

knn

2021 年 4 月 8 日

```
[ ]: from google.colab import drive

drive.mount('/content/drive', force_remount=True)

# 输入 daseCV 所在的路径
# 'daseCV' 文件夹包括 '.py', 'classifiers' 和 'datasets' 文件夹
# 例如 'CV/assignments/assignment1/daseCV/'
FOLDERNAME = 'CV/assignments/assignment1/daseCV/'

assert FOLDERNAME is not None, "[!] Enter the foldername."

%cd drive/My\ Drive
%cp -r $FOLDERNAME ../../
%cd ../../
%cd daseCV/datasets/
!bash get_datasets.sh
%cd ../../
```

```
Mounted at /content/drive
/content/drive/My Drive
/content
/content/daseCV/datasets
--2021-04-03 09:01:31-- http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
```

```
Saving to: 'cifar-10-python.tar.gz'
```

```
cifar-10-python.tar 100%[=====>] 162.60M 45.7MB/s in 4.0s
```

```
2021-04-03 09:01:35 (40.9 MB/s) - 'cifar-10-python.tar.gz' saved  
[170498071/170498071]
```

```
cifar-10-batches-py/  
cifar-10-batches-py/data_batch_4  
cifar-10-batches-py/readme.html  
cifar-10-batches-py/test_batch  
cifar-10-batches-py/data_batch_3  
cifar-10-batches-py/batches.meta  
cifar-10-batches-py/data_batch_2  
cifar-10-batches-py/data_batch_5  
cifar-10-batches-py/data_batch_1  
/content
```

1 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。

```
[ ]: # 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
```

```
[ ]: # 加载未处理的 CIFAR-10 数据.
cifar10_dir = 'daseCV/datasets/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)
```

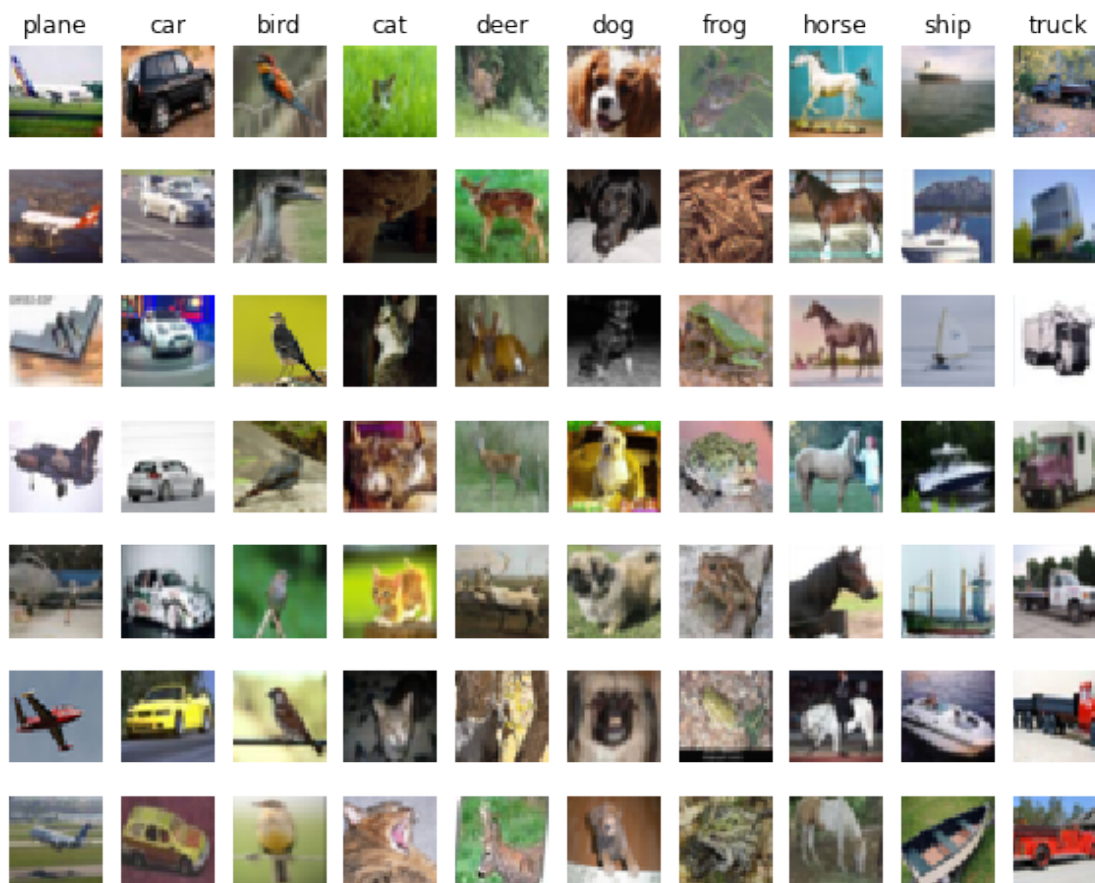
Training data shape: (50000, 32, 32, 3)

