In [1]:
```
# # Get CIFAR10
!curl -O http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
!tar -xzvf cifar-10-python.tar.gz
!rm cifar-10-python.tar.gz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Ti
me  Current
                                 Dload  Upload   Total   Spent    Le
ft  Speed
100  162M  100  162M    0     0  9698k      0  0:00:17  0:00:17 --:-
-:-- 9946k
x cifar-10-batches-py/
x cifar-10-batches-py/data_batch_4
x cifar-10-batches-py/readme.html
x cifar-10-batches-py/test_batch
x cifar-10-batches-py/data_batch_3
x cifar-10-batches-py/batches.meta
x cifar-10-batches-py/data_batch_2
x cifar-10-batches-py/data_batch_5
x cifar-10-batches-py/data_batch_1
```

k-Nearest Neighbor (kNN) exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transferring the labels of the k most similar training examples
- The value of k is cross-validated

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

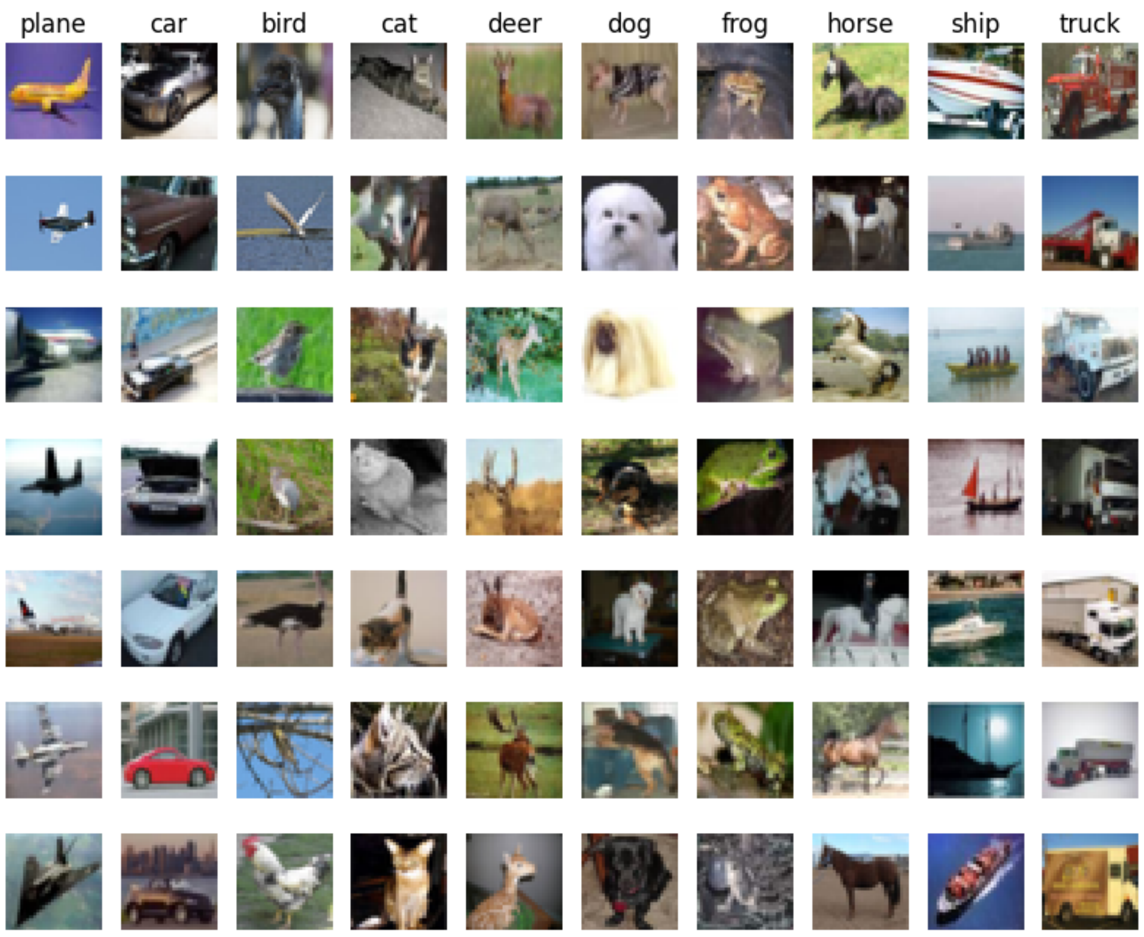
In [1]:
```
# Install required packages
# !pip3 install numpy matplotlib scipy scikit-learn imageio

# Run some setup code for this notebook.
from __future__ import print_function

import random
import numpy as np
```

```
from cs6353.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


# This is a bit of magic to make matplotlib figures appear inline i
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python
# see http://stackoverflow.com/questions/1907993/autoreload-of-modu
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs6353/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (whi
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test
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)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'h
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()
```



