In [1]:
```
# %cd daseCV/datasets/
# !bash get_datasets.sh
# %cd ../..
```

K-近邻算法 (kNN) 练习

补充并完成本练习。

kNN分类器包含两个阶段:

- 训练阶段，分类器获取训练数据并简单地记住它。
- 测试阶段, kNN将测试图像与所有训练图像进行比较，并计算出前k个最相似的训练示例的标签来对每个测试图像进行分类。
- 对k值进行交叉验证

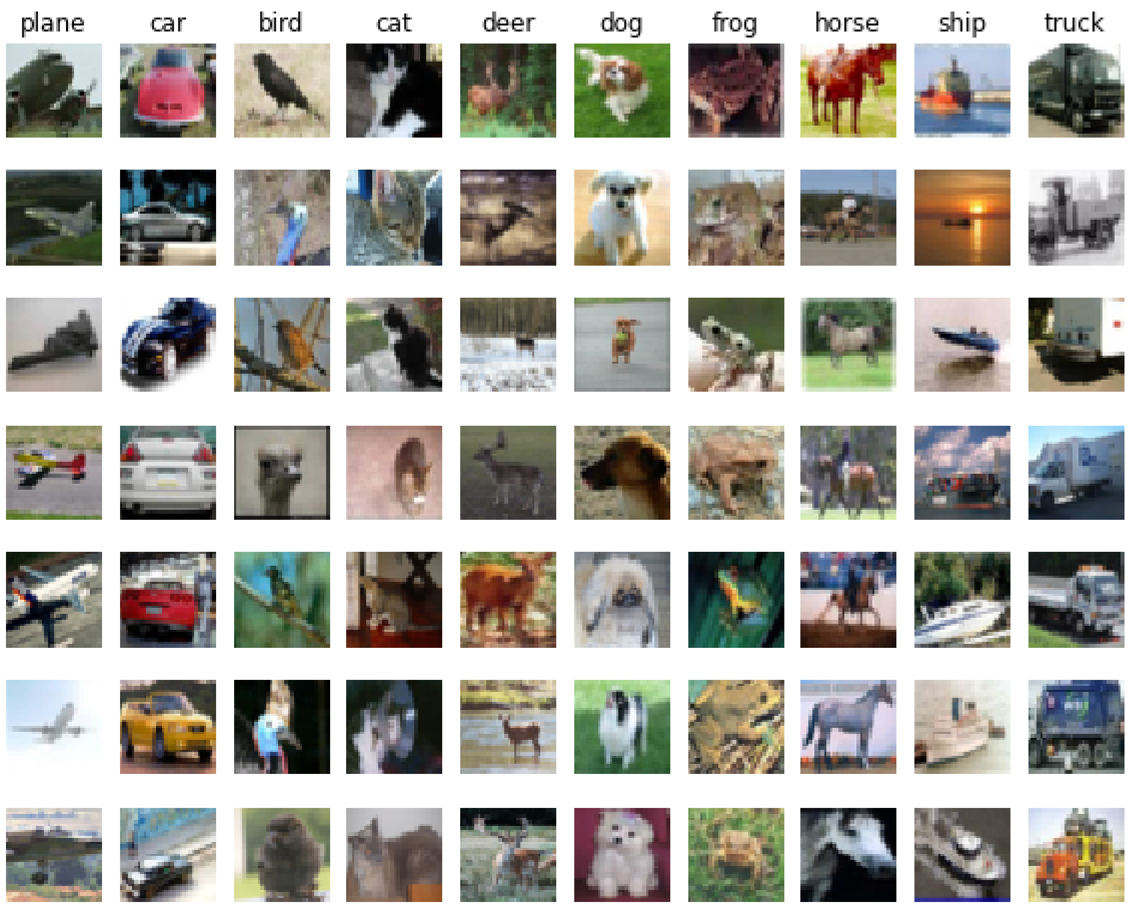
在本练习中，您将实现这些步骤，并了解基本的图像分类、交叉验证和熟练编写高效矢量化代码的能力。

