbers based on the pixel data provided.

Linear Regression Implementation

Dataset: Linear regression dataset

Features: 20

Hyperparameters:

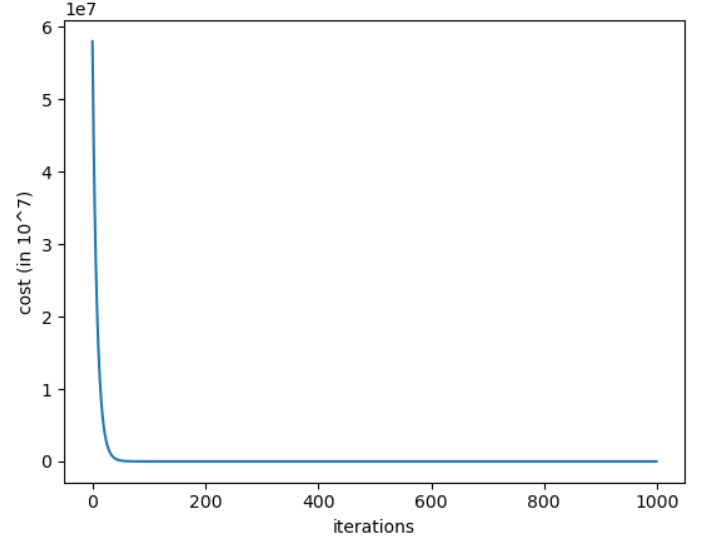
Learning Rate: 0.06

Epochs: 1000

R2 score for training examples: 0.9999999999230841

R2 score for cross-validation examples: 0.9999999999221727

Time taken: 1.3s



Plot of cost function vs. iterations

I dug into the data, attempting to make sense of it all by fitting a straight line with a simple function and dabbling in the gradient descent stuff. I thought it was interesting to normalize the X data using Z-score, however it didn't work out as planned. This got me thinking: perhaps the data was already linear.

To put my suspicions to the test, I decided to abandon the Z-score approach. Surprisingly, it made a huge difference. Without normalization, the model performed exceptionally well, with an R2 of 0.9999995859747042 on the training set and 0.9999995719017323 on the cross-validation set.

Had my fair share of struggles, especially at the start with those nested loops in the gradient descent. They were dragging everything down, and I needed a way out. That's when I decided to implement vectorization, picking up the technique from Andrew Ng in the supervised machine learning course.

While tinkering with things, I played around with different alpha values, and settled on 3.6e-5 as the sweet spot. Split the data into 40,000 for training and 10,000 for cross-validation. The model did more than just good, with an R2 score of 0.9999995719017323.

But the true eye-opener came when I realized Z-score normalization was not helping me. Sometimes you just have to trust your gut. Taking it out provided me extremely positive outcomes. Lesson learned: modeling is about more than just numbers and code; you also have to trust your gut.

Polynomial Regression Implementation

Dataset: polynomial regression dataset

Features: 3

Hyperparameters:

Learning rate: 0.28

Iterations: 10000

Lambda : 0

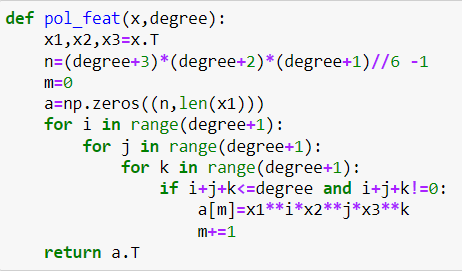
Degree : 6

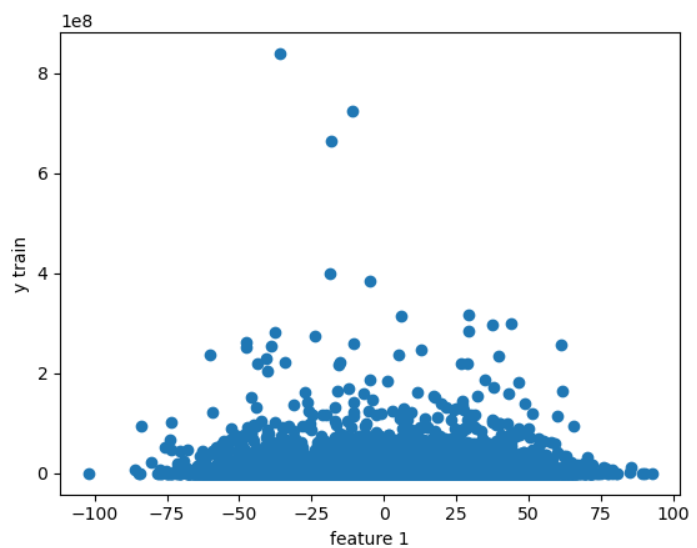
R2 score on training: 1.0

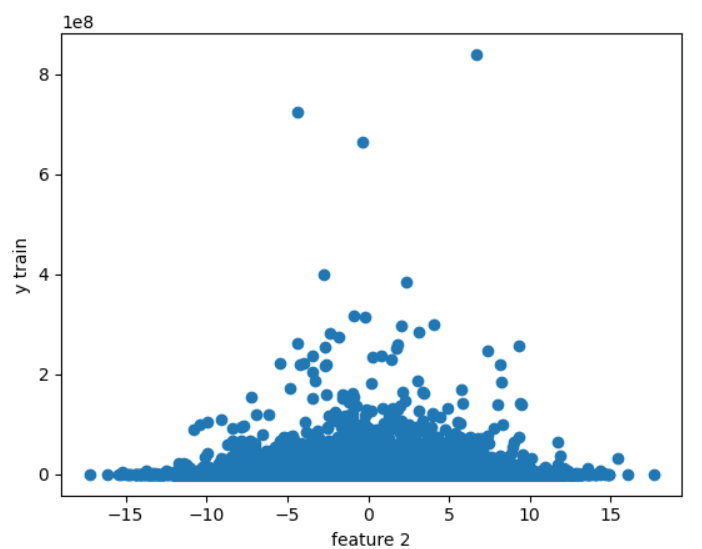
R2 score on cross validation: 1.0

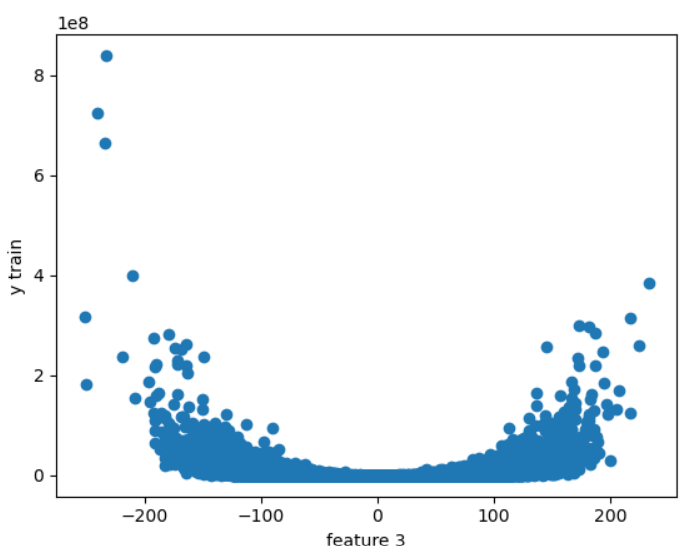
Time taken: 58.4s

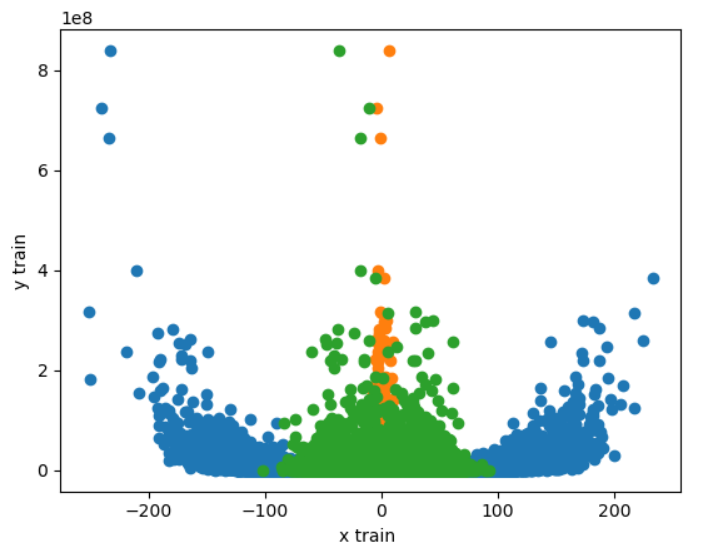
The first difficulty was to calculate the degree of the polynomial, so I began plotting the graph of each feature against the label y and obtained the following graphs and observed the that the graphs were symmetric about y-axis and estimated that the data to be one of even degree and beagn working on polynomial for even degrees. I wrote a function to build a polynomial equation for different degrees.