In [4]:
```
# Subsample the data for more efficient code execution in this exer
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]
```

In [5]:
```
# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

In [6]:
```
from cs6353.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further proc
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

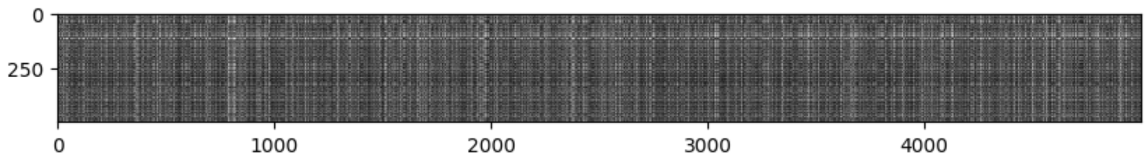
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

First, open `cs6353/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

In [7]:
```
# Open cs6353/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

In [8]:
```
# We can visualize the distance matrix: each row is a single test e
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```



Inline Question #1: Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer: The shape of the returned dists is (500, 5000), which contains the distance of each test image with all training images. The brightness shown in the visualization represents the distance values. With higher distance values, the brightness increases, and vice versa.

What in the data is the cause behind the distinctly bright rows?

It means the test image has higher distances with all training images. Hence, it is probably dissimilar to most training images, and could be an outlier.

What causes the columns?

It means the training image has high distances to most test images, indicating that this particular training image is not close to other test images.

In [9]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_tes
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

In [10]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_tes
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

Inline Question 2 We can also other distance metrics such as L1 distance. The performance of a Nearest Neighbor classifier that uses L1 distance will not change if (Select all that apply.):

1. The data is preprocessed by subtracting the mean.
2. The data is preprocessed by subtracting the mean and dividing by the standard deviation.
3. The coordinate axes for the data are rotated.
4. None of the above. (Mean and standard deviation in (1) and (2) are vectors and can be different across dimensions)

Your Answer: 1

Your explanation:

1. L1 distance: |x1 - y1| + |x2 - y2| + ...

   After subtracting mean = |(x1 - mean1) - (y1 - mean1)| + |(x2 - mean2) - (y2 - mean2)| + ... = |x1 - y1| + |x2 - y2| + ... = L1 distance.

   So, option 1 doesn't cahnge the outcome.

2. L1 distance: |x1 - y1| + |x2 - y2| + ...

   We have see the subtracting the mean, but now we need to divid by the stander devations.

   New preformance: |(x1 - mean1) - (y1 - mean1)| / sd1 + |(x2 - mean2) - (y2 - mean2)| / sd2+ ... = ||x1 - y1| / sd1 + |x2 - y2| / sd2 + ... is not equal to L1 preformance.

   So, this is not the answer.

3. Counterexample: if two points: A(1, 0) and B(0, 1), then L1 distence: |1 - 0| + |0 - 1| = 2

   After 45 degree rotation, A'(sqrt(2)/2, sqrt(2)/2) and B'(-sqrt(2)/2, sqrt(2)/2)

   New distance = |sqrt(2)/2 - (-sqrt(2)/2)| + |sqrt(2)/2 - sqrt(2)/2| = |sqrt(2)/2 + sqrt(2)/2| + |0| = |sqrt(2)/2| + 0 = sqrt(2)/2 is not equal to 2. So, the result is differ than original L1 distance.