In [2]:
```
# 运行notebook的一些初始化代码

import random
import numpy as np
from daseCV.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# 使得matplotlib的图像在当前页显示而不是新的窗口。
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# 一些更神奇的，使notebook重新加载外部的python模块;
# 参见 http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [3]:
```
# 加载未处理的 CIFAR-10 数据.
cifar10_dir = './input/cifar-10-batches-py'

# 清理变量以防止多次加载数据（这可能会导致内存问题）
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)

# 作为健全性检查，我们打印出训练和测试数据的形状。
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 [4]:
```
# 可视化数据集中的一些示例。
# 我们展示了训练图像的所有类别的一些示例。
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) # flatnonzero表示返回所给数列的非零项的索引值，这里表示
    idxs = np.random.choice(idxs, samples_per_class, replace=False) # replace表示抽取的样本
    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 [5]:
```
# 在练习中使用更小的子样本可以提高代码的效率
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]

# 将图像数据调整为行
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 daseCV.classifiers import KNearestNeighbor
```

```python
# 创建一个kNN分类器实例。
# 请记住，kNN分类器的训练并不会做什么:
# 分类器仅记住数据并且不做进一步处理
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

现在，我们要使用kNN分类器对测试数据进行分类。回想一下，我们可以将该过程分为两个步骤：

1. 首先，我们必须计算所有测试样本与所有训练样本之间的距离。
2. 给定这些距离，对于每个测试示例，我们找到k个最接近的示例，并让它们对标签进行投票

让我们开始计算所有训练和测试示例之间的距离矩阵。 假设有 **Ntr** 的训练样本和 **Nte** 的测试样本, 该过程的结果存储在一个 **Nte x Ntr** 矩阵中，其中每个元素 (i,j) 表示的是第 i 个测试样本和第 j 个 训练样本的距离。

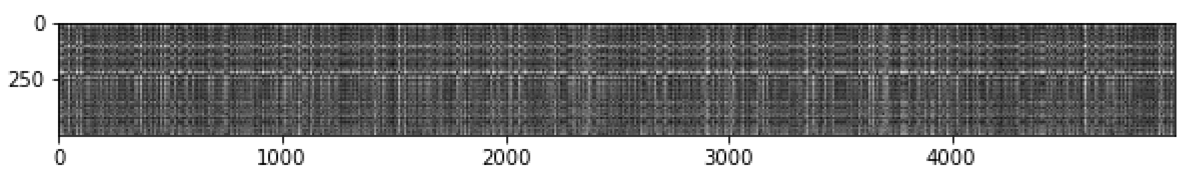
注意: 在完成此notebook中的三个距离的计算时请不要使用numpy提供的np.linalg.norm()函数。

首先打开 `daseCV/classifiers/k_nearest_neighbor.py` 并且补充完成函数 `compute_distances_two_loops` ，这个函数使用双重循环（效率十分低下）来计算距离矩阵。

In [7]:
```python
# 打开 daseCV/classifiers/k_nearest_neighbor.py 并且补充完成
# compute_distances_two_loops.

# 测试你的代码:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

In [8]:
```python
# 我们可视化距离矩阵: 每行代表一个测试样本与训练样本的距离
plt.imshow(dists, interpolation='none')
plt.show()
```



问题 1

请注意距离矩阵中的结构化图案，其中某些行或列的可见亮度更高。（请注意，使用默认的配色方案，黑色表示低距离，而白色表示高距离。）

- 数据中导致行亮度更高的原因是什么？
- 那列方向的是什么原因呢?

答: 首先, 应该明确行与列的含义, 根据维数, 行 i 列 j 表示第 i 个测试样本和第 j 个训练样本的距离, 距离越大, 则这个像素点越亮.

- 那么, 某行 i 亮度更高的说明第 i 个测试样本和每个训练样本的距离都很远. 也就是说, 训练中我们没见过这样的图片.
- 同理, 某列 j 亮度更高说明第 j 个训练样本和每个测试样本都距离很远. 也就是说, 测试中没有和这个图片相似的图片, 粗略地可以认为这张图片的效用不大.

In [9]:
```python
# 现在实现函数predict_labels并运行以下代码:
# 我们使用k = 1 (这是最近的邻居)。
y_test_pred = classifier.predict_labels(dists, k=1)
```

