k-Nearest Neighbor (kNN) exercise – Report

First, we add the necessary libraries. And then we code to make "matplotlib" figures appear inline in the notebook rather than in a new window. Load the raw CIFAR-10 data. And then, cleaning up variables to prevent loading data multiple times (which may cause memory issue) As a sanity check, we print out the size of the training and test data. We see 50000 training data(32x32x3) and label and 10000 test data(32x32x3) and label.

We show a few examples of training images from each class. To visualizing part and see your images and classes, we show a few examples(7) of training images from each class. To do this, we move through all classes and randomly select 7 samples.

Then we subsample the data for more efficient code execution. So to do this, we mask 5000 training data and 500 testing data. Then we turn the 32x32x3 data into a one-dimensional array and as a result we create 5000x3073 trainig data and 500x3073 test data.

Then we Implement compute_distances_two_loops function in cs231n/classifiers/k_nearest_neighbor.py that computes Euclidean distance (L2 distance/norm) between test and training samples.

We add “compute_distances_two_loops” function,

```
 for i in range(num_test):
          for j in range(num_train):
 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

              dists[i][j] = np.sqrt(np.sum((X[i] - self.X_train[j])**2))


  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Because L2 norm distance is d2(I1,I2) = sqrt(sum((I1(p) - I2(p))^2)). And then we call this function from notebook. As a result, we get a distance matrix of 500x5000. In each dists [i] [j], we keep the L2 distance between the ith test data and the jth training data.

Answer inline question 1:

What in the data is the cause behind the distinctly bright rows?
Answer: Distinctly bright rows show that high distances of between with "ith test data" and "jth training data". So l2 norm distance high in there.

What causes the columns?
Answer: The same occurs with the columns but this time we are comparing(compute distance of between with "jth traning data and all test data) a training data with all test data.

And to predict, we need to implement the function predict_labels.

```
def predict_labels(self, dists, k=1):
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in range(num_test):
            closest_y = []
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            closest_y = self.y_train[np.argsort(dists[i])][0:k]

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            y_pred[i] = np.bincount(closest_y).argmax()

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        return y_pred
```

# bincount() = count number of occurrences of each value in array of non-negative ints., argmax() = choose max indicate

To find "closest_y", we list the dists (distance) matrix, which we previously created with test data, from small to large. Then, for k = 1, we assign the closest value in the nearest 1 neighborhood to "closest_y". To do this, we take advantage of np.argsort funtion. This function sorts array small to large.

Then we find the most duplicate elements in the array and we put this in the output label matrix(y_pred). We use the np.bincount function to count the duplicate values and the argmax function to find the largest index.

İnline Question 2:

Answer: 1 - 3 - 5

Explanation:
1-

$$\text{L1} = ||\tilde{p}_{ij}^{(1)} - \tilde{p}_{ij}^{(2)}|| = ||p_{ij}^{(1)} - \mu - p_{ij}^{(2)} + \mu|| = ||p_{ij}^{(1)} - p_{ij}^{(2)}||$$

Subtracting the mean does not change the performance of the NN classifier. You shift every point in the same direction by the same amount, so the distances between the points remain the same.
3-

$$\text{3- } ||\tilde{p}_{ij}^{(1)} - \tilde{p}_{ij}^{(2)}|| = ||p_{ij}^{(1)} - \mu - p_{ij}^{(2)} + \mu|| = ||p_{ij}^{(1)} - p_{ij}^{(2)}|| \text{ and } ||p_{ij}^{(1)}/\sigma - p_{ij}^{(2)}/\sigma|| = ||(1/\sigma) * (p_{ij}^{(1)} - p_{ij}^{(2)})|| = (1/\sigma)|| (p_{ij}^{(1)} - p_{ij}^{(2)})||$$

Subtracting the mean and dividing by the standard deviation does not change the performance for the same reason as above. The points are first shifted in the same direction by the same amount and the distances between the points remain the same. Then the distances between the points are all scales by the same amount, and so is the distance between them, meaning the biggest distances remain the biggest distances also after scaling.

5- Rotating the axes also doesn't change the distance between the points, as neither of the points move relatively to each other.

Then we Implement compute_distances_one_loops and compute_distances_no_loops functions.

```
def compute_distances_one_loop(self, X):

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):

        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        dists[i,:] = np.sqrt(np.sum((X[i] - self.X_train) ** 2, axis=1))

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return dists
```

Calculate L2 norm distance 2(I1,I2) = sqrt(sum((I1(p) - I2(p))^2)) In here, we compute the l2 distance between the "ith test point" and "all training points", and store the result in dists[i,:].