In my pursuit of determining the optimal degree for my polynomial model, I initially attempted to assess the performance for each even degree, anticipating incremental improvements. The expectation was that increasing the degree wouldn't significantly impact the results, as the coefficients would ultimately approach zero during gradient descent. However, contrary to my expectations, I did not achieve the anticipated improvement in scores, and the computational demands became notably slower for higher degrees. Consequently, I ultimately opted for a polynomial degree of 6 as a reasonable compromise.

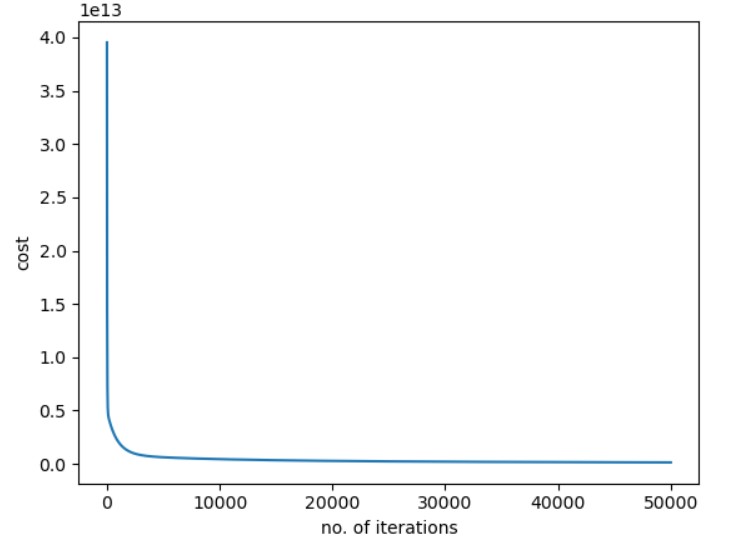
To address potential overfitting, I introduced regularization in my model. Although not critically needed for this dataset, I retained it for the sake of generalization, opting for a common lambda value of 0.1.

I experimented with different weight and bias initializations, finding that the most favorable outcomes occurred when both weights and biases were initialized to zeros. However, despite the success with zero initialization, my mentor advised against it. Following this guidance, I chose to initialize with random normal arrays, setting the values to a very low range. This alternative approach still produced positive results.

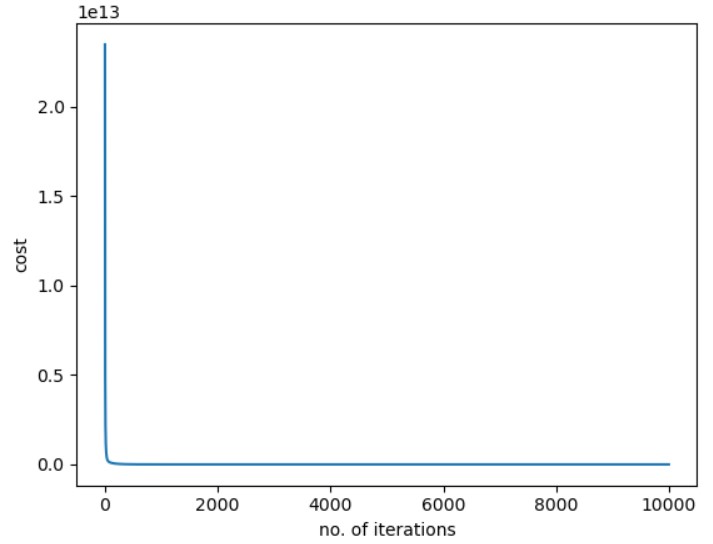
Initially, I did not get a good R2 score; I got 0.97 on training and 0.95 on cross validation. I assumed it was due to overfitting because there was such a large difference between my training and CV scores, and I implemented both l1 and l2 regularization to address this issue, but the model continued to perform poorly. When I decided not to do the normalization phase, the results were astounding—both R2 values were higher than 0.98. After additional adjustment, the R2 values were even higher than 0.99. It turned out that the data had a significant intrinsic correlation even before normalization, which made it easier for the algorithm to run without the normalization step.

