Data Analytics And Visualization

ASSIGNMENT-1B

Fruit Classification: Which Fruit is this?

Ishika Jaiswal (11940510) , Anmol Singhal (11940140)

Introduction

We have implemented the Fruit Classification using 3 methods :

1. Using Learning with Prototypes Method
2. Using K-Nearest Neighbors Classification Method
3. Using an Artificial Neural Network

For the multi-fruit dataset, we have collected 50 images that we have submitted in the link provided in the mail along with a spreadsheet with the corresponding labels.

Using Learning with Prototypes Method

Pre-Processing: We have converted all the images in sizes of 100x100 and then calculated their HOG and color histogram as features. The color histogram explains the color distribution and the HOG tells us about the edges to give information about the shapes.

Initially, we had tried to use raw flattened images (with mean taken for r,g,b values) as features along with the color distribution, but then we shifted to the current implementation for better accuracy.

To Calculate the Color Histograms and the HOG, we have used the open-cv and skimage libraries in python.

Classifier Class:

We have created a class named “LearningWithPrototypes” using which we will create the Machine Learning Model.

Our Class has the following functions:

1. **fit:** This function takes a training feature set and its label set for training and calculates the prototypes on the basis of the training data. To calculate the prototypes, we have simply taken the mean of all data points in a class as its prototype.

   To store the prototypes, we have used a dictionary as an instance variable. The coding part is clearly explained in the google collab along with proper commenting.

2. **predict_E:** This function takes as input multiple feature vectors to predict their label using the euclidean distances from the prototypes.
3. **predict_C:** This function takes as input multiple feature vectors to predict their label using the Cosine distances from the prototypes.

We were trying to compute cosine distances rather than euclidean distances to try to improve the accuracy but it turned out to be that euclidean distances are working better.

Steps we took to improve Accuracy :

1. **<u>Label Merging:</u>**

- ***Why we did it*:** In our model, Apple Yellow 1 was being predicted as Apple Golden 1 which was very close to the real output so we thought if we just match the core fruit, and check the accuracy on that basis, we should be better off in terms of accuracy. So we changed the labels in the training dataset as follows:

  Let a label name be: <Core_Fruit_Name> <Variety> .. then we just replaced it with the <Core_Fruit_Name>.

  So basically, we merged all the apples and so on for all the fruits.

  But, it turned out that the accuracy significantly decreased.

➔ On thinking about it for so long, also discussing it in the class, we found out that the accuracy decreased due to a significant shift in the means which was caused due to the following type of examples:

★ We are taking color distribution as a part of our feature vector and we know that yellow as well as red apples exist in our dataset. So, on merging the two

the mean of the color distribution shifts towards the color orange which interferes with the prototype for oranges. So, an orange can be misclassified as an apple and such cases were causing a significant decrease in the accuracy of our model.

2. **Using Cosine Distances:**

As mentioned above in the report, we tried using cosine distances rather than euclidean distances but it also caused a decrease in our model accuracy since the length/magnitude of our feature vector is also very important for our use case because of the following reason:

