DSE 10 Homework 4 - Due 02/05/2021 9:00am

Instructions

- Please upload all work to gradescope by the due date - **late work will not be graded**
  - You should submit a single ipynb file containing all your code and output (plots, numeric values, etc...)
  - Important: **name your ipynb file according to the following convention**: `<lastname>_<firstname>.ipynb` - example: (`thomas_anthony.ipynb`)
  - Please organize your notebook into sections by problem
  - Use print statements to clearly indicate which question you are ansering
    - Good: `print("Problem 4 (e): {}".format(np.pi))`
    - Good: `print("The value of pi is: {}".format(np.pi))`
    - Bad: `print(np.pi)`
    - Bad: `np.pi`
  - Use relative paths to load data:
    - Good: `loadmnist("train-images-idx3-ubyte", "train-labels-idx1-ubyte")`
    - Bad: `loadmnist("/home/anthony/class/DSE10/train-images-idx3-ubyte", ...)`
- Collaboration is encouraged, but all submissions should be in your own writing/code and written with your own understanding
- Your code must run and be able to reproduce your answers
- Unless stated otherwise in the assignemnt, your code should use only basic low-level NumPy/SciPy linear algebra and satistics commands (e.g. do not use built in estimation tools or anything from SciKitLearn). When in doubt, ask for clarification on Canvas.

In [1]:
```python
%matplotlib inline
import matplotlib.pyplot as plt
import time
import gzip
import sys
import os
import copy
import numpy as np
import pandas as pd
import pickle
import string
import operator
import bz2
import random
from scipy import stats
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import KMeans
from sklearn.neighbors import BallTree
from sklearn import metrics
from sklearn.preprocessing import scale
from pylab import rcParams
from struct import unpack
from scipy.stats import multivariate_normal

if sys.version_info[0] == 2:
    from urllib import urlretrieve
else:
    from urllib.request import urlretrieve

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Worksheet 9 Problem 3

Text classification using multinomial Naive Bayes.

(a) For this problem, you'll be using the 20 Newsgroups data set. There are several versions of it on the web. You should download "20news-bydate.tar.gz" from

`http://qwone.com/~jason/20Newsgroups/`

Unpack it and look through the directories at some of the files. Overall, there are roughly 19,000 documents, each from one of 20 newsgroups. The label of a document is the identity of its newsgroup. The documents are divided into a training set and a test set.

In [2]:
```python
def download(filename, source = 'http://qwone.com/~jason/20Newsgroups/20news-bydate.tar.gz'):
    print("Downloading %s" % filename)
    urlretrieve(source + filename, filename)

def load_data(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset = 16)
    return data / np.float32(256)
```

In [3]:
```python
newsgroup_data = load_data('20news-bydate.tar.gz')
```

(b) The same website has a processed version of the data, "20news-bydate-matlab.tgz", that is par- ticularly convenient to use. Download this and also the file "vocabulary.txt". Look at the first training document in the processed set and the corresponding original text document to under- stand the relation between the two.

In [4]:
```python
train_data = pd.read_csv('20news-bydate 2/matlab/train.data', header = None, sep =' ')
train_label = pd.read_csv('20news-bydate 2/matlab/train.label', header = None, sep = ' ')
train_map = pd.read_csv('20news-bydate 2/matlab/train.map', header = None, sep = ' ')
test_data = pd.read_csv('20news-bydate 2/matlab/test.data', header = None, sep =' ')
test_label = pd.read_csv('20news-bydate 2/matlab/test.label', header = None, sep =' ')
test_map = pd.read_csv('20news-bydate 2/matlab/test.map', header = None, sep = ' ')
vocabulary_data = pd.read_csv('vocabulary.txt', names = ['word'], header = None, sep =' ')
```

(c) The words in the documents constitute an overall vocabulary V of size 61188. Build a multinomial Naive Bayes model using the training data. For each of the 20 classes j = 1, 2, . . . , 20, you must have the following:

```
• pij, the fraction of documents that belong to that class; and
• Pj, a probability distribution over V that models the documents of that class.
```

In order to fit Pj, imagine that all the documents of class j are strung together. For each word w 2 V , let Pjw be the fraction of this concatenated document occupied by w. Well, almost: you will need to do smoothing (just add one to the count of how often w occurs).

In [5]:
```python
x_train_data, y_train_label, x_test_data, y_test_label = train_data, train_label, test_data, test_label
```

In [6]:
```python
x_train_data.columns = ['docIdx', 'wordIdx', 'count']
y_train_label['docIdx'] = range(1, len(y_train_label) + 1)
y_train_label.columns = ['class', 'docIdx']
x_train_data = x_train_data.merge(y_train_label, how = 'left', on = 'docIdx')
```

In [7]:
```python
x_test_data.columns = ['docIdx', 'wordIdx', 'count']
y_test_label['docIdx'] = range(1, len(y_test_label) + 1)
y_test_label.columns = ['class', 'docIdx']
x_test_data = x_test_data.merge(y_test_label, how = 'left',on = 'docIdx')
```

In [8]:
```python
# calculate pi
pi = np.zeros(20)
for j in range(20):
    pi[j] = np.sum(y_train_label.iloc[:,0] == j + 1)/len(y_train_label)

print('Fraction of documents in each class:\n', pi)
```

```
Fraction of documents in each class:
 [0.04259473 0.05155737 0.05075872 0.0520898  0.05102494 0.0525335
 0.05164611 0.0525335  0.05288846 0.05271098 0.05306593 0.05271098
 0.05244476 0.05271098 0.05262224 0.05315467 0.04836277 0.05004881
 0.0411749  0.03336587]
```

In [9]:
```python
# calculate p
total = vocabulary_data.shape[0]
p = np.zeros((20,total ))

for j in range(20):
    tmp = x_train_data[x_train_data['class'] == j + 1]
    p[j,:] = 1
    for i in range(tmp.shape[0]):
        wdidx = tmp['wordIdx'].iloc[i]
        count = tmp['count'].iloc[i]
        p[j, wdidx - 1] += count
    p[j,:] = p[j,:]/np.sum(p[j,:])

print('Probability Distribution:\n', p)
```

```
Probability Distribution:
 [[6.66666667e-05 3.04761905e-04 1.31428571e-03 ... 4.76190476e-06
  4.76190476e-06 4.76190476e-06]
 [3.55589754e-04 3.49760414e-04 5.82934024e-06 ... 5.82934024e-06
  5.82934024e-06 5.82934024e-06]
 [7.89707479e-05 4.60662696e-04 6.58089566e-06 ... 6.58089566e-06
  6.58089566e-06 6.58089566e-06]
 ...
 [3.48108977e-05 4.90517195e-04 3.16462706e-06 ... 3.16462706e-06
  3.16462706e-06 3.16462706e-06]
 [4.03854386e-06 1.61541755e-04 4.03854386e-06 ... 4.03854386e-06
  4.03854386e-06 4.03854386e-06]
 [5.54680393e-06 2.55152981e-04 5.54680393e-05 ... 5.54680393e-06
  5.54680393e-06 5.54680393e-06]]
```

(d) Write a routine that uses this naive Bayes model to classify a new document. To avoid underflow, work with logs rather than multiplying together probabilities.

In [10]:
```python
def decision(x_test_data, pi, p):
    wdidx = x_test_data['wordIdx']
    prob = [np.log(pi[j]) + np.sum(x_test_data['count'] * np.log(p[j,wdidx - 1 ])) for j in range(20)]
    pred = np.argmax(prob)

    return pred , prob
```

(e) Evaluate the performance of your model on the test data. What error rate do you achieve?

In [11]:
```python
result = []
for i in range(y_test_label.shape[0]):
    x_test = x_test_data.loc[x_test_data['docIdx'] == i + 1,:]
    pred, prob = decision(x_test, pi, p)
    result.append(pred + 1 != y_test_label['class'].iloc[i])

# calculate the error rate of naive Bayes model
error_rate = np.mean(result) * 100
print('Error rate of naive Bayes model is:\n{}%'.format(error_rate))
```

