What was the main problem/task/tool implemented? Give brief theoretical information about it (using visuals and formulas/equations is encouraged).

The main problem was the k-nearest neighbors (KNN) algorithm. KNN is a supervised machine learning technique that can be used to classify data. It predicts the label of the new data point by comparing the labels of the k nearest neighbors with the k training data points that are most similar to it.

How did you fulfill the given steps? For the main functions/classes, explain your implementation in general (focus on elementary components).

The following steps are the implementation of the KNN algorithm in Python:

1. **Data Preprocessing and Visualization**

Using `eecs598.data.cifar10()` function the CIFAR-10 dataset was loaded and divided into training and testing. The data set consists of 60000 color images with dimensions of 32x32 for 10 classes.

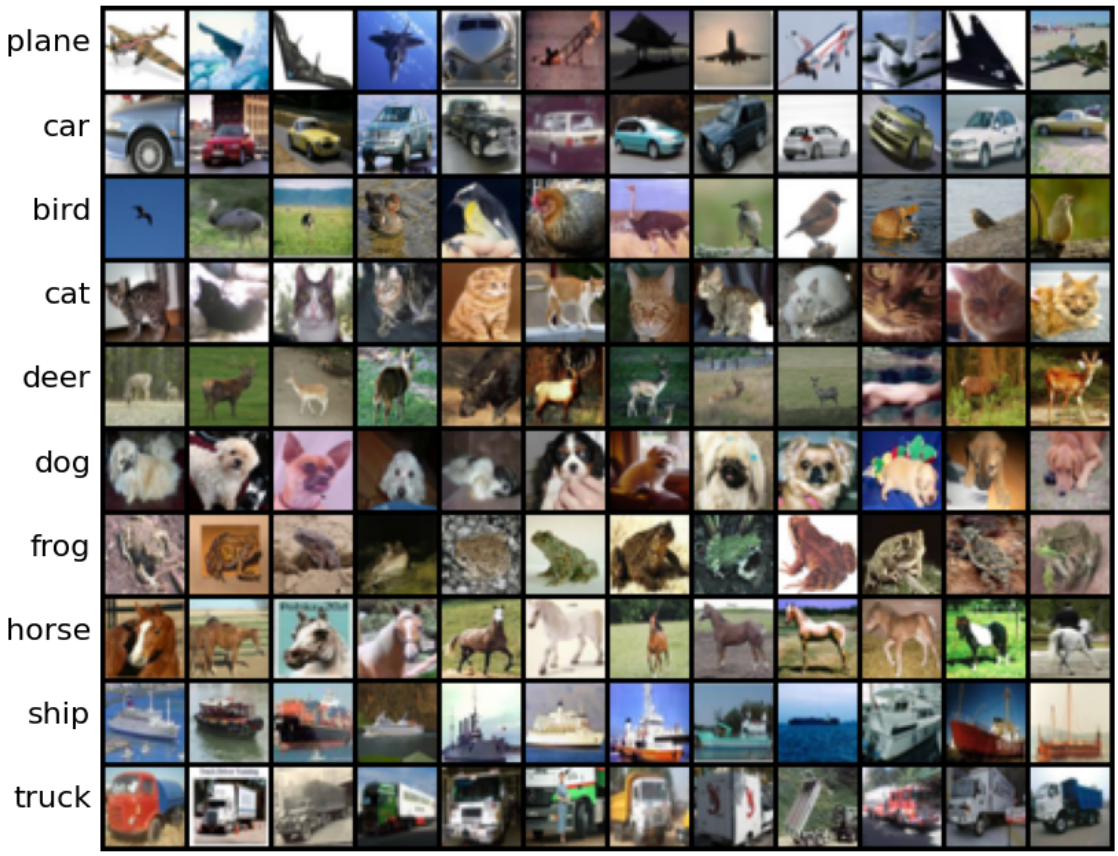


Figure 1: CIFAR-10 Dataset

2. Calculating the Euclidean Distances

Euclidean Distance Formula:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$$

To compute the Euclidean distances, three functions were implemented: naive version and two vectorized versions. Vectorization is used to calculate the distances between all training examples and all test examples in a single operation. Also it is a powerful technique that can be used to improve the performance. It is especially important for large datasets, where naive implementations can be very slow.