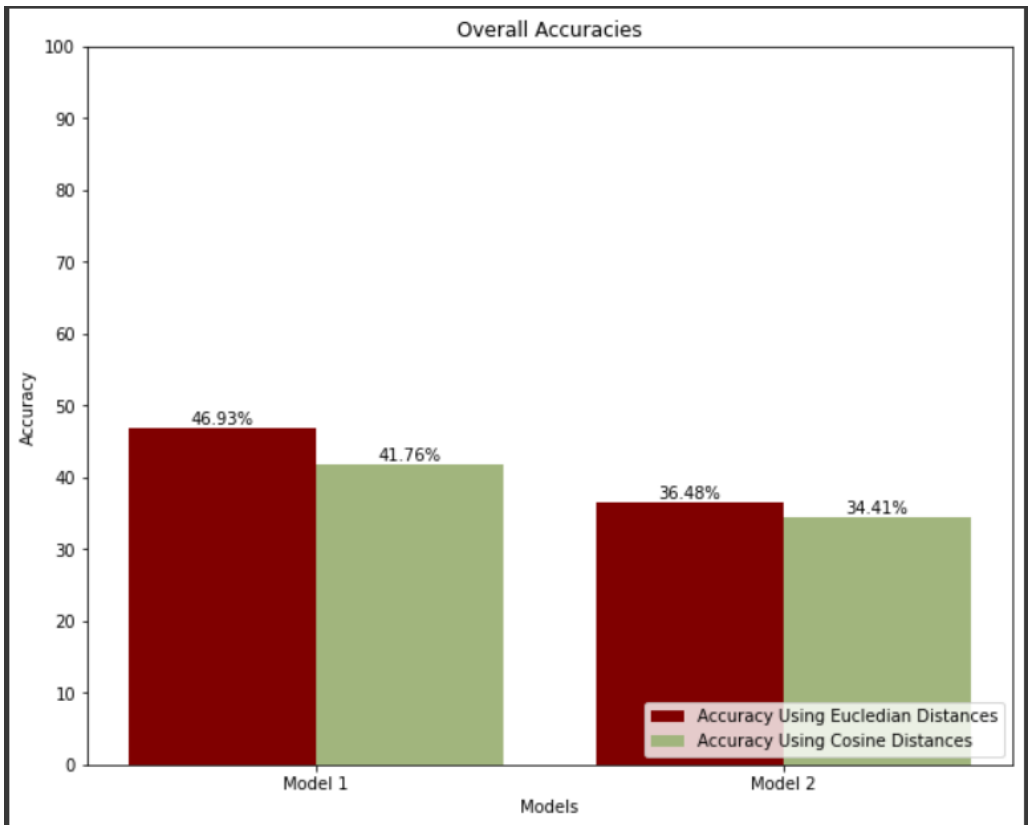
- ★ It is possible that the data points of some fruit-1 are clustered in the same direction as some other fruit-2. But both the clusters have a huge difference in their distance from the origin. If we take only cosine similarity, we end up mixing the clusters which might cause fruit-1 being misclassified as fruit-2 and vice-versa. Hence, taking Euclidean Distance is giving a better accuracy.

Comparison Charts :

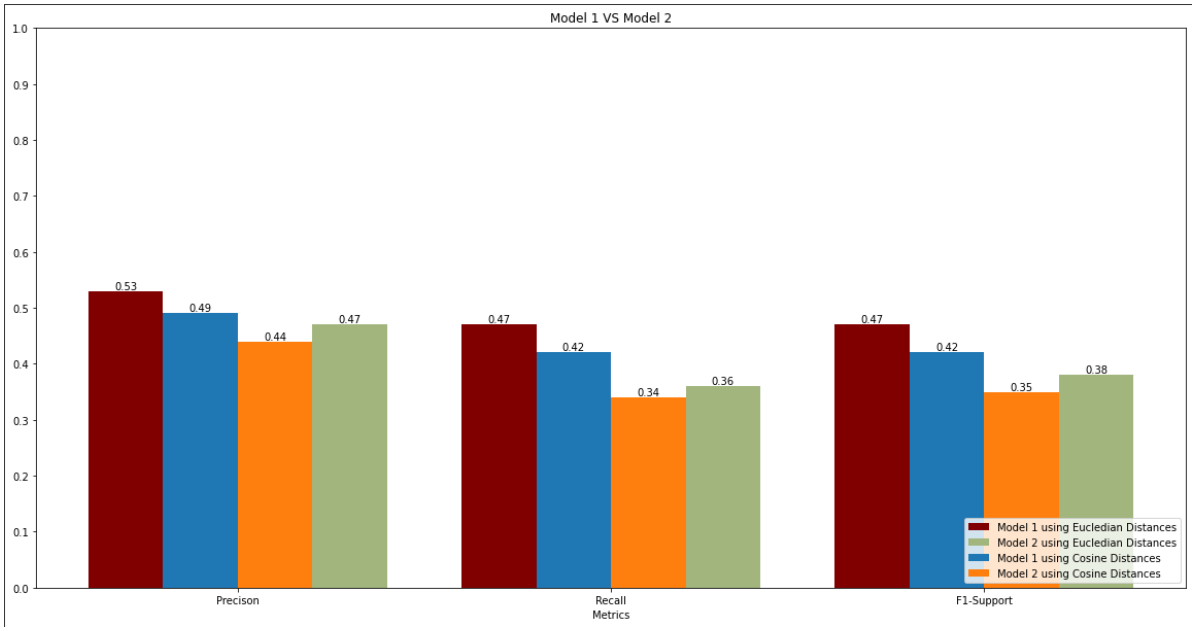
Model - 1: Model Trained on Original Labels