```
Error rate of naive Bayes model is:
21.892071952031976%
```

(f) Split the training data into a smaller training set and a validation set. The split could be 80-20, for instance. You'll use this training set to estimate parameters and the validation set to decide between different options.

In [12]:
```python
x, y = x_train_data, y_train_label
```

In [13]:
```python
# randomly select 80% data from data set as training data set.
count = int(x.shape[0]*0.8)
random.seed(10)
sel = random.sample(range (0, x.shape[0]),  count)
x_train_set, y_train_set = x_train_data.loc[x_train_data.index.intersection(sel),:], y_train_label.loc[y_tra

print('Shape of x_train_set:\n', x_train_set.shape)
print('\nShape of y_train_set:\n', y_train_set.shape)
```

```
Shape of x_train_set:
 (1173876, 4)

Shape of y_train_set:
 (9087, 2)
```

In [14]:
```python
# get the remaining data as validation data set.
remain = np.setdiff1d(range(0, x.shape[0]), sel)
x_validation_set, y_validation_set = x_train_data.loc[x_train_data.index.intersection(remain),], y_train_lab

print('Shape of x_validation_set:\n', x_validation_set.shape)
print('\nShape of y_validation_set:\n', y_validation_set.shape)
```

```
Shape of x_validation_set:
 (293469, 4)

Shape of y_validation_set:
 (2182, 2)
```

(i) replacing the frequency f of a word in a document by log(1 + f)

In [15]:
```python
# calculate pi_f
pi_f = np.zeros(20)
for j in range(20):
    pi_f[j] = np.sum(y_train_set.iloc[:,0] == j + 1)/len(y_train_set)

print('Fraction of documents in each class:\n', pi_f)
```

```
Fraction of documents in each class:
 [0.0432486  0.05051172 0.05150215 0.05227248 0.05216243 0.05172224
 0.05271267 0.05227248 0.05249257 0.05238252 0.05425333 0.053483
 0.0532629  0.05216243 0.05205238 0.05172224 0.04842082 0.04963134
 0.04071751 0.0330142 ]
```

In [16]:
```python
# calculate p_f
p_f = np.zeros((20,total))

for j in range(20):
    tmp = x_train_set[x_train_set['class'] == j + 1]
    p_f[j,:] = 1
    for i in range(tmp.shape[0]):
        wdidx = tmp['wordIdx'].iloc[i]
        count = tmp['count'].iloc[i]
        p_f[j, wdidx - 1] += count
    p_f[j,:] = p_f[j,:]/np.sum(p_f[j,:])

print('Probability Distribution:\n', p_f)
```

```
Probability Distribution:
 [[7.77168995e-05 2.38701906e-04 1.29343126e-03 ... 5.55120711e-06
  5.55120711e-06 5.55120711e-06]
 [4.04435307e-04 2.83104715e-04 6.74058845e-06 ... 6.74058845e-06
  6.74058845e-06 6.74058845e-06]
 [8.21925997e-05 3.88546835e-04 7.47205452e-06 ... 7.47205452e-06
  7.47205452e-06 7.47205452e-06]
 ...
 [4.13868352e-05 4.74067385e-04 3.76243957e-06 ... 3.76243957e-06
  3.76243957e-06 3.76243957e-06]
 [4.74858611e-06 1.28211825e-04 4.74858611e-06 ... 4.74858611e-06
  4.74858611e-06 4.74858611e-06]
 [6.37897490e-06 2.74295921e-04 5.10317992e-05 ... 6.37897490e-06
  6.37897490e-06 6.37897490e-06]]
```

In [17]:
```python
def decision_f(x_test, pi, p):
    wdidx = x_test['wordIdx']
    prob = [np.log(pi[j]) + np.sum(np.log(1 + x_test['count']) * np.log(p[j,wdidx - 1])) for j in range(20)]
    pred = np.argmax(prob)

    return pred, prob
```

In [18]:
```python
result_f = []
for i in range(y_validation_set.shape[0]):
    x_test = x_validation_set.loc[x_validation_set['docIdx'] == i + 1,:]
    pred_f, prob_f = decision_f(x_test, pi_f, p_f)
    result_f.append(pred_f + 1 != y_validation_set['class'].iloc[i])

# calculate the error rate of this regular model
error_rate_f = np.mean(result_f) * 100
print('Error rate of smooth model on validation model is:\n{}%'.format(error_rate_f))
print('The error rate for validation data is high. The reason why it is so high is because the validation se
```