a. compute_distances_two_loops

```
for i in range(num_train):
  for j in range(num_test):
    dists[i,j] = ((x_train[i] - x_test[j])**2).sum()**(1/2)
```

This function calculates the distance between the test data and training data using two nested loops. The first loop iterates over the training data points, and the second loop iterates over the test data points.

b. compute_distances_one_loop

```
x_train_flatten = x_train.reshape(x_train.shape[0],-1)
x_test_flatten = x_test.reshape(x_test.shape[0],-1)

for i in range(num_train):
  dists[i,:] = torch.sum((x_train_flatten[i] - x_test_flatten)**2 , 1)**(1/2)
```

This function uses a single loop to calculate the distance between each training example and each test example. First training and test sets were reshaped into 2D and calculated the distances. Before reshaping, the size of set was [100, 3, 16, 16] after reshaping, the size of set was [100, 768].

c. compute_distances_no_loops

```
#Flatten the data sets
x = x_train.reshape(num_train,-1)
y = x_test.reshape(num_test,-1)

#Calculate the distance
x_square = (x**2).sum(dim = 1).reshape(-1,1)
y_square = (y**2).sum(dim = 1).reshape(1,-1)
xy = x.mm(y.T)
dists = (x_square + y_square - 2*xy)**(1/2)
```

This function computes the squared Euclidean distance between each element of the training set and each element of the test set. It does this without using any loops or creating any large intermediate tensors.

The function works by first flattening the training and test images into vectors. Then, it computes the squared Euclidean distance between each training vector and each test vector using the following formula:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2} = \sqrt{x^2 + y^2 - 2xy}$$

Comparison

For 500 training and 500 test data points;

```
Two loop version took 10.71 seconds
One loop version took 1.20 seconds (8.9X speedup)
No loop version took 0.05 seconds (222.6X speedup)
```

Two loops version is straightforward to implement, but it is also the slowest of the three functions because it uses two loops. Also, no loop version has slight difference but it is ignorable.

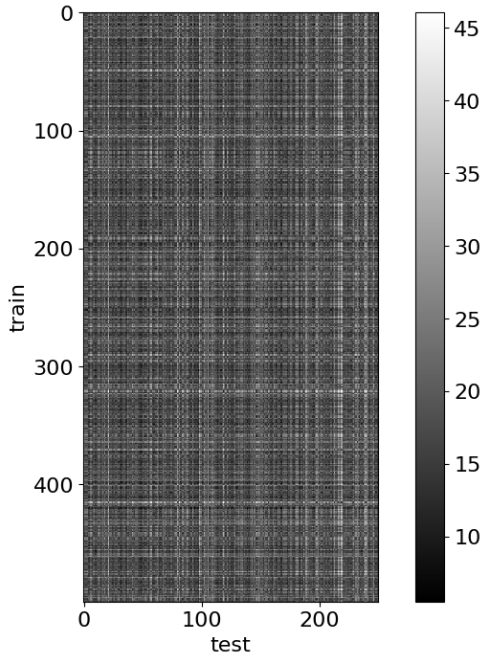


Figure 2: Visializaation of distance matrix

The visualization shows that the training set data points are generally closer to test set, the overall gray scale of image is low so that means the distance values are low. But there are straight white lines which means these data points are not close to any data points.