```python
# 计算并打印出预测的精度
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_test, accuracy))
```

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

你预期的精度应该为 `27%` 左右。 现在让我们尝试更大的 `k`, 比如 `k = 5`:

In [10]:
```python
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_test, accuracy))
```

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

你应该能看到一个比 `k = 1` 稍微好一点的结果。

问题 2

我们还可以使用其他距离指标，例如L1距离。

记图像 I_k 的每个位置 (i,j) 的像素值为 $p_{ij}^{(k)}$，

所有图像上的所有像素的均值 μ 为

$$\mu = \frac{1}{nhw}\sum_{k=1}^{n}\sum_{i=1}^{h}\sum_{j=1}^{w} p_{ij}^{(k)}$$

并且所有图像的每个像素的均值 μ_{ij} 为

$$\mu_{ij} = \frac{1}{n}\sum_{k=1}^{n} p_{ij}^{(k)}.$$

标准差 σ 以及每个像素的标准差 σ_{ij} 的定义与之类似。

以下哪个预处理步骤不会改变使用L1距离的最近邻分类器的效果？选择所有符合条件的答案。

1. 减去均值 μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. 减去每个像素均值 μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. 减去均值 μ 然后除以标准偏差 σ.
4. 减去每个像素均值 μ_{ij} 并除以每个素标准差 σ_{ij}.
5. 旋转数据的坐标轴。

你的回答：

[语义1] 5表达的是指标的变换, 则: 1,2,3,5

[语义2] 5表达的是数值的变换, 则: 1,2,3

你的解释：有三种保持度量的变换.

a. 对给定的 i 和 j, 对所有的图像 p, p_{ij} 进行了相同大小的平移, 则距离不会有任何变化(范数的平移不变性).

b. 如果存在常数 c 使得对任一i, j, p_{ij} 变换后变为 $c \cdot p_{ij}$, 则距离的比值不会有任何变化(范数的齐次性).

c. 对图像 p 的指标作任何的变换 $p_{\sigma(ij)}$, 如果 σ 是双射, 则对图片 p 和 q 恒有 $p_{\sigma(ij)} - q_{\sigma(ij)} = p_{ij} - q_{ij}$.

对于语义[1]

1. 是a类变换.
2. 是a类变换.
3. 是a类变换和b类变换的复合.
4. $\{A:(0,0), B:(0,1), C:(2,0)\}$ AB和AC的距离不相等, 变换后AB的距离和AC相等了.
5. 旋转数据的坐标轴是c类变换.

对于语义[2]

项 1..4 没有歧义.

第5项中, 明显地存在旋转变换使得两个点不同, 即 $|\sin\theta + \cos\theta| = \|(\sin\theta, \cos\theta) - (0,0)\|_{l1}$, 其中 $(\sin\theta, \cos\theta)$ 可视为 $(0,1)$ 以 $(0,0)$ 为中心旋转后的结果.

In [11]:

```
# 现在，通过部分矢量化并且使用单层循环的来加快距离矩阵的计算。
# 需要实现函数compute_distances_one_loop并运行以下代码:

dists_one = classifier.compute_distances_one_loop(X_test)

# 为了确保我们的矢量化实现正确，我们要保证它的结果与最原始的实现方式结果一致。
# 有很多方法可以确定两个矩阵是否相似。最简单的方法之一就是Frobenius范数。
# 如果您以前从未了解过Frobenius范数，它其实是两个矩阵的所有元素之差的平方和的平方根;
# 换句话说，就是将矩阵重整为向量并计算它们之间的欧几里得距离。

difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop 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')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

In [12]:

```
# 现在完成compute_distances_no_loops实现完全矢量化的版本并运行代码
dists_two = classifier.compute_distances_no_loops(X_test)

# 检查距离矩阵是否与我们之前计算出的矩阵一致:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop 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')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

In [13]:

```
# 让我们比较一下三种实现方式的速度
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic
```