```
Error rate of smooth model on validation model is:
96.05866177818515%
The error rate for validation data is high. The reason why it is so high is because the validation set is to
o small. If you use the test data set, the error rate reduce to around 20.9%
```

(ii) removing stopwords

In [19]:
```python
# stop words from online
stop_words = [
"a", "about", "above", "across", "after", "afterwards",
"again", "all", "almost", "alone", "along", "already", "also",
"although", "always", "am", "among", "amongst", "amoungst", "amount", "an", "and", "another", "any", "anyhow
"this", "those", "though", "through", "throughout",
"thru", "thus", "to", "together", "too", "toward", "towards",
"under", "until", "up", "upon", "us",
"very", "was", "we", "well", "were", "what", "whatever", "when",
"whence", "whenever", "where", "whereafter", "whereas", "whereby",
"wherein", "whereupon", "wherever", "whether", "which", "while",
"who", "whoever", "whom", "whose", "why", "will", "with",
"within", "without", "would", "yet", "you", "your", "yours", "yourself", "yourselves"
]
```

In [20]:
```python
# find all bad words
vocabulary_data = vocabulary_data.reset_index()
vocabulary_data['index'] = vocabulary_data['index'].apply(lambda x: x+1)
total_list = set(vocabulary_data['index'])
vocabulary_data = vocabulary_data[~vocabulary_data['word'].isin(stop_words)]
good_list = vocabulary_data['index'].tolist()
good_list = set(good_list)
bad_list = total_list - good_list
```

In [21]:
```python
a = 0.001

for bad in bad_list:
    for j in range(20):
        # a Laplace Smoothing with low α for bad words
        p_f[j][bad] = a/(np.sum(p_f[j,:]) + vocabulary_data.shape[0] + 1)
```

In [22]:
```python
def decision_s(x_test, pi, p):
    wdidx = x_test['wordIdx']
    IDF = np.log(len(x_test['docIdx'].unique())/(x_test.groupby(['wordIdx'])['docIdx'].count()) + 1)
    prob = [np.log(pi[j]) + np.sum(x_test['count'] * np.log(IDF * p[j,wdidx - 1])) for j in range(20)]
    pred = np.argmax(prob)

    return pred, prob
```

In [23]:
```python
result_s = []
for i in range(y_validation_set.shape[0]):
    x_test = x_validation_set.loc[x_validation_set['docIdx'] == i + 1,:]
    pred_s, prob_s = decision_s(x_test, pi_f, p_f)
    result_s.append(pred_s + 1 != y_validation_set['class'].iloc[i])

# calculate the error rate of this regular model
error_rate_s = np.mean(result_s) * 100
print('Error rate of IDF model on validation data is:\n{}%'.format(error_rate_s))
```

```
Error rate of IDF model on validation data is:
95.18790100824931%
```

Evaluate your final model on the test data. What error rate do you achieve?

In [24]:
```python
# evaluate naive Bayes model on the test data

result = []
for i in range(y_test_label.shape[0]):
    x_test = x_test_data.loc[x_test_data['docIdx'] == i + 1,:]
    pred, prob = decision(x_test, pi, p)
    result.append(pred + 1 != y_test_label['class'].iloc[i])

# calculate the error rate of naive Bayes model
error_rate = np.mean(result) * 100
print('Error rate of naive Bayes model is:\n{}%'.format(error_rate))
```

```
Error rate of naive Bayes model is:
21.892071952031976%
```

In [25]:
```python
# evaluate smooth model on the test data

result_f_test_data = []
for i in range(y_test_label.shape[0]):
    x_test = x_test_data.loc[x_test_data['docIdx'] == i + 1,:]
    pred, prob = decision_f(x_test, pi, p)
    result_f_test_data.append(pred + 1 != y_test_label['class'].iloc[i])

# calculate the error rate of naive Bayes model
error_rate_f_test_data = np.mean(result_f_test_data) * 100
print('Error rate of smooth model on test data is:\n{}%'.format(error_rate_f_test_data))
```

```
Error rate of smooth model on test data is:
20.919387075283144%
```

In [26]:
```python
# evaluate IDF model on the test data

result_s_test_data = []
for i in range(y_test_label.shape[0]):
    x_test = x_test_data.loc[x_test_data['docIdx'] == i + 1,:]
    pred, prob = decision_s(x_test, pi_f, p_f)
    result_s_test_data.append(pred + 1 != y_test_label['class'].iloc[i])