Model - 2: Model Trained on Merged Labels

1. **Accuracy Comparison**



2. Other Metrics Comparison



Google Collab with Complete Code with Step by Step Explanations : **LINK**

Using K-Nearest Neighbors:

Pre-Processing is the same as the above part.

Initially, our KNN Classifier class had three functions:

1. **Fit:** The model does not learn from training data to predict output, instead uses entire data to predict output for new data. Therefore, our whole feature vector and labels are stored as instance attributes.
2. **Predict:** This function is to predict the label of the new Image with the help of predict_one function.
3. **Predict_one:** In this function, we use euclidean distance to calculate the distance between new data and all training feature vectors, and then assign the label to the mode of k- shortest distances.

However, when we were doing preliminary testing for hyper-parameter tuning. Our code was taking a long time for each value of k since it was calculating the distance between all training feature vectors and the test feature vector again and again.

Therefore we have made 2 more function optimize the time taken during the preliminary testing:

1. **Predict_hyper_tuning :** For making predictions for a set of features during the hyperparameter tuning on a fixed validation set
2. **Predict_hyper_tuning_one :** For making prediction for a single feature during hyperparameter tuning on a fixed validation step

Basically, in the Predict_Hyper_Tuning function we are saving the distance matrix (distance from each feature to each datapoint) as the distance between training feature vectors and

test feature vectors is the same for all **k**. Therefore, after the first iteration of **k**, it just takes **O(k) time per feature vector** as it just extracts the first k neighbors from the existing distance matrix**.**

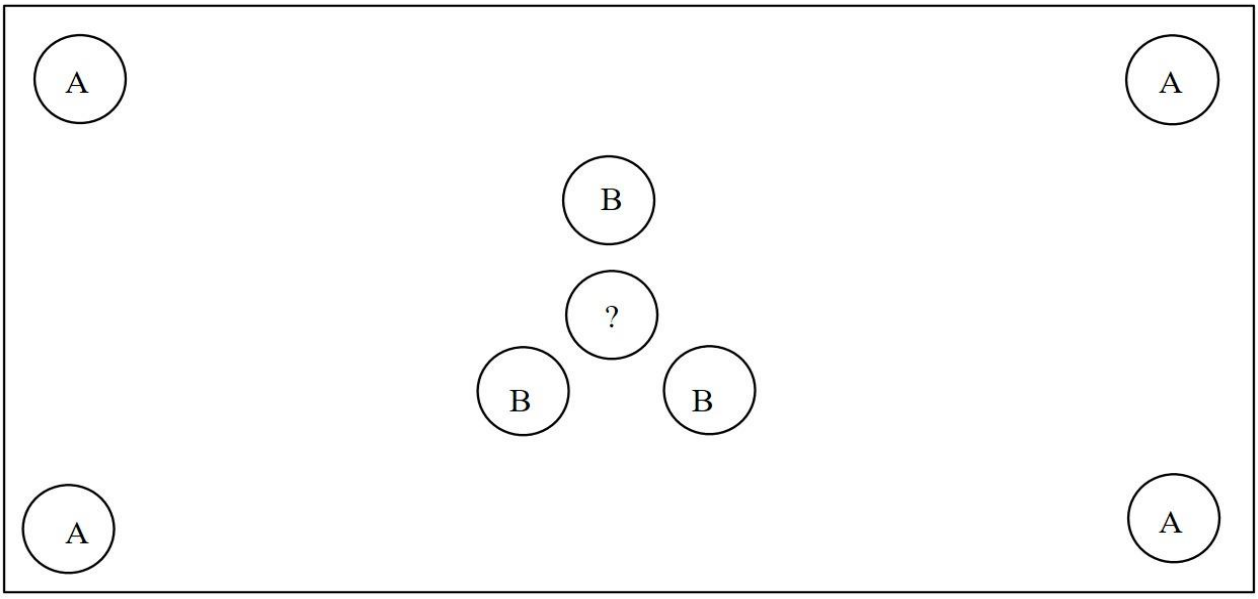
Also, when we tried to run our code for the complete dataset, we were facing the ***RAM crash***. Therefore, we are sampling the dataset and training our model on only 100 images per class.