```python
two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

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

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

# 你应该会看到使用完全矢量化的实现会有明显更佳的性能!

# 注意：在部分计算机上，当您从两层循环转到单层循环时，
# 您可能看不到速度的提升，甚至可能会看到速度变慢。
```

```
Two loop version took 15.232972 seconds
One loop version took 31.535704 seconds
No loop version took 0.185876 seconds
```

交叉验证

我们已经实现了kNN分类器，并且可以设置k = 5。现在，将通过交叉验证来确定此超参数的最佳值。

In [14]:

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

X_train_folds = []
y_train_folds = []
################################################################################
# 需要完成的事情:
# 将训练数据分成多个部分。拆分后，X_train_folds和y_train_folds均应为长度为num_folds的列表，
# 其中y_train_folds [i]是X_train_folds [i]中各点的标签向量。
# 提示：查阅numpy的array_split函数。
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
X_train_folds = np.array_split(X_train, num_folds, axis=0)
y_train_folds = np.array_split(y_train, num_folds, axis=0)

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

# A dictionary holding the accuracies for different values of k that we find when runnin
# 一个字典，存储我们进行交叉验证时不同k的值的精度。
# 运行交叉验证后，k_to_accuracies[k]应该是长度为num_folds的列表，存储了k值下的精度值。
k_to_accuracies = {}

################################################################################
# 需要完成的事情:
# 执行k的交叉验证，以找到k的最佳值。
# 对于每个可能的k值，运行k-最近邻算法 num_folds 次，
# 在每次循环下，你都会用所有拆分的数据（除了其中一个需要作为验证集）作为训练数据。
# 然后存储所有的精度结果到k_to_accuracies[k]中。
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 交叉验证。有时候，训练集数量较小（因此验证集的数量更小），人们会使用一种被称为
# 交叉验证的方法，这种方法更加复杂些。还是用刚才的例子，如果是交叉验证集，我们就
# 不是取1000个图像，而是将训练集平均分成5份，其中4份用来训练，1份用来验证。然后
# 我们循环着取其中4份来训练，其中1份来验证，最后取所有5次验证结果的平均值作为算
# 法验证结果。

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
```

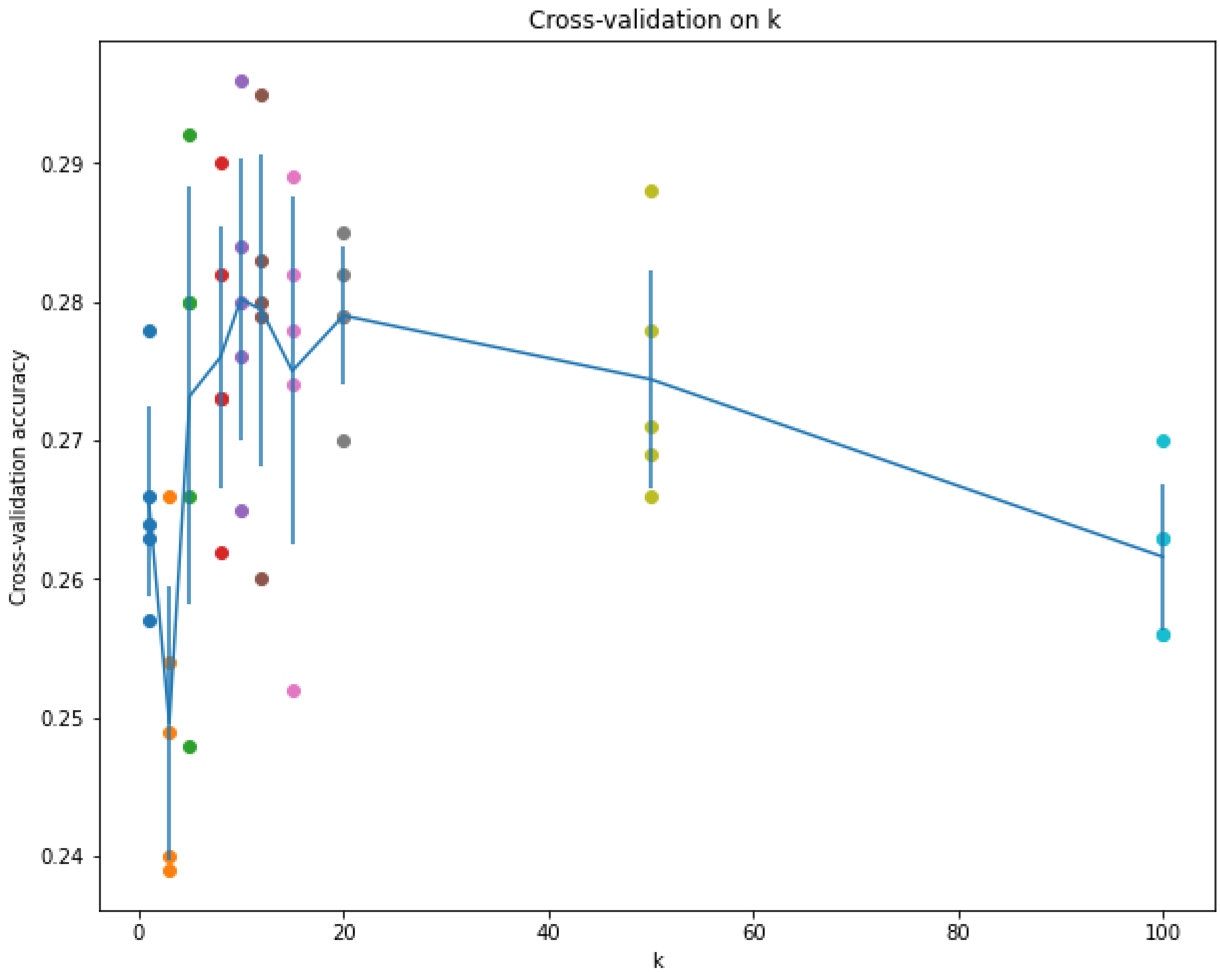
```python
        y_train_fold = np.concatenate([ fold for j, fold in enumerate(y_train_folds) if

        # 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)

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

# 打印出计算的精度
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
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 [15]:

```
# 绘制原始观察结果
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# 用与标准偏差相对应的误差线绘制趋势线
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.show()
```



In [16]:

```
# 根据上述交叉验证结果，为k选择最佳值，使用所有训练数据重新训练分类器，
# 并在测试中对其进行测试数据。您应该能够在测试数据上获得28%以上的准确性。

best_k = k_choices[accuracies_mean.argmax()]

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

# 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_test, accuracy))
```

```
[4. 8. 8. 8. 4. 2. 6. 4. 2. 8. 0. 8. 4. 6. 8. 8. 5. 3. 8. 2. 2. 0. 0. 6.
 2. 4. 4. 7. 4. 2. 4. 2. 4. 3. 8. 6. 2. 8. 2. 4. 8. 6. 2. 4. 0. 8. 5. 0.
```

```
 4. 2. 8. 8. 6. 2. 8. 8. 5. 6. 0. 2. 2. 6. 6. 0. 4. 2. 8. 0. 3. 9. 2. 4.
 8. 8. 0. 2. 8. 3. 6. 8. 8. 6. 2. 0. 2. 8. 2. 8. 8. 8. 0. 2. 0. 2. 2. 2.
 4. 0. 0. 4. 4. 4. 3. 3. 4. 8. 4. 6. 5. 5. 4. 0. 6. 2. 4. 4. 0. 4. 4. 4.
 8. 4. 2. 2. 8. 4. 8. 6. 4. 2. 2. 8. 8. 6. 2. 2. 9. 0. 2. 8. 2. 8. 0. 2.
 8. 0. 6. 2. 5. 5. 0. 3. 0. 8. 4. 4. 0. 8. 5. 4. 0. 2. 2. 4. 8. 0. 8. 2.
 5. 0. 8. 7. 2. 8. 4. 9. 0. 4. 4. 0. 8. 3. 2. 4. 2. 0. 6. 6. 1. 8. 2. 4.
 2. 2. 4. 4. 8. 0. 4. 0. 2. 7. 8. 4. 8. 4. 0. 2. 2. 0. 4. 8. 0. 7. 8. 0.
 0. 9. 0. 4. 6. 6. 0. 2. 2. 2. 4. 2. 3. 4. 3. 1. 2. 4. 0. 2. 6. 9. 0. 0.
 0. 0. 8. 0. 2. 4. 6. 0. 4. 3. 2. 3. 4. 6. 2. 8. 2. 0. 8. 4. 5. 8. 4. 2.
 2. 8. 8. 4. 0. 0. 2. 6. 6. 3. 8. 2. 8. 2. 6. 6. 8. 4. 2. 8. 0. 4. 5. 8.
 2. 8. 2. 4. 2. 8. 3. 4. 2. 0. 8. 2. 2. 6. 4. 2. 4. 8. 8. 2. 1. 4. 8. 4.
 0. 0. 2. 0. 2. 4. 6. 4. 5. 6. 6. 4. 2. 8. 0. 2. 2. 2. 4. 9. 2. 4. 2. 4.
 6. 8. 0. 4. 4. 2. 0. 4. 8. 4. 0. 6. 8. 8. 2. 4. 8. 2. 2. 2. 2. 4. 8. 0.
 2. 5. 8. 0. 0. 8. 4. 2. 4. 4. 2. 8. 0. 8. 4. 2. 8. 2. 2. 2. 2. 4. 0. 0.
 2. 0. 8. 2. 2. 6. 0. 0. 2. 5. 2. 4. 4. 4. 8. 3. 9. 4. 2. 8. 0. 6. 0. 0.
 0. 2. 8. 8. 2. 4. 9. 8. 0. 5. 4. 7. 6. 4. 8. 2. 2. 8. 5. 4. 4. 6. 2. 8.
 2. 4. 8. 4. 0. 4. 6. 4. 2. 4. 9. 8. 5. 4. 8. 0. 2. 0. 6. 8. 4. 2. 8. 2.
 2. 4. 6. 8. 4. 4. 2. 4. 0. 2. 4. 2. 3. 8. 2. 4. 8. 4. 2. 4. 2. 0. 2. 4.
 8. 2. 8. 4. 2. 8. 9. 4. 8. 0. 0. 8. 8. 1. 8. 8. 2. 0. 2. 0.]
Got 141 / 500 correct => accuracy: 0.282000
```

问题 3

下列关于k-NN的陈述中哪些是在分类器中正确的设置，并且对所有的k都有效？选择所有符合条件的选项。

1. k-NN分类器的决策边界是线性的。
2. 1-NN的训练误差将始终低于5-NN。
3. 1-NN的测试误差将始终低于5-NN。
4. 使用k-NN分类器对测试示例进行分类所需的时间随训练集的大小而增加。
5. 以上都不是。

你的回答：2, 4

你的解释：

1. k-NN分类器可能得到分段线性的决策边界, 或者甚至拓扑上和线性边界不同胚的边界.
2. 因为1-NN的样本总是和自己最近, 而5-NN的样本可能周围有很近的其他类别样本.
3. 未必, 可能存在错误的孤例.
4. 是, k-NN分类器单次分类的时间复杂度是 $O(N)$ 的, 其中 N 是训练样本个数.
5. 2,4正确, 因此5错误.

Data for leaderboard

这里额外提供了一组未给标签的测试集X，用于leaderborad上的竞赛。

提示：该题的目的是鼓励同学们探索能够提升模型性能的方法。

In [7]:
```python
################################################################################
# 需要完成的事情:
# 找到更合适的classifer
# 找到更合适的best_k
# 提示：如果你不想花时间，你也可以直接使用上面已经训练好的classifer和已确定的best_k.
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
from final_clf.model import *
from final_clf.data_util import *
from final_clf.metric import *
import torch
```

```python
best_k = 85
device = torch.device('cuda:0')
# leaderboard的测试数据
x_tst = np.load("./input/X_3072.npy").reshape(-1, 32, 32, 3)