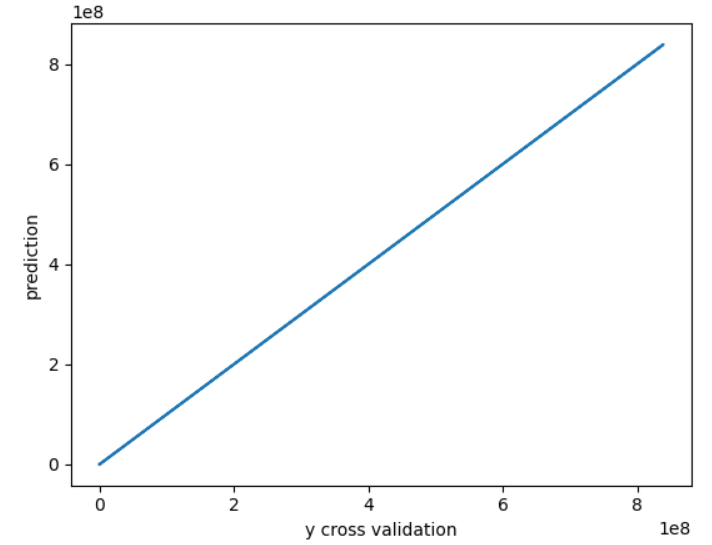
An initial learning rate of 0.01 became troublesome upon evaluation, resulting in excessive oscillations. The price increased so much that squaring it caused an overflow and the display became illegible with NaN numbers. After that, it was beneficial to change the learning rate to 1.6e-25, which resulted in a steady drop in cost and an excellent R2 score of 0.9976 on the cross-validation set. Two subsets of the dataset were split: 10,000 for cross-validation and 40,000 for training.

Subsequent testing revealed that the score was not significantly affected by adjusting lambda to 0. I firmly set lambda at 0 because there were no signs of excessive variance or overfitting, as seen by roughly identical R2 values for both training and cross-validation. Gradient descent worked smoothly as a result of this calculated decision, producing forecasts that were quite similar to the actual results.



I was utilizing the incorrect normalization implementation, as I found out when I evaluated the normalizing function. This error clarified the previously noted low R2 scores and significant cross-validation costs. After that, A significant improvement in cost convergence was seen when the normalization method was corrected and the dataset was divided into training and testing sets. The cost stabilized at less than 1. R2 values for testing and training both approached 1.0, demonstrating the model's exceptional accuracy. Now with 10,000 iterations and a learning rate of 0.28, the model operated effectively, finishing the task in 58 seconds. Interestingly, the smoothness of the cost and the lack of oscillations confirmed the improved performance of the improved model.

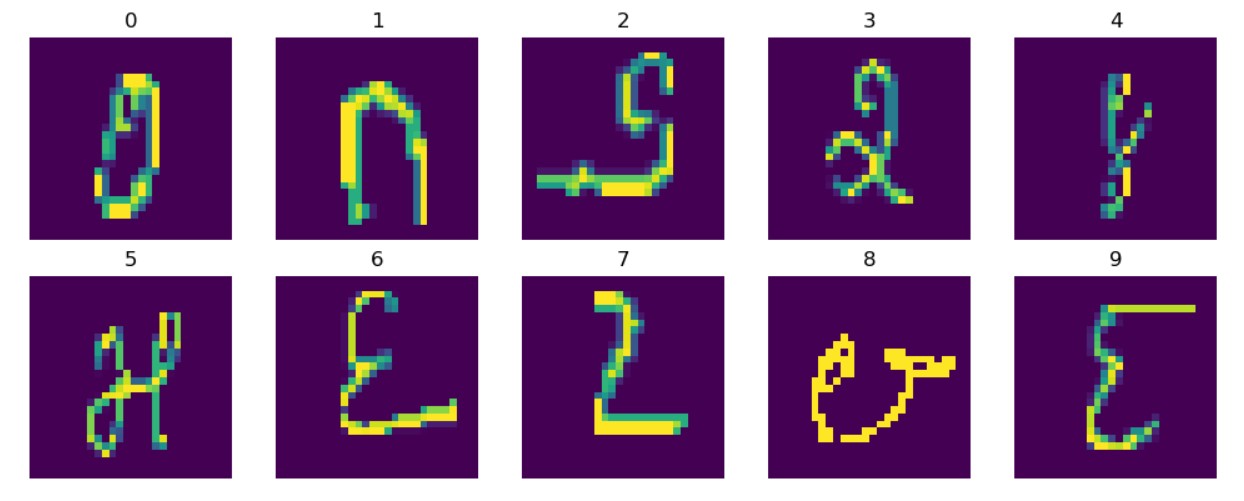


Here’s the graph of actual y values of cross validation set and predicted values:

Classification

Our project consisted of working with a dataset that was quite similar to the well-known MNIST dataset. The major aim was to create multiclassification models capable of detecting pictures with digits ranging from 0 to 9. Each picture in the collection was represented by a 28x28 pixel grid, and the given data included information about the intensity or colour values of each pixel.

To achieve this categorization aim, we implemented a variety of models, including logistic regression, k-nearest neighbours (KNN), and a multi-layer neural network. The emphasis was on using these models to accurately categorize and detect numerical content within the pictures. This project sought to demonstrate the adaptability of machine learning algorithms for handling multiclassification problems on picture data.



Logistic Regression Implementation

Learning rate: 3.2

Epoch: 100

Mini batch size: 128

Training accuracy : 0.9815

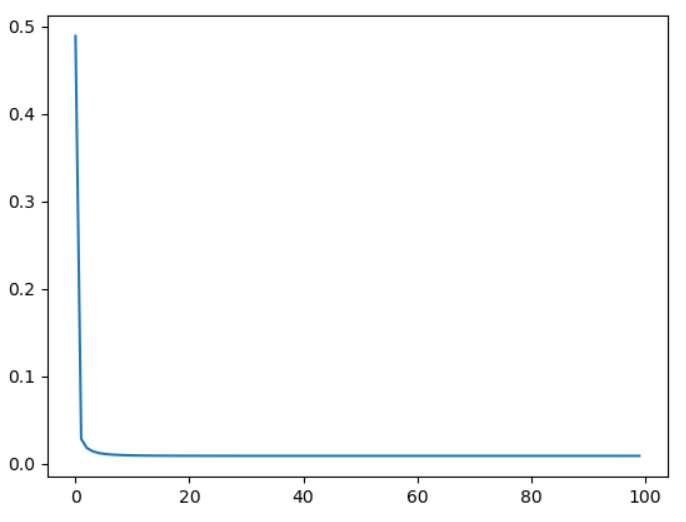
Cross validation accuracy: 0.967

Time taken: 22.6s

Initially, my grasp of logistic regression was limited to binary classification with the sigmoid function. However, when presented with a dataset that required multiclass classification, I conducted study to update my expertise. I found the one-vs-all method, which is a realistic approach to multiclass logistic regression. To address this, I used one-hot encoding for the target variable, converting it into a format suitable for multiclass classification.

My first approach consisted of ten independent loops using np.where to do distinct categorization procedures for each category. Despite its effectiveness, it was time demanding. I then improved the code to vectorize these processes and then I started evaluation with setting learning rate to 0.01 but the cost was decreasing very slowly so I started increasing the learning rate and was determined to choose the maximum learning rate I can use before overflow occurred and determined it to be 3.2 . Furthermore, I investigated the advantages of mini-batch gradient descent, which improves accuracy with fewer iterations, I was able to achieve highest accuracy on batch size of 128 with appropriate time.this batch size was neither too small nor too large so neither there was issue of much overfitiing nor was there much error in calculation of gradients.

With these improvements, the algorithm attained a training accuracy of 0.9815 and a cross-validation accuracy of 0.967. This evolution demonstrated the usefulness of the customized logistic regression model for multiclass classification, with a significant improvement over the early phases of implementation. The iterations vs cost graph was as follows:



KNN Implementation

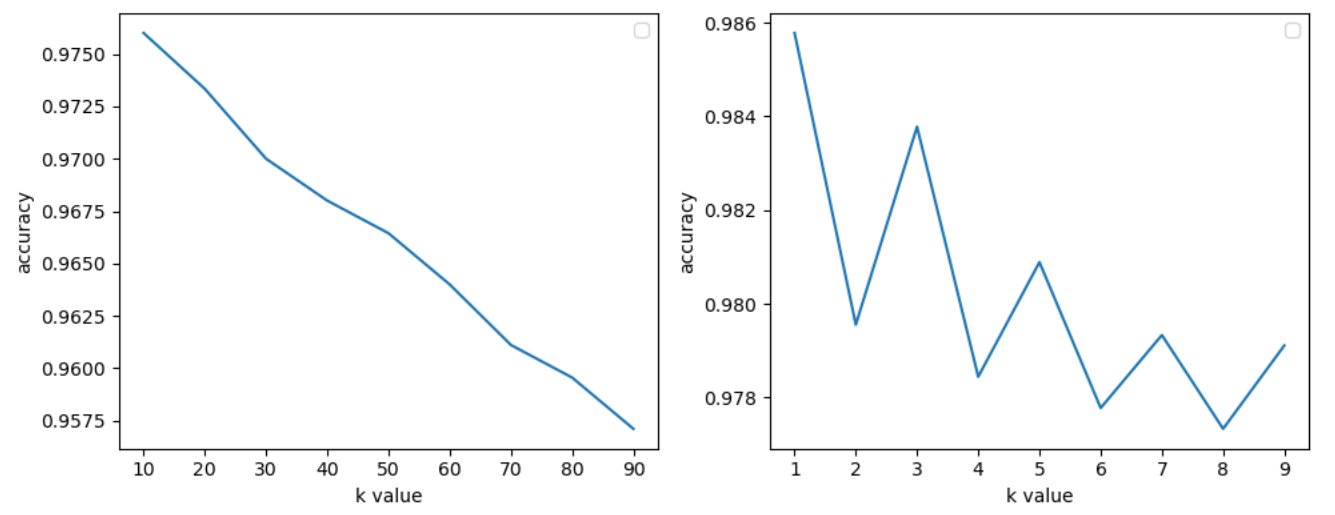
My primary objective in developing the K-Nearest Neighbors (KNN) method was to identify the optimal hyperparameter, K, for precisely classifying new data points. I appreciated the simplicity of not having to contend with various parameters since the model did not involve any training or fitting processes. However, things took a challenging turn.

I began my exploration of the K-Nearest Neighbors (KNN) algorithm with a value of K set to 3, as guided by instructional videos. At the outset, I grappled with the task of calculating distances between points in both the training and cross-validation datasets. My initial implementation lacked vectorization and relied on loops for each data point, resulting in a prediction time of approximately 30 minutes and an accuracy of 0.9620.

I attempted to vectorize the entire operation by extending the computation into the third dimension to calculate distances. However, memory errors occurred as the RAM filled up due to the formation of large matrices with a shape of (5000, 25000, 784) during distance calculation. To address this issue, I implemented a mini-batch strategy. I utilized loops to process only 100 datapoints at a time for prediction, iterating through 50 loops. This approach allowed my computer to handle the computation efficiently and complete the operation in under 7 minutes. The resulting accuracy achieved for K=3 was 0.9811.

Now, Dissatisfied with the initial computation time of the code, I sought to optimize its efficiency. During my research on effective methods for calculating distances between points, I discovered a valuable insight in a PyTorch forum.

Leveraging the mathematical identity **(a-b)\*\*2 = a\*\*2 + b\*\*2 - 2\*a\*b** , I implemented a more efficient approach to compute distances between points in the dataset. This change sought to simplify the code and improve its overall efficiency. Now the code was working under **20 seconds** and gave me test accuracy of **0.9838** for 5000 test points split from the training dataset divided into 1:5 ratio. Now to check the hypothesis of putting **k=3** for getting best accuracy, I plotted graphs of k vs accuracy and got the following results:



While k=1 yielded the maximum accuracy, worries about outliers prompted the choice to choose k=3. Outliers, or anomalies, have the ability to alter forecasts, especially when located near our data points. Although this dataset did not include any outliers, the cautious choice of k=3 was intended to reduce the influence of any outliers that may lead to erroneous predictions. This choice demonstrates a proactive strategy to prepare for instances in which the existence of outliers may jeopardize the accuracy of the algorithm's predictions.

