NAME : Punam Das

ID : 21-44946-2

Section : B

Image Classification using CIFAR-10 Dataset: Applying K-Nearest Neighbors Classifier using Euclidean (L2) Distance with 5-fold Cross-Validation

**Model Building and Evaluation:**

import numpy as np

import matplotlib.pyplot as plt

import data\_utils

import download

def download\_data():

    url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"

    download\_dir = "./data"

    download.maybe\_download\_and\_extract(url,download\_dir)

# Class to initialize and apply K-nearest neighbour classifier

class KNearestNeighbor(object):

    def \_\_init\_\_(self):

        pass

    # Method to initialize classifier with training data

    def train(self, X, y):

        self.X\_train = X

        self.y\_train = y

    def predict(self, X, k=1, num\_loops=0, distance='euclidean'):  # Added distance parameter

        if num\_loops == 0:

            dists = self.compute\_distances(X, distance)  # Pass distance parameter

        else:

            raise ValueError('Invalid value %d for num\_loops' % num\_loops)

        return self.predict\_labels(dists, k=k)

def compute\_distances(self, X, distance='euclidean'):  # Added distance parameter

        num\_test = X.shape[0]  # Number of test samples

        num\_train = self.X\_train.shape[0]  # Number of training samples

        dists = np.zeros((num\_test, num\_train))  # Initialize distance matrix with zeros

        for i in range(num\_test):  # Iterate over each test sample

            if distance == 'euclidean':  # If distance metric is Euclidean

                dists[i, :] = np.sqrt(np.sum(np.square(self.X\_train - X[i, :]), axis=1))  # Calculate Euclidean distance

            elif distance == 'manhattan':  # If distance metric is Manhattan

                dists[i, :] = np.sum(np.abs(self.X\_train - X[i, :]), axis=1)  # Calculate Manhattan distance

            else:  # If distance metric is not supported

                raise ValueError('Invalid distance metric')  # Raise an error

        return dists  # Return computed distances

def predict\_labels(self, dists, k=1):  # Predict labels using nearest neighbors

        num\_test = dists.shape[0]  # Number of test samples

        y\_pred = np.zeros(num\_test)  # Initialize predicted labels

        for i in range(num\_test):  # Iterate over each test sample

            closest\_y = self.y\_train[np.argsort(dists[i, :])[:k]]  # Find k nearest neighbors

            y\_pred[i] = np.argmax(np.bincount(closest\_y))  # Predict the label with most occurrences

        return y\_pred  # Return predicted

def visualize\_data(X\_train, y\_train):

    classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

    num\_classes = len(classes)

    samples\_per\_class = 7

    for y, cls in enumerate(classes):

        idxs = np.flatnonzero(y\_train == y)

        idxs = np.random.choice(idxs, samples\_per\_class, replace=False)

        for i, idx in enumerate(idxs):

            plt\_idx = i \* num\_classes + y + 1

            plt.subplot(samples\_per\_class, num\_classes, plt\_idx)

            plt.imshow(X\_train[idx].astype('uint8'))

            plt.axis('off')

            if i == 0:

                plt.title(cls)

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    # Download CIFAR10 data and store it in the current directory if you have not done it.

    download\_data()

    cifar10\_dir = './data/cifar-10-batches-py'

    # Load training and testing data from CIFAR10 dataset

    X\_train, y\_train, X\_test, y\_test = data\_utils.load\_CIFAR10(cifar10\_dir)

    # Checking the size of the training and testing data

    print('Training data shape: ', X\_train.shape)

    print('Training labels shape: ', y\_train.shape)

    print('Test data shape: ', X\_test.shape)

    print('Test labels shape: ', y\_test.shape)

    # Memory error prevention by subsampling data. We sample 10000 training examples and 1000 test examples.

    num\_training = 7000

    mask = list(range(num\_training))

    X\_train = X\_train[mask]

    y\_train = y\_train[mask]

    num\_test = 700

    mask = list(range(num\_test))

    X\_test = X\_test[mask]

    y\_test = y\_test[mask]

    # Flatten the training and test data so each row consists of all pixels of an example

    X\_train = np.reshape(X\_train, (X\_train.shape[0], -1))  # Flatten training data

    X\_test = np.reshape(X\_test, (X\_test.shape[0], -1))  # Flatten test data

    print(X\_train.shape, X\_test.shape)  # X\_train should be (10000, 3072) and X\_test should be (1000, 3072)

    # Performing KNN

    classifier = KNearestNeighbor()

    classifier.train(X\_train, y\_train)  # Train classifier

    # Use Euclidean distance

    y\_test\_pred\_euclidean = classifier.predict(X\_test, k=5, distance='euclidean')

    num\_correct = np.sum(y\_test\_pred\_euclidean == y\_test)

    accuracy\_euclidean = float(num\_correct) / num\_test

    print('Got %d / %d correct with k=5 =>accuracy\_euclidean: %f'% (num\_correct, num\_test,accuracy\_euclidean\*100))