4. The option 1 is currect. So, can't be option 4.

In [11]:
```
# Now lets speed up distance matrix computation by using partial ve
# with one loop. Implement the function compute_distances_one_loop
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make
# agrees with the naive implementation. There are many ways to deci
# two matrices are similar; one of the simplest is the Frobenius no
# you haven't seen it before, the Frobenius norm of two matrices is
# root of the squared sum of differences of all elements; in other
# the matrices into vectors and compute the Euclidean distance betw
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

In [12]:
```
# Now implement the fully vectorized version inside compute_distanc
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed be
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('Difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

In [13]:
```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) th
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loop
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops,
print('No loop version took %f seconds' % no_loop_time)

# you should see significantly faster performance with the fully ve
```

```
Two loop version took 7.882575 seconds
One loop version took 10.853509 seconds
No loop version took 0.058333 seconds
```

Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

In [16]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

x_train_folds = []
y_train_folds = []
####################################################################
# TODO:
# Split up the training data into folds. After splitting, X_train_f
# y_train_folds should each be lists of length num_folds, where
# y_train_folds[i] is the label vector for the points in X_train_fo
# Hint: Look up the numpy array_split function.
####################################################################
# Split training data into num_folds pieces
x_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
####################################################################
```

```
#                                 END OF YOUR CODE
###################################################################

# A dictionary holding the accuracies for different values of k tha
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving th
# accuracy values that we found when using that value of k.
k_to_accuracies = {}


###################################################################
# TODO:
# Perform k-fold cross validation to find the best value of k. For 
# possible value of k, run the k-nearest-neighbor algorithm num_fol
# where in each case you use all but one of the folds as training d
# last fold as a validation set. Store the accuracies for all fold 
# values of k in the k_to_accuracies dictionary.
###################################################################
# Initialize the dictionary
for k in k_choices:
    k_to_accuracies[k] = []

# Perform cross-validation
for k in k_choices:
    print(f'Evaluating k = {k}')

    for fold in range(num_folds):
        # Create training set: collect all folds except the current
        train_folds_x = []
        train_folds_y = []

        for i in range(num_folds):
            if i != fold:
                train_folds_x.append(x_train_folds[i])
                train_folds_y.append(y_train_folds[i])

        # Concatenate all training folds
        temp_x_train = np.concatenate(train_folds_x)
        temp_y_train = np.concatenate(train_folds_y)