Training labels shape: (50000,)

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

Test labels shape: (10000,)

```
[ ]: # 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，这里表示返回所有属于 y 类的索引
    idxs = np.random.choice(idxs, samples_per_class, replace=False) # replace 表
示抽取的样本是否能重复
    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()
```



```
[ ]: # 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)
```

```
[ ]: X_test[1:3,2:6][1,:]
```

```
[ ]: array([222., 158., 187., 218.])
```

```
[ ]: np.sqrt(sum([3,6]))
```

```
[ ]: 3.0
```

```
[ ]: [i for i in range(3)]
```

```
[ ]: [0, 1, 2]
```

```
[ ]: from daseCV.classifiers import KNearestNeighbor
```

```
# 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。

```
[ ]: X_train[0,2:5]**2+X_train[1,2:5]**2
```

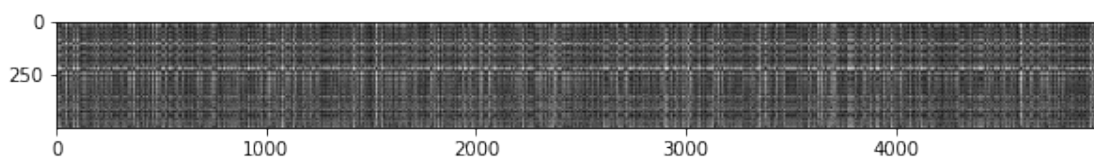
```
[ ]: array([38938., 17725., 20885.])
```

```
[ ]: # 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)
```

```
[ ]: # 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：对行：某测试数据与原数据距离越近，越相似，亮度越低。对列：某训练数据与测试数据距离越近，越相似，亮度越低。

```
[ ]: # 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)

# 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:

```
[ ]: 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 : 123

你 : 1, 问的是分类效果, 而平移不影响相对位置和临界线, 所以没影响 2, 每个位置像素减去同样的均值, 不影响相对距离 3, 相当于放缩, 对分类效果没影响 4, 图像与图像的距离会变化 5, 对于 L2 距离而言不变, 对 L1 距离不符合

```
[ ]: # 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

```
[ ]: # 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

```
[ ]: # 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

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 40.961476 seconds

One loop version took 39.660156 seconds

No loop version took 0.575312 seconds

1.0.1 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。

```
[ ]: 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)*****

y_train_ = y_train.reshape(-1, 1)
X_train_folds, y_train_folds = np.array_split(X_train, 5), np.
    ↳array_split(y_train_, 5)

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

# A dictionary holding the accuracies for different values of k that we find
    ↳when running cross-validation.
# 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.setdefault(k_, [])
for i in range(num_folds):
    classifier = KNearestNeighbor()
    X_val_train = np.vstack(X_train_folds[0:i] + X_train_folds[i+1:])
    y_val_train = np.vstack(y_train_folds[0:i] + y_train_folds[i+1:])
    y_val_train = y_val_train[:,0]
    classifier.train(X_val_train, y_val_train)
    for k_ in k_choices:
        y_val_pred = classifier.predict(X_train_folds[i], k=k_)
        num_correct = np.sum(y_val_pred == y_train_folds[i][:,0])
        accuracy = float(num_correct) / len(y_val_pred)
        k_to_accuracies[k_] = k_to_accuracies[k_] + [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