```
def compute_distances_no_loops(self, X):

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    test2 = np.sum(X**2, axis = 1).reshape(num_test, 1)
    train2 = np.sum(self.X_train**2, axis = 1).reshape(1, num_train)
    dists = np.sqrt(test2 + train2 - 2 * np.dot(X, self.X_train.T))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return dists
```

Calculate L2 norm distance 2(I1,I2) = sqrt(sum((I1(p) - I2(p))^2)) In here, we compute the l2 distance between all test points and all training points without using any explicit loops, and store the result in dists. To do this, we calculate the l2 distance using matrix multiplication and two broadcast sums. [(x-y)^2 = x^2 + y^2 - 2*XY] Shape of dists is num_test x num_train.

All three implementations by their execution time is as follows.

```
Two loop version took 45.656703 seconds
One loop version took 51.825686 seconds
No loop version took 0.588670 seconds
```

Here we see that the shortest “No loop version” takes, then the “Two loop version” and the longest “One loop version” takes.

Then we perform cross-validation with the training set split in 5 folds .

```
X_train_folds = np.array_split(X_train, num_folds)#split 5 folds
y_train_folds = np.array_split(y_train, num_folds)#split 5 folds
```

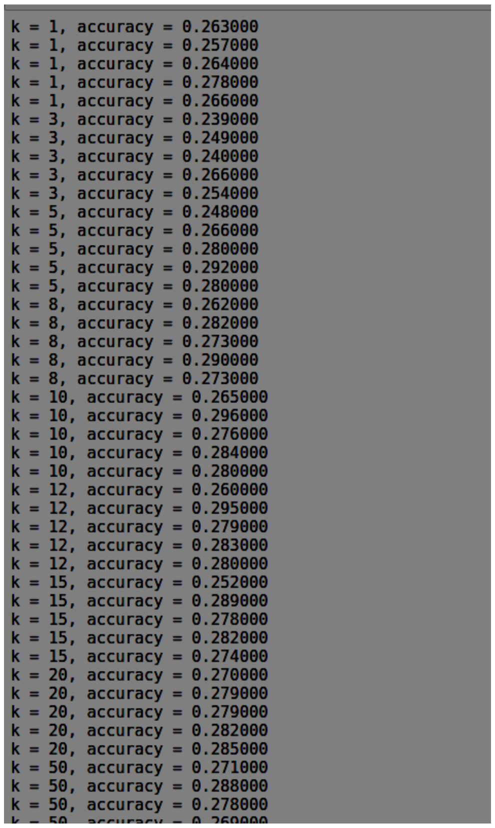
And

```
for k in k_choices:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        # prepare training data for the current fold
        X_train_fold = np.concatenate([ fold for j, fold in enumerate(X_train_folds) if i != j ])
        y_train_fold = np.concatenate([ fold for j, fold in enumerate(y_train_folds) if i != j ])

        # use of k-nearest-neighbor algorithm
        classifier.train(X_train_fold, y_train_fold)
        y_pred_fold = classifier.predict(X_train_folds[i], k = k, num_loops=0)

        # Compute the fraction of correctly predicted examples
        num_correct = np.sum(y_pred_fold == y_train_folds[i])
        accuracy = float(num_correct) / X_train_folds[i].shape[0]
        k_to_accuracies[k].append(accuracy)
```

And we print out the computed accuracies.



```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
```

Then we find “best k” with “best_k = k_choices[accuracies_mean.argmax()]”

In my example, "best k" is 10 and its accuracy 0.296. And then we evaluate the number of correct predictions for the test set by using the best k. The result is “Got 141 / 500 correct => accuracy: 0.282000”

İnline Question 3:

Answer: 4

Explanation:

1- Because the distance function used to find the k nearest neighbors is not linear, so it usually won't lead to a linear decision boundary.(false)

2- The training error of a 1-NN will always be lower than that of 5-NN because for each training example, its nearest neighbor is always going to be itself(false)

3- The test error of a 1-NN will not always be lower than 5-NN. It will depend on the data we are working with, thus cross validation is a way to determine the best k for the algorithm.(false)

4- Increasing the size of the training dataset incurs large number of comparisons between a test example and all the training dataset, so the time that to classify a test example grows. (true)