We also compared our model to Google's skLearn KNNClassifier, and found that while both models are providing almost the ***same accuracy***, the skLearn model takes significantly less time than our custom implementation which can be due to various types of optimizations.

To improve the accuracy of our model, we took the following steps :

Introduced Weighted Prediction :

Intuition: If we have, let's say, chosen **k = 7**, then in that case suppose this is the situation:



In this case, it seems better to predict the output as **B** rather than **A.**

But, an unweighted KNN will give the output as **A.**

Hence, we need to associate higher weights with nearest neighbors.

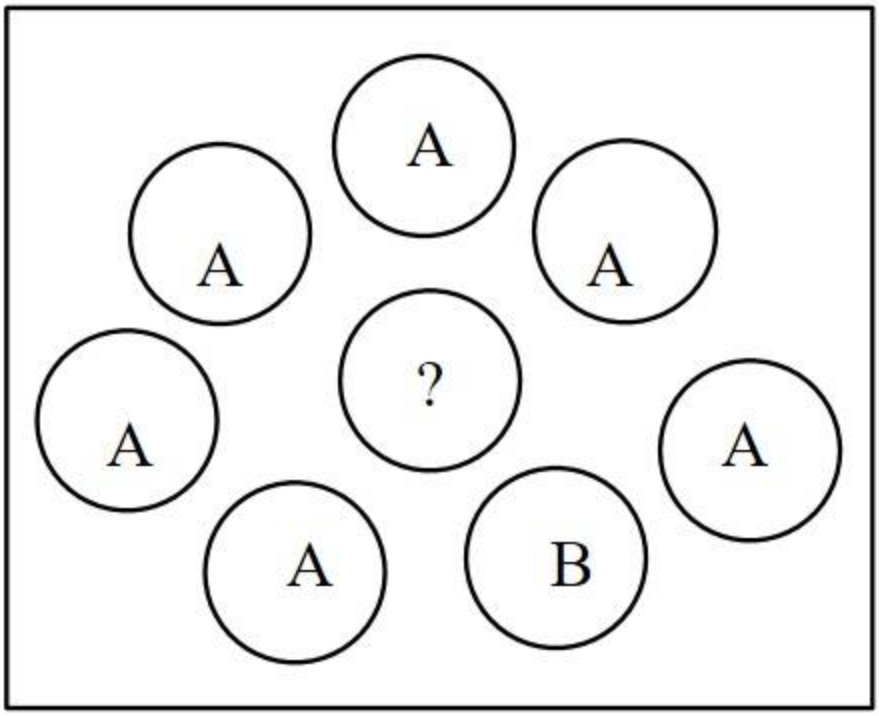
Or alternatively, associate lower weights with nearest neighbors and then pick the label which minimizes the net weight.