# calculate the error rate of naive Bayes model
error_rate_s_test_data = np.mean(result_s_test_data) * 100
print('Error rate of IDF model on test data is:\n{}%'.format(error_rate_s_test_data))
```

```
Error rate of IDF model on test data is:
94.53697534976683%
```

Conclusion: We have evaluated 3 models on the test data, and it turns out that the smooth model performs a better accurate, whose error rate is 20.3%.

Worksheet 9 Problem 4

1. Handwritten digit recognition using a Gaussian generative model. In class, we mentioned the MNIST data set of handwritten digits. You can obtain it from:

   http://yann.lecun.com/exdb/mnist/index.html

   In this problem, you will build a classifier for this data, by modeling each class as a multivariate (784-dimensional) Gaussian.

(a) Upon downloading the data, you should have two training files (one with images, one with labels) and two test files. Unzip them. In order to load the data into Python you will find the following code helpful:

`http://cseweb.ucsd.edu/~dasgupta/dse210/loader.py`

For instance, to load in the training data, you can use:

`x,y = loadmnist('train-images-idx3-ubyte', 'train-labels-idx1-ubyte')`

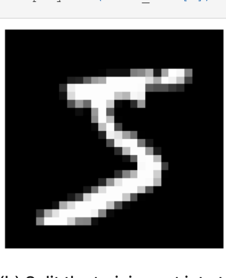
This will set x to a 60000 x 784 array where each row corresponds to an image, and y to a length-60000 array where each entry is a label (0-9). There is also a routine to display images: use displaychar(x[0]) to show the first data point, for instance.

In [27]:
```python
def download(filename, source = 'http://yann.lecun.com/exdb/mnist/index.html'):
    print("Downloading %s" % filename)
    urlretrieve(source + filename, filename)

def load_mnist_images(filename):
    if not os.path.exists(filename):
        download(filename)
    # Read the inputs in Yann LeCun's binary format.
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset = 16)
    data = data.reshape(-1,784)
    return data / np.float32(256)

def load_mnist_labels(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset = 8)
    return data

def displaychar(image):
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.axis('off')
    plt.show()
```

In [28]:
```python
# Load the training data set
train_data = load_mnist_images('train-images-idx3-ubyte.gz')
train_labels = load_mnist_labels('train-labels-idx1-ubyte.gz')

print('Shape of the train data:\n', train_data.shape)
print('\nShape of the train labels:\n', train_labels.shape)
```

```
Shape of the train data:
 (60000, 784)

Shape of the train labels:
 (60000,)
```

In [29]:
```python
# Display the first data point
displaychar(train_data[0])
```



(b) Split the training set into two pieces – a training set of size 50000, and a separate validation set of size 10000. Also load in the test data.

In [30]:
```python
x, y = train_data, train_labels

# randomly select 50000 data from data set as training data set.
random.seed(10)
sel = random.sample(range (0, x.shape[0]), 50000 )
x_train_data, y_train_data = train_data[sel,], train_labels[sel,]

print('Shape of x_train_data:\n', x_train_data.shape)
print('\nShape of y_train_data:\n', y_train_data.shape)
```

```
Shape of x_train_data:
 (50000, 784)

Shape of y_train_data:
 (50000,)
```

In [31]:
```python
# get the remaining data as validation data set.
remain = np.setdiff1d(range(0, x.shape[0]), sel)
x_validation, y_validation = train_data[remain,], train_labels[remain,]

print('Shape of x_validation:\n', x_validation.shape)
print('\nShape of y_validation:\n', y_validation.shape)
```

```
Shape of x_validation:
 (10000, 784)

Shape of y_validation:
 (10000,)
```

In [32]:
```python
# Load the testing data set
test_data = load_mnist_images('t10k-images-idx3-ubyte.gz')
test_labels = load_mnist_labels('t10k-labels-idx1-ubyte.gz')

print('Shape of the test data:\n', test_data.shape)
print('\nShape of the test labels:\n', test_labels.shape)
```

```
Shape of the test data:
 (10000, 784)