    # Use Manhattan distance

    y\_test\_pred\_manhattan = classifier.predict(X\_test, k=5, distance='manhattan')

    num\_correct = np.sum(y\_test\_pred\_manhattan == y\_test)

    accuracy\_manhattan = float(num\_correct) / num\_test

    print('Got %d / %d correct with k=5 =>accuracy\_manhattan: %f'% (num\_correct, num\_test,accuracy\_manhattan\*100))

    # Perform 5-fold cross-validation to find the optimal k from choices below

    num\_folds = 5

    k\_choices = [1, 3, 5, 8, 10]

    X\_train\_folds = np.array\_split(X\_train, num\_folds)

    y\_train\_folds = np.array\_split(y\_train, num\_folds)

    k\_to\_accuracies = {}  # dictionary to hold validation accuracies for each k

    for k in k\_choices:

        k\_to\_accuracies[k] = []  # each key, k, should hold its list of 5 validation accuracies

        # For each fold of cross validation

        for i in range(num\_folds):

        # Split training data into validation fold and training folds

            X\_val\_fold = X\_train\_folds[i]  # Validation data

            y\_val\_fold = y\_train\_folds[i]  # Validation labels

            X\_train\_fold = np.concatenate(X\_train\_folds[:i] + X\_train\_folds[i+1:])  # Training data

            y\_train\_fold = np.concatenate(y\_train\_folds[:i] + y\_train\_folds[i+1:])  # Training labels

            # Initialize classifier with training folds and compute distances between examples in validation fold and training folds

            classifier.train(X\_train\_fold, y\_train\_fold)  # Train classifier

            dists\_fold = classifier.compute\_distances(X\_val\_fold)  # Compute distances

            # Use classifier to predict labels of validation fold for the given k value

            y\_val\_pred = classifier.predict\_labels(dists\_fold, k=k)  # Predict labels for validation data

            # Number of test examples correctly predicted, where y\_val\_pred contains labels predicted by classifier on validation fold

            num\_correct = np.sum(y\_val\_pred == y\_val\_fold)

            accuracy = float(num\_correct) / y\_val\_fold.shape[0]

            k\_to\_accuracies[k].append(accuracy)

    print("Printing our 5-fold accuracies for varying values of k:")

    print()

    for k in sorted(k\_to\_accuracies):

        for accuracy in k\_to\_accuracies[k]:

            print('k = %d, accuracy = %f' % (k, accuracy))

    for k in sorted(k\_to\_accuracies):

        print('k = %d, avg. accuracy = %f' % (k, sum(k\_to\_accuracies[k])/5))

    for k in k\_choices:

        accuracies = k\_to\_accuracies[k]

        plt.scatter([k] \* len(accuracies), accuracies)

    # Plot the trend line with error bars that correspond to standard deviation

    accuracies\_mean = np.array([np.mean(v) for k,v in sorted(k\_to\_accuracies.items())])

    accuracies\_std = np.array([np.std(v) for k,v in sorted(k\_to\_accuracies.items())])

    plt.errorbar(k\_choices, accuracies\_mean, yerr=accuracies\_std)

    plt.title('Cross-validation on k')

    plt.xlabel('k')

    plt.ylabel('Cross-validation accuracy')

    plt.savefig('cross-validation\_accuracy.jpg')

    # Choose the best value of k based on cross-validation results

    best\_k = k\_choices[np.argmax(accuracies\_mean)]  # Choose k with highest mean accuracy

    # Intialize classifier and predict labels of test data, X\_test, using the best value of k

    classifier = KNearestNeighbor()  # Initialize classifier

    classifier.train(X\_train, y\_train)  # Train classifier

    y\_test\_pred = classifier.predict(X\_test, k=best\_k)  # Predict labels for test data

    # Computing and displaying the accuracy for the best k found during cross-validation

    num\_correct = np.sum(y\_test\_pred == y\_test)

    accuracy = float(num\_correct) / num\_test

    print('Got %d / %d correct on test data => accuracy: %f' % (num\_correct, num\_test, accuracy\*100))