Hence, in the Weighted Prediction:

1. We select the first K nearest neighbors
2. Then we associate the weights with them by considering the distance of the data point to be the weight associated with it
3. Then we select the label '**L**' such that to minimize the summation of weights of data points associated with that label.

The Complete Code is written in the google collab with proper commenting.

In doing this, our accuracy ultimately decreased probably because of such cases:



Here, it is better to predict **A** than **B,** But our weighted prediction will predict it to be **B.**

This method lacks to consider the difference between the label frequency for different labels in our **k nearest neighbors**. Hence, we also see that accuracy using this weighted method decreases at a high rate as we **increase** the value of **k.**

Idea to Try in Future: If one label has a high frequency, then it will also have a higher weight sum, but we prefer having a higher frequency. So we can try to minimize the sum of weights of data points divided by the label frequency rather than what we have implemented.

The coding part is well-explained in the google collab with proper commenting.

Introduced Tie Breaking :

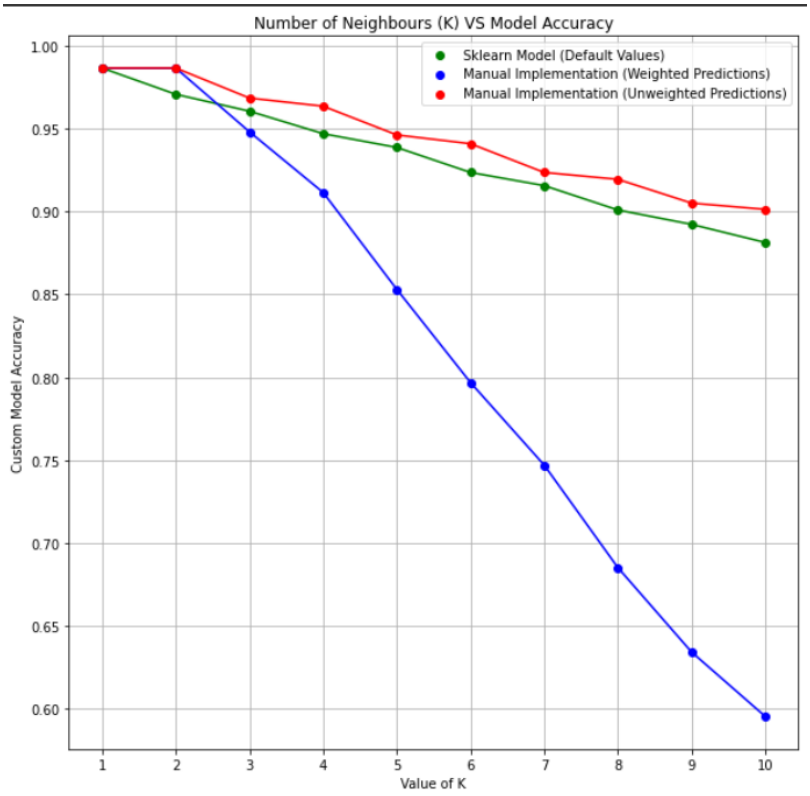
Here, our first priority is maximizing the frequency, we will only check weights when there's a tie in the frequencies.

In case of unweighted prediction, suppose we have **k = 6** and amongst the 6 neighbors, 3 belong to class A and 3 belong to class B.

In such cases of tie, we are selecting the class which minimizes the sum of distances from test point to the data points.

This significantly improves the accuracy of our model and our model is giving even better accuracy than the sklearn model if we use the sklearn model with the default values.