Shape of the test labels:
 (10000,)
```

(c) Now fit a Gaussian generative model to the training data of 50000 points.

• Determine the class probabilities.

• Fit a Gaussian to each digit, by finding the mean and the covariance of the corresponding data points.

In [33]:
```python
# create a multivariabte Gaussian model

def MultivariateGaussian(x, y):
    #labels 1,2,...,k
    k = 10

    #number of features
    d = (x.shape)[1]

    mu = np.zeros((k,d))
    sigma = np.zeros((k,d,d))
    pi = np.zeros(k)

    for label in range(10):
        indices = np.where(y == label)
        indices = indices[0]
        mu[label] = np.mean(x[indices,:], axis=0)
        sigma[label] = np.cov(x[indices,:], rowvar=0, bias=1)
        pi[label] = float(len(indices))/float(len(y))
    return mu, sigma, pi
```

In [34]:
```python
# get the mu, sigma, and pi
mu, sigma, pi = MultivariateGaussian(x_train_data, y_train_data)
```

In [35]:
```python
print('Class probabilities:\n', pi)
```

```
Class probabilities:
 [0.0989  0.1129  0.09792 0.10202 0.09642 0.091   0.09882 0.10514 0.0978
 0.09908]
```

In [36]:
```python
print('Mean of the corresponding data points:\n ', mu)
```

```
Mean of the corresponding data points:
  [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

In [37]:
```python
print('Covariance of the corresponding data points:\n ', sigma)
```

```
Covariance of the corresponding data points:
  [[[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 ...

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]]
```

(d) One last step is needed: it is important to smooth the covariance matrices, and the usual way to do this is to add in cI, where c is some constant and I is the identity matrix. What value of c is right? Use the validation set to help you choose. That is, choose the value of c for which the resulting classifier makes the fewest mistakes on the validation set. What value of c did you get?

In [38]:
```python
# classify funtion using Bayes' rule

def decision(x, pi, mu, sigma):
    prob = np.zeros((10, x.shape[0]))

    for i in range (0,10):
        prob[i,:] = pi[i]* multivariate_normal.pdf(x, mean = mu[i,:], cov = sigma[i,:,:])
        #if the sigma matrix cannot be inversed, we need to add a np.eye() to avoid singular matrix error.
        #since we will smooth in step (d), so I didn't use np.eye() here.
        #prob[i,:] = pi[i]* multivariate_normal.pdf(x, mean = mu[i,:], cov = sigma[i,:,:] + np.eye(sigma.sha

    pred = np.argmax(prob, axis = 0)

    return pred, prob
```

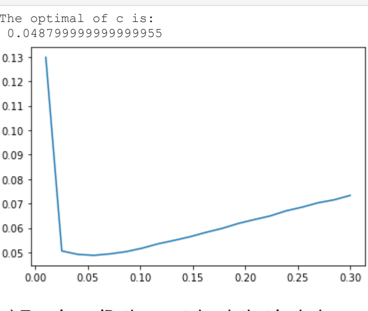
In [39]:
```python
# set c as a constant and I as the identity matix
# randomly give c a number
c = 0.1
iden = np.zeros((10,784,784))

for i in range (0,10) :
    iden[i,:,:] = np.diag([1] * 784)

# smooth the covariance matrix
decision(x_validation[:5,], pi, mu, sigma + c * iden)
```

Out[39]:
```
(array([0, 9, 4, 0, 9]),
 array([[3.52395032e+36, 9.94963706e+02, 4.04319827e-42, 1.73069617e+31,
         2.60846830e+03],
        [2.76205313e-59, 3.89010386e+23, 1.37779283e-87, 1.71922354e-70,
         5.36199288e+13],
        [1.67822220e+09, 2.20521407e+12, 3.34169997e-05, 1.33258778e+04,
         6.56761242e+10],
        [5.52541293e+03, 1.30537050e+18, 3.96725820e-08, 6.44940176e+04,
         1.77372353e+14],
        [2.12330838e-18, 1.03923436e+24, 1.47574376e+20, 2.85214873e-19,
         1.13417968e+20],
        [4.32712370e+03, 5.06260109e+15, 1.08791462e-21, 4.09520025e+06,
         8.53140112e+08],
        [7.30100850e+07, 4.61689670e+05, 1.08491729e-28, 2.60239295e+06,
         3.10196619e-05],
        [1.51033108e-17, 3.29536606e+32, 2.50752743e-12, 1.05837175e-27,
         1.41461342e+35],
        [1.00110917e-06, 1.36214676e+24, 5.55170895e-10, 6.06793379e-15,
         1.18698631e+20],
        [1.17818041e-04, 4.03108896e+38, 1.23801240e-04, 1.00516671e-12,
         1.53564586e+37]]))
```