‎N-layer Neural Network Implementation

Hyperparameters:

Learning rate: 0.05

Epoch: 30

Layer structure : [128,64,32,10]

Batch size: 32

Accuracy on training data: 1.0

Accuracy on cross validation dataset: 0.985

Time taken: 1m 32s

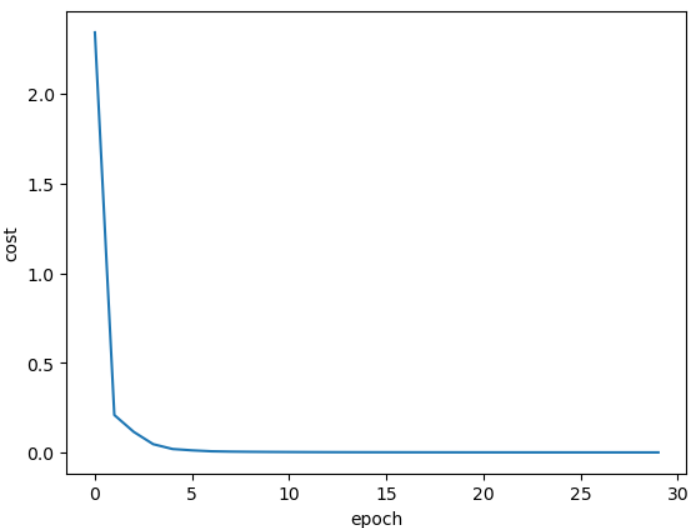
I started by carefully watching instructional videos to gain a grasp of neural networks and then I implemented a basic neural network from scratch. My initiative was motivated by a complete comprehension of the workings of the neural network, which was promoted by self-derivation of backpropagation equations and reference to 3Blue1Brown's deep learning series.

Initially, overflow issues were caused by random weight and bias initialization. This led to a use Xavier's uniform initialization. Backpropagation was implemented in conjunction with a side-by-side updation algorithm to guarantee continuous improvement with each repetition.

The creation of an adaptable neural network with n layers required flexible layers that could accommodate different counts of neurons and layers. I implemented lists to hold each layer's outputs and gradients in order to overcome the backpropagation problem and allow for dynamic computations and updates. The algorithm's flexibility was improved by adding loops for iterative layer-wise calculations.

When the algorithm was run, a puzzling problem surfaced: it continually yielded the same answers with an accuracy of a mere 0.10. An incorrect use of softmax derivatives was found during a thorough examination of the backpropagation equations, leading to a correction and eventual improvement to an accuracy of 0.86. Even with this development, the algorithm performed slowly; training took around fifteen minutes.

Using a batch size of 32, I developed mini-batch gradient descent to speed up calculations and reduce the chance of overfitting. Using a small batch size allowed for quick calculations, and each iteration's exposure to fresh data improved model's adaptability and reduced the chance of overfitting. Initially learning rate was taken to be 0.01 and trained for 10 epochs. Model performance was further enhanced by careful reevaluation and modifications to the learning rate and number of epochs. A significant improvement in accuracy occurred to reach 1.0 on the training dataset and 0.985 on the cross-validation dataset after setting learning rate to be 0.05 and trained for 30 epochs.



Investigating optimization methods like RMSprop resulted in undesired time extension and produced no appreciable benefits. As a result, I decided to leave RMSprop out of the optimization plan. At the moment, the neural network runs in less than 1.5 minutes and achieves a remarkable accuracy of 0.985 on the cross-validation dataset and 1.0 on the training dataset. This trip has reinforced my commitment to fine-tuning model performance via ongoing learning and careful optimization.

‎K-Means Clustering Implementation

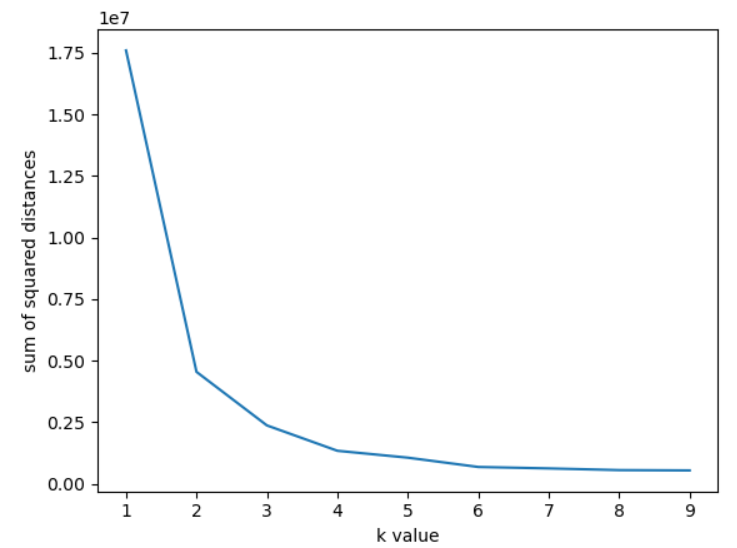
Hyperparameters:

Optimal K : 6

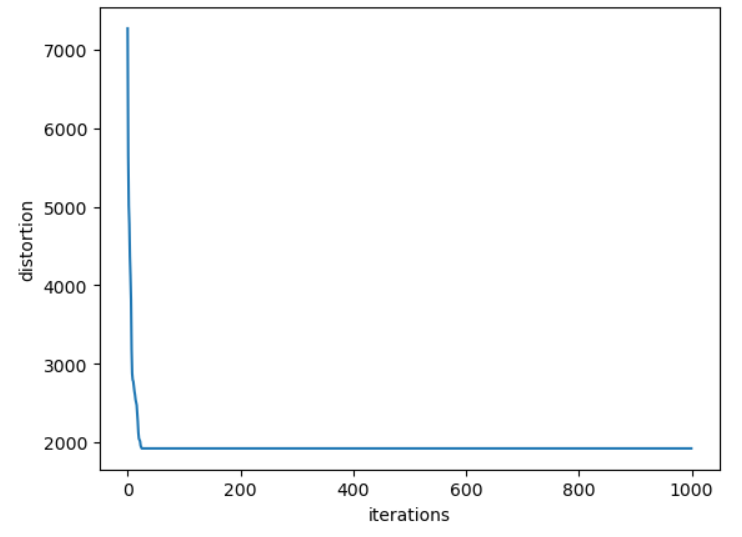
Iterations : 10000

K-Means is an iterative clustering technique that divides a dataset into K unique, non-overlapping subsets (clusters). It works by grouping data points into clusters based on their closeness to centroids, which are then repeatedly modified to reduce the within-cluster sum of squares.

The elbow method was used to determine the optimal value of the hyperparameter k in the K-Means algorithm. This method includes training the K-means model with various k values and then evaluating its performance on a validation set. The resulting accuracy scores were plotted against the matched k values to generate a graph. The graph's "elbow" depicts the point at which further increases in k result in a decline in accuracy improvement. This clearly defined elbow point serves as the optimal k value for balancing model accuracy and simplicity. To ensure trustworthy performance, the chosen k value was validated on an independent test set. This methodical approach provides a reliable way of hyperparameter tuning for the algorithm.

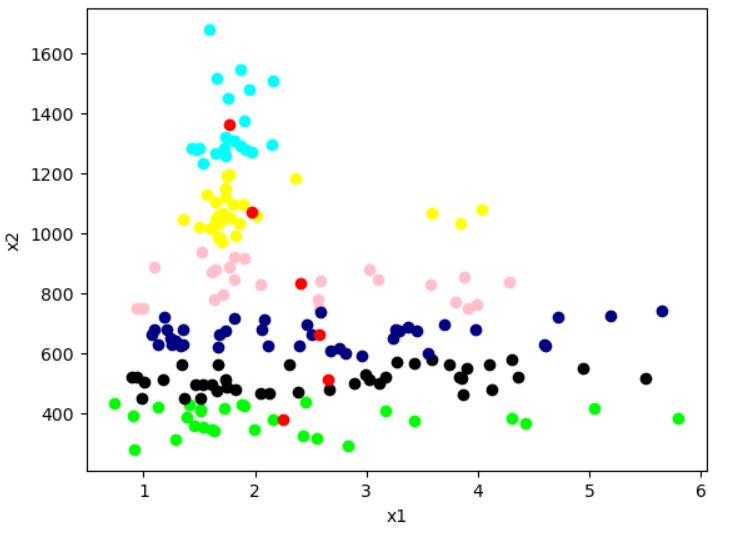


After discovering that the ideal value for k in the K-Means method was **6**, the next step was to develop the K-Means clustering algorithm as advised by Andrew Ng. The algorithm was programmed, and centroids were initialized using the dataset points. The model carried on through 1000 iterations, and the distortion function gradually dropped until it achieved a stable value of 1921.49.



This observation indicates that the K-Means method has converged, since more iterations did not result in substantial changes to the distortion function.

I enhanced the analysis by ploting two randomly selected characteristics, as well as centroids. This graph provided a simple picture of the spatial distribution of data points and the critical function of centroids in establishing separate clusters. This visual aspect allowed for a clearer comprehension of the clustering findings and gave vital insights into the dataset's underlying trends.



(Red Dots are Centroids)

‎