This was the comparison chart after the hyper-parameter tuning for **k** values in the range [1, 10] both inclusive:

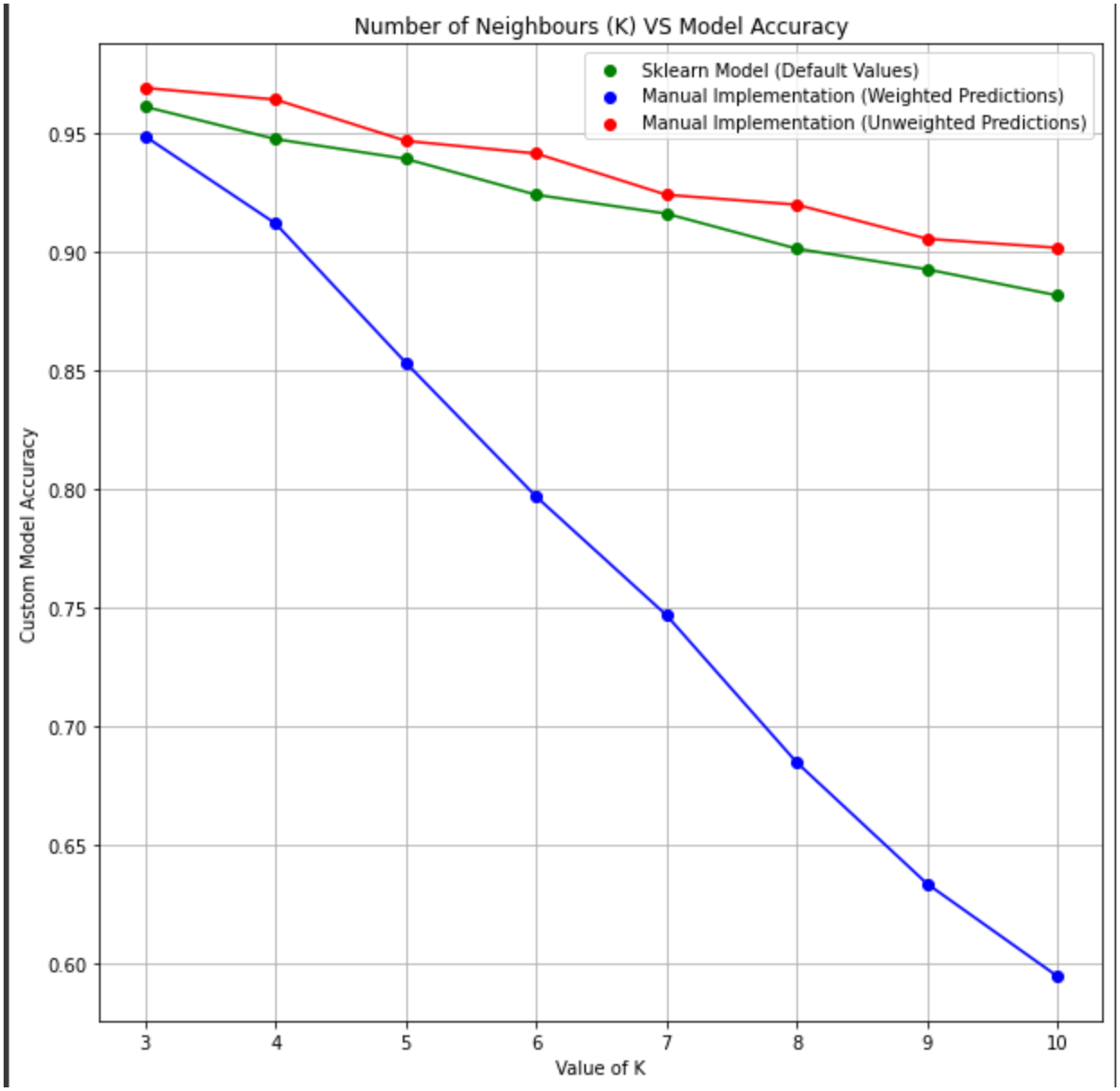


As we can see, all the models were giving best performance at **k = 1,** But using k = 1 in a real life scenario is very ***susceptible to noise and outliers*** in the data. This can cause huge changes in the accuracy based on the data distribution. Hence, we tuned the hyper-parameter **k** only on values [3, 10].

