NCC **VS** KNN CLASSIFY ALGORITHMS USING CIFAR-10

Nikolaos Karapatsias 10661 (Department of Electrical and Computing Engineering AUTH)

This document provides an implementation of the algorithms k-Nearest Neighbors and Nearest Centroid Classifier (known as KNN and NCC respectively) using as framework the library of sklearn. This model is trained and tested with the dataset of Cifar-10 which is consisted of 50.000 training samples and 10.000 testing samples all cassified in 10 groups. More details can be found on cifar-10 website.

I. INTRODUCTION

Lets start providing a brief introduction on how each of CNN and KNN algorithms work.

The Nearest Centroid Classifier (NCC) works by representing each class of data points with a single centroid, which is the mean (or centroid) of all points in that class. To classify a new data point, the algorithm calculates its distance to each class centroid and assigns it to the class with the closest centroid. This simple approach is computationally efficient and effective when data points within each class are grouped around a central point, making it well-suited for linearly separable datasets. However, NCC may struggle with complex or non-linearly separable data distributions.

The k-Nearest Neighbors (KNN) algorithm classifies a new data point based on the classes of its *k* closest neighbors in the feature space. It calculates the distance between the new point and all other points in the training set, selecting the *k* nearest ones (usually based on Euclidean distance). The new point is then assigned to the class most common among these neighbors. This makes KNN a simple, instance-based algorithm that works well with well-separated data but can be computationally intensive and sensitive to irrelevant features and noise.

II. CODE

A. Setting up

First of all, in order to make it easier for the reader to run the code, we loaded the database of Cifar10 throughout a well-known framework called Tensorflow. In that way, our program installs completely independently, without any user action, using the commands "
```
import tensorflow as tf
```
and
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
```
In that way, we store consecutively in numpy array type our images x and their class y of both train and test sets. We are also using
```
os.environ["LOKY_MAX_CPU_COUNT"] = "6"
```
command , included in OS library, for multitasking purposes in order for our code to run faster. In that way we split the project in 6 cores and running the code simultaneously. User is strongly adviced to change number "6" to their number of CPU cores. Last but not least, we are loading up in our project the parts of sklearn library we are gonna use (completely optional, its is done only for convenience)

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neighbors import NearestCentroid
from sklearn.metrics import accuracy_score
```

B. Flatten the dataset

After loading our dataset and setting up our environment, we are trasforming x_train, y_train and their respected test arrays into 1d and 2d arrays in order to fit in the algorithms included in sklearn to work. To be more specific, x_train and x_test received from tensorflow are 4d arrays where the first diamension is the image number with diamension 50.000 and 10.000 respectively, the next 2 are the pixel locations and ,baring in mind that we got 32x32 pixel images, then these two diamensions will be 32. Lastly the fourth dimension specifies the intensity of either red blue or green color so it has diamension 3. Overally these two arrays got a final dimension of 50.000x32x32x3. In order for sklearn.neighbors and sklearl.NearestCentroid to work, we need to feed them with 2d arrays. Thats why we flatten each image in these two arrays with the command:

```
x_train_flat = x_train.reshape(x_train.shape[0], -1)
x_test_flat = x_test.reshape(x_test.shape[0], -1)
```

in that way transform the diamension of each array in 50.000x3.072 (32x32x3 = 3.072) storing all the pixels and intensity information in a line array. In the same way we transform both y_train and y_test and the final diamensions of all the flatten arrays are shown below:

```
X_train_flat diamension: (50000, 3072)
Y_train_flat diamension: (50000,)
X_test_flat diamension: (50000, 3072)
Y_test_flat diamension: (10000,)
```

```
knn(1) #running knn for k = 1
knn(3) #running knn for k = 3
ncc()
```

C. Creating NCC and KNN reusable functions

1) k-Nearest Neighbors:

```
def knn(k):
    #setting up a knn model with given k
    knn = KNeighborsClassifier(n_neighbors=k)

    # Training our model
    knn.fit(x_train_flat, y_train_flat)

    # Make predictions on the test dataset
    y_pred = knn.predict(x_test_flat)

    # Calculating the accuracy of knn
    accuracy = accuracy_score(y_test_flat, y_pred)
    print(f"Accuracy of {k}-NN:", accuracy*100, "%")
```

As show in the picture above, we are setting up our knn model using sklearn.neighbors and then using the command knn.fit in order to train our model. Once the model is trained, we are ready to make our predictions on test set using knn.predict command. Finally, since we know the classes (y_test) of each data in test set (x_test) and by using sklearn.metrics we are able to calculate the accuracy of this algorithm in %.