    # Accuracy above should be ~ 57-58%

**OUTPUT:-**

**Training data shape: (50000, 32, 32, 3)**

**Training labels shape: (50000,)**

**Test data shape: (10000, 32, 32, 3)**

**Test labels shape: (10000,)**

**(7000, 3072) (700, 3072)**

**Got 208 / 700 correct with k=5 =>accuracy\_euclidean: 29.714286**

**Got 218 / 700 correct with k=5 =>accuracy\_manhattan: 31.142857**

**Printing our 5-fold accuracies for varying values of k:**

**k = 1, accuracy = 0.272857**

**k = 1, accuracy = 0.274286**

**k = 1, accuracy = 0.257857**

**k = 1, accuracy = 0.282143**

**k = 1, accuracy = 0.267143**

**k = 3, accuracy = 0.264286**

**k = 3, accuracy = 0.264286**

**k = 3, accuracy = 0.262143**

**k = 3, accuracy = 0.265714**

**k = 3, accuracy = 0.253571**

**k = 5, accuracy = 0.274286**

**k = 5, accuracy = 0.282143**

**k = 5, accuracy = 0.276429**

**k = 5, accuracy = 0.270714**

**k = 5, accuracy = 0.272143**

**k = 8, accuracy = 0.275714**

**k = 8, accuracy = 0.288571**

**k = 8, accuracy = 0.288571**

**k = 8, accuracy = 0.281429**

**k = 8, accuracy = 0.269286**

**k = 10, accuracy = 0.272143**

**k = 10, accuracy = 0.279286**

**k = 10, accuracy = 0.289286**

**k = 10, accuracy = 0.283571**

**k = 10, accuracy = 0.282857**

**k = 1, avg. accuracy = 0.270857**

**k = 3, avg. accuracy = 0.262000**

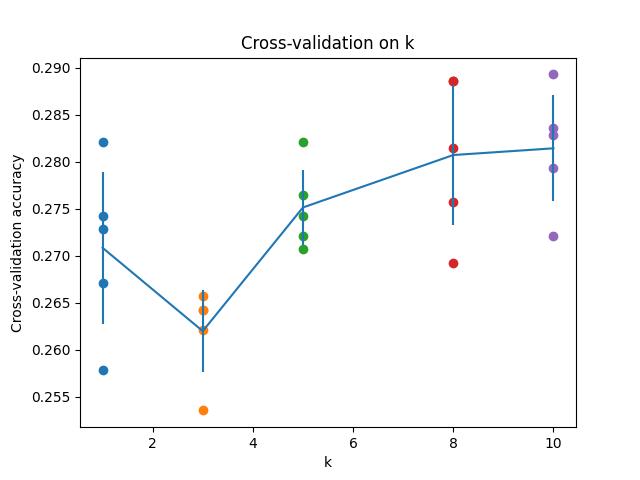
**k = 5, avg. accuracy = 0.275143**

**k = 8, avg. accuracy = 0.280714**

**k = 10, avg. accuracy = 0.281429**

**Got 198 / 700 correct on test data => accuracy: 28.285714**

**Cross-validation\_accuracy graph:-**

****

**Results and Analysis:**

**Model Performance Comparison using Manhattan and Euclidean Distance:**

* Using Manhattan distance: Accuracy = 31.14%
* Using Euclidean distance: Accuracy = 29.71%

From these results, it appears that the model performs slightly better using Manhattan distance compared to Euclidean distance.

**Cross Validation Results:**

* Cross-validation was performed using 5-fold validation.
* Average accuracies were computed for different values of k.
* The average accuracies for various values of k are as follows:
* k = 1: 27.09%
* k = 3: 26.20%
* k = 5: 27.51%
* k = 8: 28.07%
* k = 10: 28.14%

These results suggest that k = 10 yields the highest average accuracy among the tested values of k. The average accuracies range from around 27.09% to 28.14% across different values of k.

On the test data, the model achieves an accuracy of 28.28%.

1. **Discussion**:
   * **Cross-Validation for Finding Optimal k**: Cross-validation is performed to ensure that the model's performance evaluation is not dependent on the particular random choice of train/test split. It helps to assess how well the model generalizes to unseen data and provides a more reliable estimate of the model's performance. By testing the model with different values of k and averaging the results, we can identify the k value that yields the best performance on average across different data splits.
   * **Impact of Optimal k on Model Fitting**: The optimal value of k is crucial as it directly affects the bias-variance trade-off. A smaller value of k (like 1 or 3) tends to overfit the data, leading to high variance and poor generalization to unseen data. On the other hand, a larger value of k (like 50 or 100) may result in underfitting, leading to high bias and again poor generalization. The optimal value of k balances this trade-off by providing the best compromise between bias and variance, resulting in a well-fitted model.

**Comparison with Neural Networks (NN) or Convolutional Neural Networks (CNN)**:

* + While k-NN is a simple and interpretable model, it might not be the best choice for complex datasets like CIFAR-10 due to its high computational cost during inference, especially with large training datasets.
  + More sophisticated models like NNs or CNNs often outperform k-NN on image classification tasks like CIFAR-10. These models can learn hierarchical features from data automatically and generally achieve higher accuracies.
  + CNNs, in particular, are designed to handle spatial information in images efficiently, making them well-suited for tasks like image classification.
  + However, k-NN can still serve as a useful baseline or complementary method for comparison with more complex

In conclusion, cross validation helps in determining the optimal value of k in the k-Nearest Neighbors algorithm, leading to better fitting of the model. While k-Nearest Neighbors is a simple and interpretable algorithm, its performance might be surpassed by more sophisticated models like Neural Networks or Convolutional Neural Networks on complex datasets like CIFAR-10.