def predict(timestamp, x_tst, path='final_clf/ckpt'):
    with torch.no_grad():
        print('load data')
        x_trn, y_trn, _, _ = load_cifar_tensor(None, None, device)
        x_trn = x_trn.to(torch.float)
        y_trn = y_trn.to(torch.float)
        x_tst = torch.from_numpy(x_tst).to(torch.float).to(device)
        print('load model')
        model = ContrastClf(l2_metric).to(device)
        model.load_state_dict(torch.load(f'{path}/{timestamp}.model'))
        print('compute features')
        x_trn_feat = model.batch_feat_as_numpy(x_trn)
        x_tst_feat = model.batch_feat_as_numpy(x_tst)
        del x_trn, x_tst, model
        print('compute distance')
        dist = torch.cdist(
            torch.from_numpy(x_tst_feat).to(device),
            torch.from_numpy(x_trn_feat).to(device), p=2
        ).cpu()
        del x_tst_feat, x_trn_feat
        print('sort distance')
        index = dist.argsort(dim=-1)
        del dist
        print('get label of k-nn instances')
        y_top = y_trn[index[:, :best_k]]
        y_out = torch.stack(tuple((y_top == i).sum(-1) for i in range(10)), dim=-1).argm
        return y_out.cpu().numpy()

preds = predict('202210051318', x_tst)
```

```
load data
load model
compute features
compute 0/50000
compute 500/50000
compute 1000/50000
compute 1500/50000
compute 2000/50000
compute 2500/50000
compute 3000/50000
compute 3500/50000
compute 4000/50000
compute 4500/50000
compute 5000/50000
compute 5500/50000
compute 6000/50000
compute 6500/50000
compute 7000/50000
compute 7500/50000
compute 8000/50000
compute 8500/50000
compute 9000/50000
compute 9500/50000
compute 10000/50000
compute 10500/50000
compute 11000/50000
compute 11500/50000
compute 12000/50000
compute 12500/50000
compute 13000/50000
compute 13500/50000
```

```
compute 14000/50000
compute 14500/50000
compute 15000/50000
compute 15500/50000
compute 16000/50000
compute 16500/50000
compute 17000/50000
compute 17500/50000
compute 18000/50000
compute 18500/50000
compute 19000/50000
compute 19500/50000
compute 20000/50000
compute 20500/50000
compute 21000/50000
compute 21500/50000
compute 22000/50000
compute 22500/50000
compute 23000/50000
compute 23500/50000
compute 24000/50000
compute 24500/50000
compute 25000/50000
compute 25500/50000
compute 26000/50000
compute 26500/50000
compute 27000/50000
compute 27500/50000
compute 28000/50000
compute 28500/50000
compute 29000/50000
compute 29500/50000
compute 30000/50000
compute 30500/50000
compute 31000/50000
compute 31500/50000
compute 32000/50000
compute 32500/50000
compute 33000/50000
compute 33500/50000
compute 34000/50000
compute 34500/50000
compute 35000/50000
compute 35500/50000
compute 36000/50000
compute 36500/50000
compute 37000/50000
compute 37500/50000
compute 38000/50000
compute 38500/50000
compute 39000/50000
compute 39500/50000
compute 40000/50000
compute 40500/50000
compute 41000/50000
compute 41500/50000
compute 42000/50000
compute 42500/50000
compute 43000/50000
compute 43500/50000
compute 44000/50000
compute 44500/50000
compute 45000/50000
compute 45500/50000
compute 46000/50000
compute 46500/50000
```

```
compute 47000/50000
compute 47500/50000
compute 48000/50000
compute 48500/50000
compute 49000/50000
compute 49500/50000
compute 0/500
compute distance
sort distance
get label of k-nn instances
```

提醒：运行完下面代码之后，点击下面的submit，然后去leaderboard上查看你的成绩。本模型对应的成绩在phase1的leaderboard中。

In [8]:

```python
import os
#输出格式
def output_file(preds, phase_id=1):
    path=os.getcwd()
    if not os.path.exists(path + '/output/phase_{}'.format(phase_id)):
        os.mkdir(path + '/output/phase_{}'.format(phase_id))
    path=path + '/output/phase_{}/prediction.npy'.format(phase_id)
    np.save(path,preds)
def zip_fun(phase_id=1):
    path=os.getcwd()
    output_path = path + '/output'
    files = os.listdir(output_path)
    for _file in files:
        if _file.find('zip') != -1:
            os.remove(output_path + '/' + _file)
    newpath=path+'/output/phase_{}'.format(phase_id)
    os.chdir(newpath)
    cmd = 'zip ../prediction_phase_{}.zip prediction.npy'.format(phase_id)
    os.system(cmd)
    os.chdir(path)
output_file(preds)
zip_fun()
```