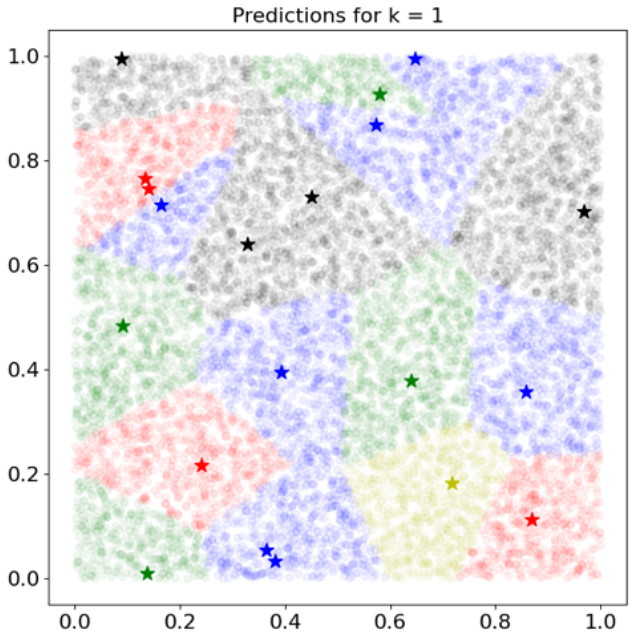
3. Predict the Labels

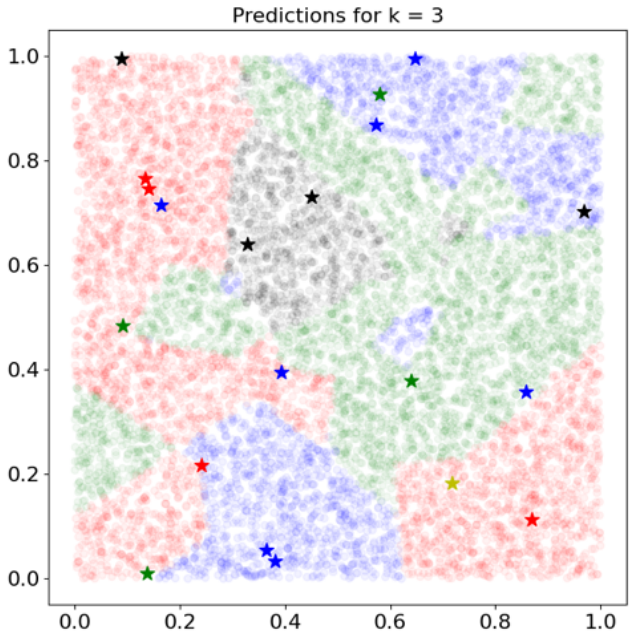
To predict labels for test samples, predict_labels function was implemented that uses those distances together with the training labels.

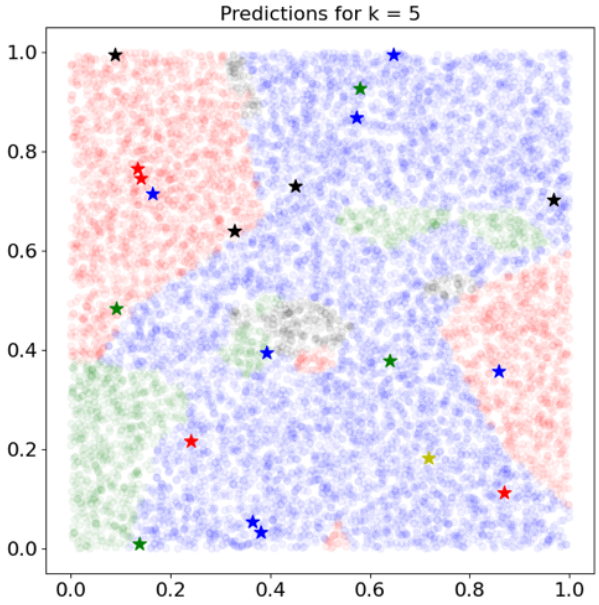
```
for i in range(num_test):
  # Compute the indices
  _, topk_indices = torch.topk(-dists[:,i], k)

  # Count the occurrences of each label
  y_pred[i] = torch.mode(y_train[topk_indices]).values.item()
```

K-Nearest Neighbors Algorithm label prediction works with a “majority vote” concept. K value is selected, then, the nearest classes are lined up. The closest k number of classes are turned into a list. Most appeared classes on that list is predicted as the label. If the appearances of the classes are same, smallest class is selected.

After implementing the predict_label, whole methods were ready to create KnnClassifier class.







In these pictures, KNN classifications with different k values are shown. The first picture uses k value as 1. Predictions search for the nearest training point and predicts itself as the same class with that training point.

In the second and third pictures, point are predicted by looking at the 3 and 5 nearest training points and its classes to predict the point's class.

Training values are assigned to the the knn classifier.

```
self.x_train = x_train
self.y_train = y_train
```

New distances are calculated, and the new classes are predicted.

```
dists = compute_distances_no_loops(self.x_train, x_test)
y_test_pred =  predict_labels(dists, self.y_train, k)
```

After predictions, the accuracies of KnnClassifier with k=1 and k=5 are calculated on CIFAR-10. The accuracies are calculated as 27.4 % and 27.8 % respectively.