        # Set validation set
        temp_x_val = x_train_folds[fold]
        temp_y_val = y_train_folds[fold]

        # Train classifier
        classifier.train(temp_x_train, temp_y_train)

        # Compute distances on validation fold
        dists_val = classifier.compute_distances_no_loops(temp_x_va

        # Predict labels
        y_pred = classifier.predict_labels(dists_val, k=k)

        # Compute accuracy
        accuracy = np.mean(y_pred == temp_y_val)
        k_to_accuracies[k].append(accuracy)
```

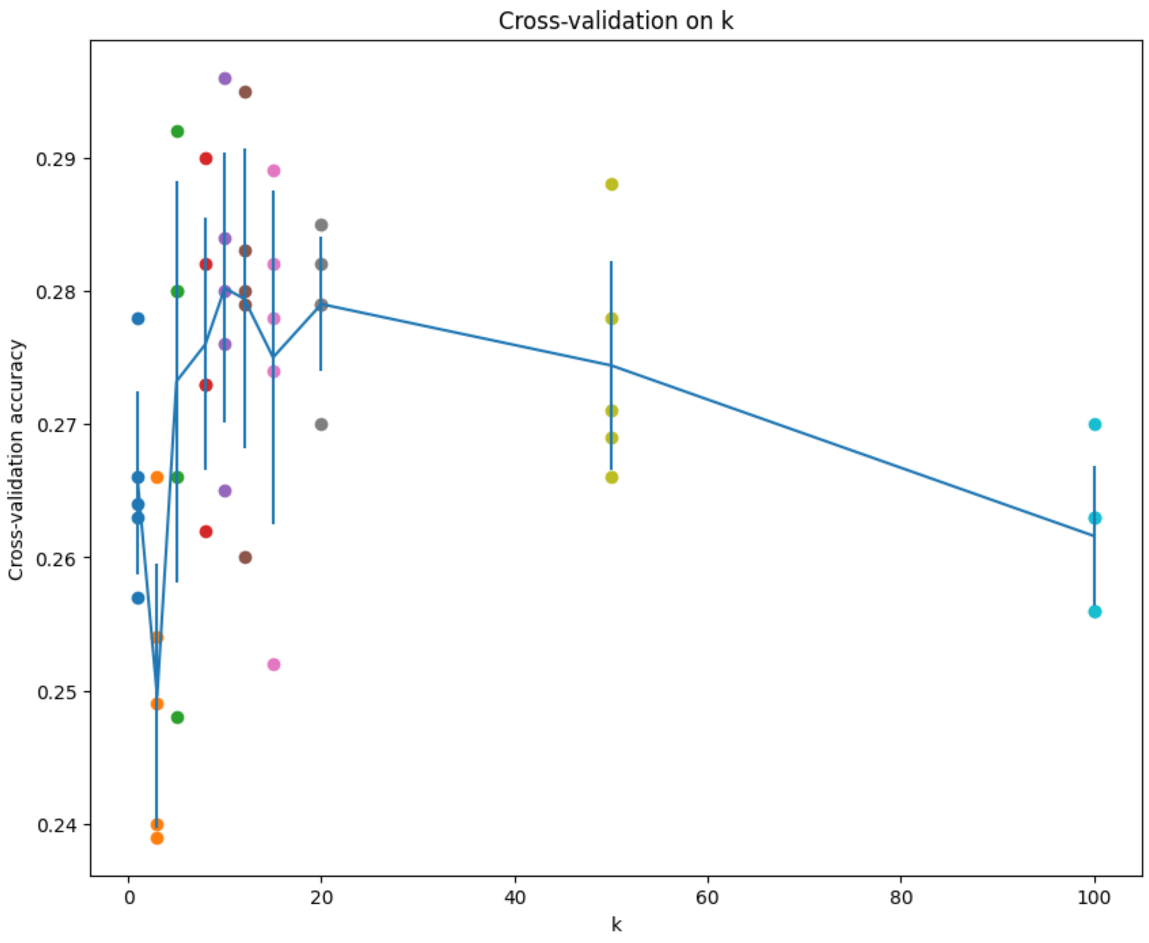
```python
    ################################################################################
    #                                 END OF YOUR CODE
    ################################################################################

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
Evaluating k = 1
Evaluating k = 3
Evaluating k = 5
Evaluating k = 8
Evaluating k = 10
Evaluating k = 12
Evaluating k = 15
Evaluating k = 20
Evaluating k = 50
Evaluating k = 100
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
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

In [17]:

```
# plot the raw observations
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 d
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accur
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accurac
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



In [18]:
```
# Based on the cross-validation results above, choose the best valu
# retrain the classifier using all the training data, and test it o
# data. You should be able to get above 28% accuracy on the test da
best_k = 1

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_tes
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

Inline Question 3 Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply.

1. The training accuracy of a 1-NN (nearest neighbor) will always be better than or equal to that of 5-NN.
2. The test accuracy of a 1-NN will always be better than that of a 5-NN.
3. The decision boundary of the k-NN classifier is linear.
4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.
5. It remembers all of the training data.
6. remembers all of the test data.
7. KNN cannot be applied for distance metrics other than L1 and L2.
8. KNN does more computation during test time than training time.
9. KNN does more computation during training time than test time.
10. KNN does equal computation during training time and test time.
11. KNN does equal computation during training time and test time.

Your Answer: 1, 4, 5, 8

Your explanation:

1. True. On the training set, 1-NN classifies each point as its own label (distance = 0), so its training accuracy should be 100%. Therefore, it is better than or equal to 5-NN.
2. False. 5-NN often generalizes better than 1-NN, as seen in typical results (including my earlier run).
3. False. k-NN creates complex, especially when k increse, piecewise boundaries.
4. True. To classify a test point, k-NN must compute distances to all training points. So, Larger training set = longer classification time.
5. True. k-NN remembers all training data. Like the provided code shows.

6. False. Each test example is classified independently. It shouldn't remember test data we are now using test data to training.
7. False. k-NN works with any distance metric, and it can be used with any to measure distance.
8. True. Most computation is at test time; training is essentially storing data.
9. False. Opposite of 8.
10. False. Training and test costs are not equal, no time was taken during training time.
11. False. Same as 10.

```
In [ ]:
```