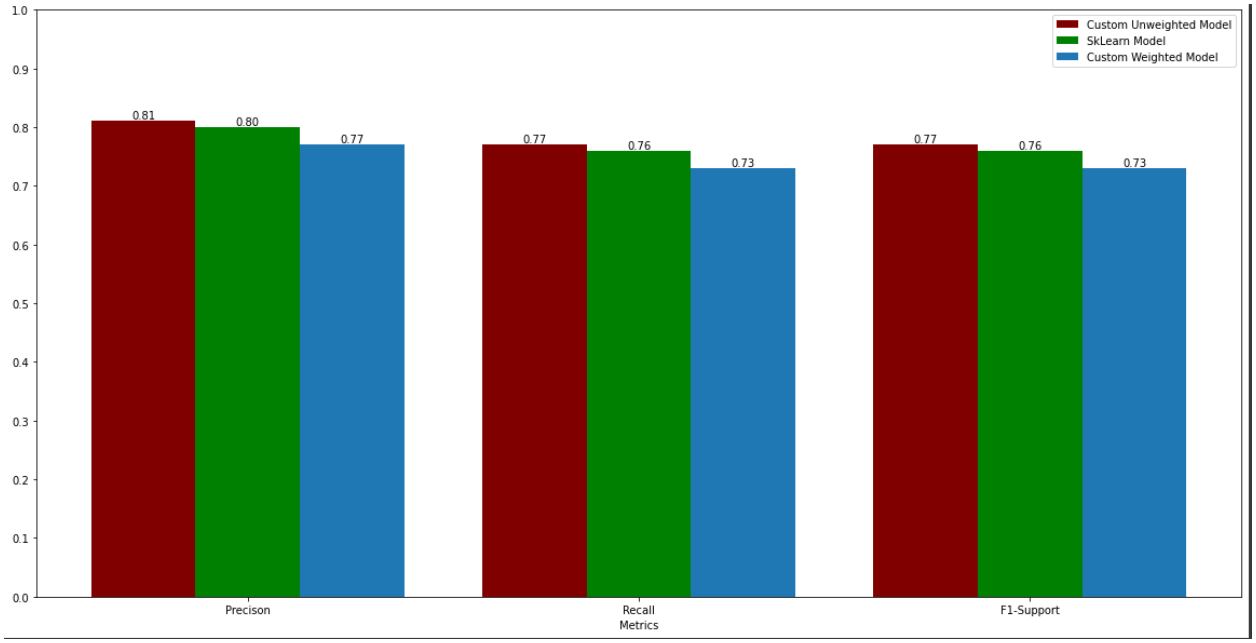
For the testing purposes, we sampled the database as 50 images per class.

Comparison Charts :

1. **Accuracy for different values of K during hyper parameter tuning**



2. Metrics Comparison over Test Images



Complete Code with proper commenting for 1.B : **LINK**

Using Artificial Neural Networks

Pre-Processing is the same as the above parts except for the following exception:

We have one-hot encoded our labels array before feeding it to the network successfully. One Hot Encoding converts our labels into binary arrays (i.e. arrays consisting of 0's and 1's).

Example: Labels Array: ["A", "B", "C"]

➔ One-Hot Encoded Labels: [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

Also, we are sampling the database by taking 200 images per class for the training set and 50 images per class for the test set since when we were training and testing on the full set of images, our **RAM** was crashing.

We have used the TensorFlow library to create and train our network. First, We made the following changes in the starter code:

1. We took the input shape to be equal to the length of a single feature vector.
2. We changed the activation function in the last layer to be 'softmax' as it shows the probability of each output in the output layer.
3. We selected the 'adam' optimizer for implementing stochastic gradient descent method. Because of this, if we train the same model on the same dataset multiple times, the accuracy might differ since the method is stochastic and training inputs are coming in a random order.
4. We selected the 'categorical_crossentropy' loss function. This implements the categorical cross entropy loss function which is used for multi-label classifications.
5. We selected the training metric to be 'accuracy'
6. We trained the model for 150 epochs

To try to improve the **Accuracy** : We created another model with the same parameters but changed the number of neurons in both the hidden layers. We replaced 12 neurons with 24 neurons in hidden layers. This resulted in a **significant growth** in the **rate of increase** in **accuracy per epoch** when the model was being trained. Hence, we only had to train our model over around 50 epochs for around 99% accuracy on the training dataset.