2) Nearest Centroid Classifier:

```
def ncc():
    #setting up a ncc model
    clf = NearestCentroid()

    # Training our model
    clf.fit(x_train_flat, y_train_flat)
    # Make predictions on the test dataset
    y_pred = clf.predict(x_test_flat)

    # Calculating the accuracy of knn
    accuracy = accuracy_score(y_test, y_pred)
    print("Accuracy of NCC:", accuracy*100, "%")
```

In the same way implementing our NCC model with the help of sklearl.NearestCentroid and then training our model, make predictions in the given test set and last but not least calculating the accuracy of our algorithm in %.

D. Running our program

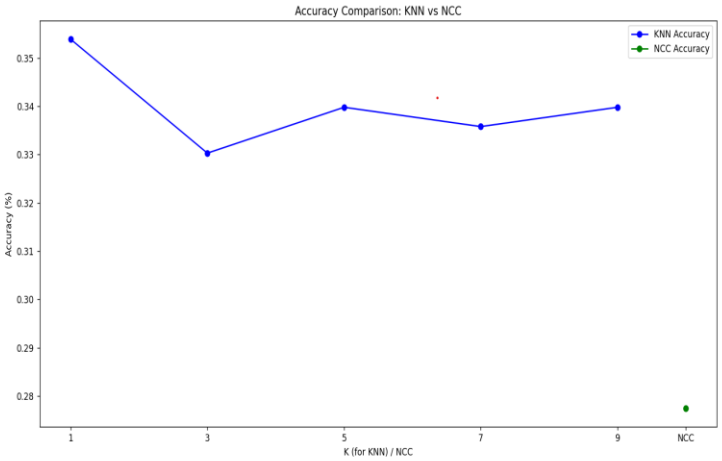
After all of this, we are ready to run our code by calling the functions we defined on step C. as given below:

And getting the results of the accuracy of each algorithm

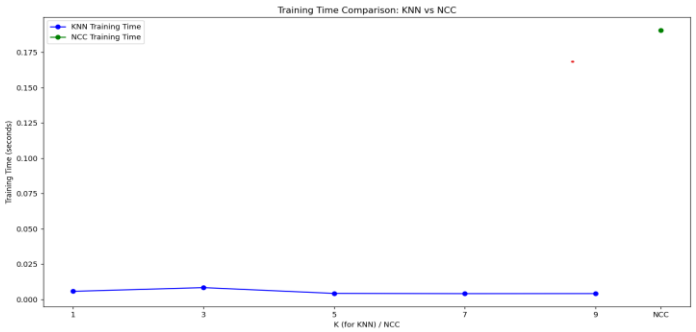
```
Accuracy of 1-NN: 35.39 %
Accuracy of 3-NN: 33.03 %
Accuracy of NCC: 27.74 %
```

III. GRAPHS

A. Accuracy Comparison



B. Training time Comparison



C. Terminal Output

```
Accuracy of 1-NN: 35.39 %
Training time for 1-NN: 0.005760100000003376 seconds
Accuracy of 3-NN: 33.03 %
Training time for 3-NN: 0.008388400000001184 seconds
Accuracy of 5-NN: 33.98 %
Training time for 5-NN: 0.0042938999999933 seconds
Accuracy of 7-NN: 33.58 %
Training time for 7-NN: 0.004131699999987859 seconds
Accuracy of 9-NN: 33.98 %
Training time for 9-NN: 0.004165699999987282 seconds
Accuracy of NCC: 27.74 %
Training time for NCC: 0.19054139999997233 seconds
```

Note: For simplicity reasons I didn't provide any of the Graph code in the main project. All of the graphs were implemented by using **Matplotlib** library. The first graph is just a presentation of the accuracy of each algorithm and the Training time comparison graph was implemented by using the **Time** library as shown below:

```
start_time = time.perf_counter()
clf.fit(x_train_flat, y_train_flat)
training_time = time.perf_counter() - start_time
```

```
start_time = time.perf_counter()
knn.fit(x_train_flat, y_train_flat)
training_time = time.perf_counter() - start_time
```

IV. CONCLUSION

As we can observe from the graphs and the output, KNN significantly outperforms NCC in classifying the CIFAR-10 dataset. KNN achieves much faster training times—approximately 50 times faster than NCC—while also providing a higher classification accuracy (around 6.5% higher). For example, 1-NN achieves an accuracy of 35.39% with a training time of 0.00576 seconds, whereas NCC only reaches 27.74% accuracy and takes 0.1905 seconds to train. This demonstrates KNN's efficiency and effectiveness over NCC for this dataset.