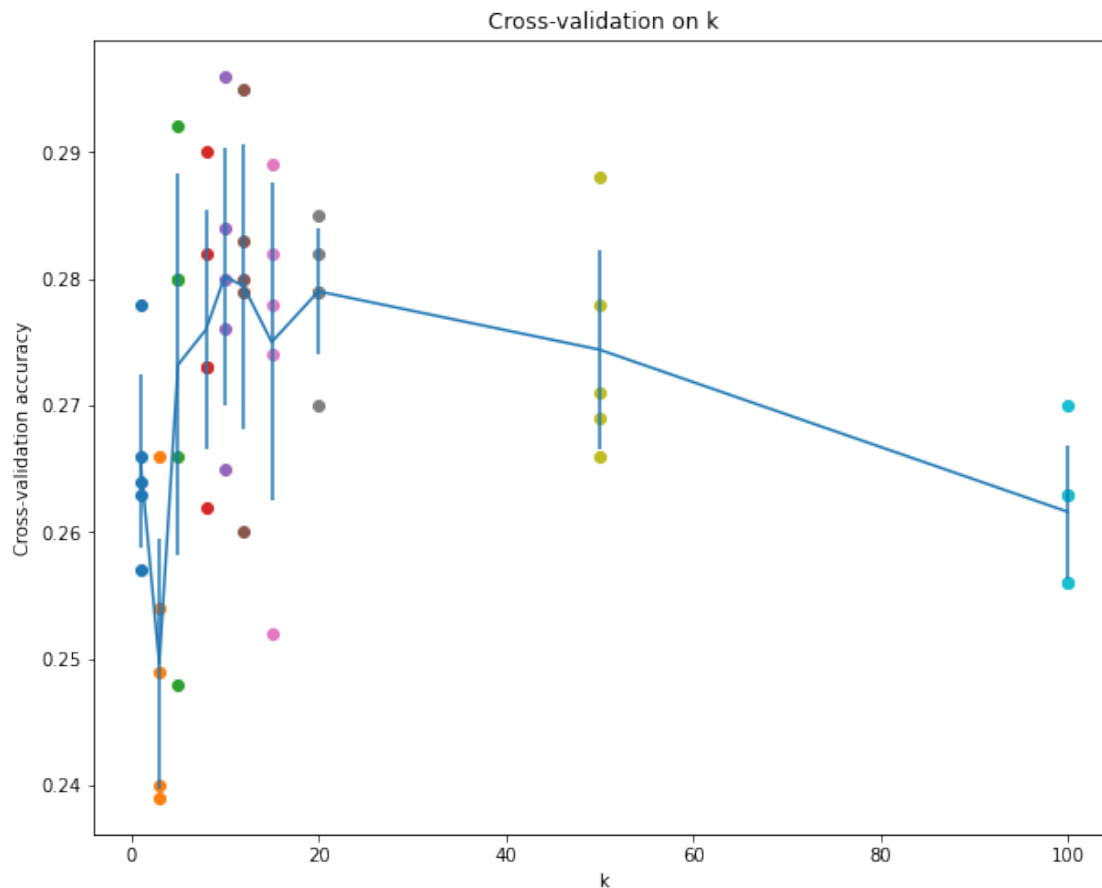
k = 3, accuracy = 0.249000

```
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
```

```
[ ]: # 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()
```



```
[ ]: # 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)

# 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))
```

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 : 24

你 : 1NN 训练不始终百分百吗 4 是的，但是增加复杂度没这么高，因为可以 noloops 去写

2 重要

这里是作业的结尾处，请执行以下步骤：

1. 点击 File -> Save 或者用 `control+s` 组合键，确保你最新的 notebook 的作业已经保存到谷歌云。
2. 执行以下代码确保 `.py` 文件保存回你的谷歌云。

```
[ ]: import os

FOLDER_TO_SAVE = os.path.join('drive/My Drive/', FOLDERNAME)
FILES_TO_SAVE = ['daseCV/classifiers/k_nearest_neighbor.py']

for files in FILES_TO_SAVE:
    with open(os.path.join(FOLDER_TO_SAVE, '/' + files.split('/')[1:]), 'w') as f:
        f.write(''.join(open(files).readlines()))
```