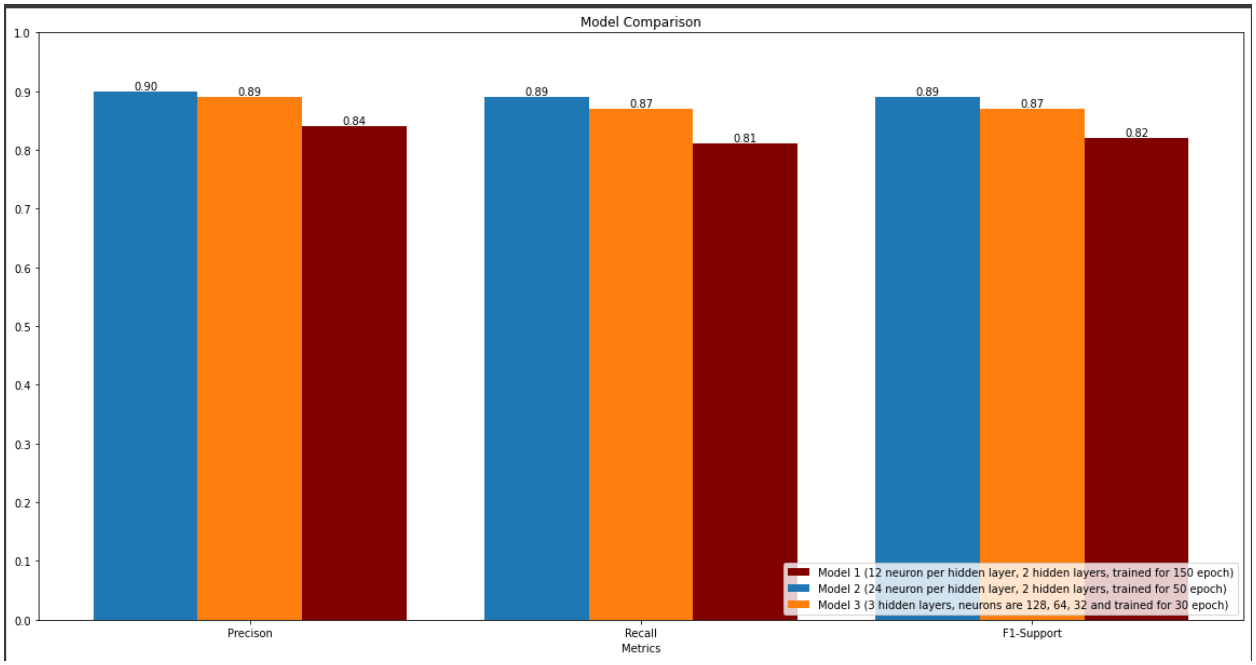
More neurons can learn more features about an image and hence, are learning to predict images faster than when we have 12 neurons in hidden layers.

So, we tried to improve the accuracy more by adding another hidden layer as well as increasing the number of neurons in our layers. In the final model, we had 3 hidden layers with the number of neurons in layers being 128, 64, 32 respectively. Since we had a huge number of neurons, we just trained the model for 30 epochs.

After this, we have simply calculated the required metrics for both the models and plotted the comparison charts.

Comparison Charts:

1. **Metrics Comparison over test images**



Google Collab with Complete Code with Step by Step Explanations : **LINK**

As a whole, the order of accuracy of the classification models in given in the 3 parts is :

Neural Network based Model (~90%) > KNN Model (~80%) > Learning with Prototypes Model (~47%)