In [40]:
```python
c_grid = np.linspace(0.01, 0.3, num=20 )
validation_error =[]

for c in c_grid:
    pred = decision(x_validation, pi, mu, sigma + c * iden)[0]
    validation_error.append(1 - np.mean(pred == y_validation))
```

In [41]:
```python
# smaller erorr is desired
plt.plot(c_grid,validation_error)

# find the optimal c value
c_optim = c_grid[np.argmin(validation_error)]

print('The optimal of c is:\n', np.min(validation_error))
```

```
The optimal of c is:
 0.048799999999999955
```



(e) Turn in an iPython notebook that includes:

```
• All your code.

• Error rate on the MNIST test set.

• Out of the misclassified test digits, pick five at random and display them. For each
instance, list the posterior probabilities Pr(y|x) of each of the ten classes.
```

In [42]:
```python
# calculate the error rate on the MNIST test set.
accuracy = np.mean(decision(test_data, pi, mu, sigma + c_optim * iden)[0] == test_labels)
error_rate = (1 - accuracy)*100

print('Error rate on the MNIST test set:\n{}%'.format(error_rate))
```

```
Error rate on the MNIST test set:
4.3799999999999955%
```

In [43]:
```python
pred, prob = decision(test_data,pi, mu, sigma + c_optim * iden)
```

In [44]:
```python
mis = np.where(pred != test_labels)[0]
random.seed(14)

for i in random.sample(range(0,len(mis)),5):
    print('posterior probabilities are:\n', prob[:,mis[i]]/np.sum(prob[:,mis[i]]))
    #print(np.where(np.argmax(prob[:,mis[i]]/np.sum(prob[:,mis[i]]))))
    print('\nmissclassified digits', test_labels[mis[i]])
    print('---')
```

```
posterior probabilities are:
 [1.39773549e-28 3.29542606e-31 9.44407064e-07 8.93100210e-11
 5.10498694e-32 2.15188108e-35 1.76340732e-59 3.71674998e-20
 9.99999056e-01 4.60969151e-25]

missclassified digits 2
---
posterior probabilities are:
 [9.90840610e-001 1.96922947e-134 5.59213076e-006 2.73862179e-015
 9.26492386e-015 8.29549696e-003 2.31647883e-010 2.36242297e-026
 1.17461319e-034 8.58301070e-004]

missclassified digits 9
---
posterior probabilities are:
 [1.33288214e-022 4.78235805e-141 1.39248990e-009 9.99999999e-001
 1.79715520e-067 4.52804023e-026 5.12737030e-081 1.69816726e-043
 1.51046950e-021 3.78681257e-032]

missclassified digits 9
---
posterior probabilities are:
 [7.24531355e-001 8.06604028e-122 2.07758767e-013 2.68522993e-001
 5.10092467e-055 1.94953938e-007 3.58494474e-014 8.39106303e-081
 6.94545720e-003 1.19623255e-040]

missclassified digits 8
---
posterior probabilities are:
 [3.60704806e-23 4.73306638e-08 1.62911845e-22 3.03170718e-25
 1.07755734e-29 4.39101476e-22 9.99999953e-01 5.30893474e-48
 1.17243012e-17 1.22402935e-40]

missclassified digits 1
---
```

Worksheet 10 Problem 2

For this problem, we'll be using the animals with attributes data set. Go to

`http://attributes.kyb.tuebingen.mpg.de`

and, under "Downloads", choose the "base package" (the very first file in the list). Unzip it and look over the various text files.

About the dataset: This is a small dataset that has information on about 50 animals. The animals are listed in classes.txt. For each animal, the information consists of values for 85 features: does the animal have a tail, is it slow, does it have tusks, etc. The details of the features are in the predicates.txt. The full data consists of a 50 x 85 matrix of real values, in predicate-matrix-continuous.txt. There is also a binarized version of this data, in predicate-matrix-binary.txt.