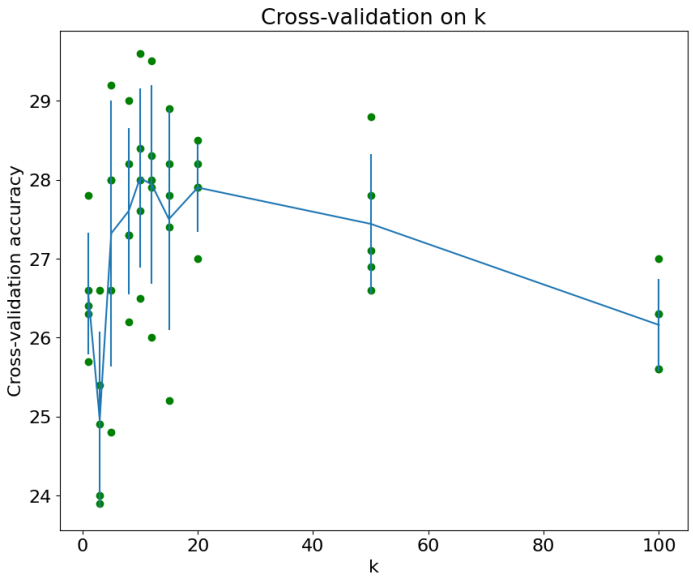
4. Cross-Validation

The knn_cross_validate() function performs cross-validation for a KNN classifier. It is a useful function for finding the best value of k for a KNN classifier.
The function works by first splitting the training data into num_folds equally-sized folds using the torch.chunk() function. Then, for each value of k in k_choices(there were 10 different k values which were already determined), the function trains a KNN classifier on all but one fold of the data and evaluates the classifier on the remaining fold. This process is repeated num_folds times for each value of k, and the average accuracy across all folds is stored in the k_to_accuracies dictionary.

```
for k in k_choices:
  accuracy_per_fold = []
  for i in range(num_folds):
    # training data
    x_train_fold = torch.cat(x_train_folds[:i] + x_train_folds[i + 1:], dim=0)
    y_train_fold = torch.cat(y_train_folds[:i] + y_train_folds[i + 1:], dim=0)

    # validation data
    x_validation_fold = x_train_folds[i]
    y_validation_fold = y_train_folds[i]

    # train the KnnClassifier
    knn_classifier = KnnClassifier(x_train_fold, y_train_fold)

    # check the accuracy of the fold
    accuracy_fold = knn_classifier.check_accuracy(x_validation_fold,
y_validation_fold, k)
    accuracy_per_fold.append(accuracy_fold)

  # Store the accuracies for all folds and all values in k in k_to_accuracies
  k_to_accuracies[k] = accuracy_per_fold
```

The cross- validation results are shown in right. As you can see in the figure, k=10 model has the best mean accuracy.

```
k=1 [26.3, 25.7, 26.4, 27.8, 26.6]
k=3 [23.9, 24.9, 24.0, 26.6, 25.4]
k=5 [24.8, 26.6, 28.0, 29.2, 28.0]
k=8 [26.2, 28.2, 27.3, 29.0, 27.3]
k=10 [26.5, 29.6, 27.6, 28.4, 28.0]
k=12 [26.0, 29.5, 27.9, 28.3, 28.0]
k=15 [25.2, 28.9, 27.8, 28.2, 27.4]
k=20 [27.0, 27.9, 27.9, 28.2, 28.5]
k=50 [27.1, 28.8, 27.8, 26.9, 26.6]
k=100 [25.6, 27.0, 26.3, 25.6, 26.3]
```



After determining the k value, k-NN algorithm was performed on CIFAR-10 data and Got 3386 / 10000 correct; accuracy is 33.86%.

How did it perform? Were you satisfied with your results? Write down your observations and maybe criticize yourself.

An accuracy of 33.86% on the CIFAR-10 dataset is not very good, but it is not terrible either. KNN is a simple and effective algorithm for image classification, but it is not the most accurate algorithm. On the other hand, It is easy to implement and understand.

Observations and criticisms:

One observation is that the KNN algorithm is relatively simple to implement. However, it can be slow to classify new images, especially for large datasets. This is because the algorithm needs to compute the distance between the query image and all of the training images.

Another observation is that the KNN algorithm is sensitive to the value of the k parameter. A too small value of k can lead to overfitting, while a too large value of k can lead to underfitting. It is important to carefully select the value of k using hyperparameter tuning.