In [45]:
```python
def download(filename, source = 'http://attributes.kyb.tuebingen.mpg.de'):
    print("Downloading %s" % filename)
    urlretrieve(source + filename, filename)

def load_data(filename):
    if not os.path.exists(filename):
        download(filename)
    with bz2.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset = 16)
    return data / np.float32(256)
```

In [46]:
```python
base_package_data = load_data('AwA-base.tar.bz2')
```

(a) Load the real-valued array, and also the animal names, into Python. Run k-means on the data (from sklearn.cluster) and ask for k = 10 clusters. For each cluster, list the animals in it. Does the clustering make sense?

In [47]:
```python
# classes with animal names.
classes_with_animal_names = pd.read_fwf("Animals_with_Attributes/classes.txt", header = None)[1].values
print('Shape of the classes.txt:\n', classes_with_animal_names.shape, '\n')
print('Data of the classes.txt:\n', classes_with_animal_names)
```

```
Shape of the classes.txt:
 (50,)

Data of the classes.txt:
 ['antelope' 'grizzly+bear' 'killer+whale' 'beaver' 'dalmatian'
 'persian+cat' 'horse' 'german+shepherd' 'blue+whale' 'siamese+cat'
 'skunk' 'mole' 'tiger' 'hippopotamus' 'leopard' 'moose' 'spider+monkey'
 'humpback+whale' 'elephant' 'gorilla' 'ox' 'fox' 'sheep' 'seal'
 'chimpanzee' 'hamster' 'squirrel' 'rhinoceros' 'rabbit' 'bat' 'giraffe'
 'wolf' 'chihuahua' 'rat' 'weasel' 'otter' 'buffalo' 'zebra' 'giant+panda'
 'deer' 'bobcat' 'pig' 'lion' 'mouse' 'polar+bear' 'collie' 'walrus'
 'raccoon' 'cow' 'dolphin']
```

In [48]:
```python
# predicates with feature names.
predicates_with_feature_names = pd.read_fwf("Animals_with_Attributes/predicates.txt", header = None)[1].valu
print('Shape of the predicates.txt:\n', predicates_with_feature_names.shape, '\n')
print('Data of the predicates.txt:\n', predicates_with_feature_names)
```

```
Shape of the predicates.txt:
 (85,)

Data of the predicates.txt:
 ['black' 'white' 'blue' 'brown' 'gray' 'orange' 'red' 'yellow' 'patches'
 'spots' 'stripes' 'furry' 'hairless' 'toughskin' 'big' 'small' 'bulbous'
 'lean' 'flippers' 'hands' 'hooves' 'pads' 'paws' 'longleg' 'longneck'
 'tail' 'chewteeth' 'meatteeth' 'buckteeth' 'strainteeth' 'horns' 'claws'
 'tusks' 'smelly' 'flys' 'hops' 'swims' 'tunnels' 'walks' 'fast' 'slow'
 'strong' 'weak' 'muscle' 'bipedal' 'quadrapedal' 'active' 'inactive'
 'nocturnal' 'hibernate' 'agility' 'fish' 'meat' 'plankton' 'vegetation'
 'insects' 'forager' 'grazer' 'hunter' 'scavenger' 'skimmer' 'stalker'
 'newworld' 'oldworld' 'arctic' 'coastal' 'desert' 'bush' 'plains'
 'forest' 'fields' 'jungle' 'mountains' 'ocean' 'ground' 'water' 'tree'
 'cave' 'fierce' 'timid' 'smart' 'group' 'solitary' 'nestspot' 'domestic']
```

In [49]:
```python
# matrix of real values.
```

```python
predicate_matrix = pd.read_fwf("Animals_with_Attributes/predicate-matrix-continuous.txt", header = None).val
print('Shape of the predicate-matrix-continuous.txt:\n', predicate_matrix.shape, '\n')
print('Data of the predicate-matrix-continuous.txt:\n', predicate_matrix)
```

```
Shape of the predicate-matrix-continuous.txt:
 (50, 85)

Data of the predicate-matrix-continuous.txt:
 [[-1.   -1.   -1.   ...  2.35  9.7    8.38]
 [39.25  1.39  0.   ... 58.64 20.14 11.39]
 [83.4  64.79  0.   ... 15.77 13.41 15.42]
 ...
 [63.57 43.1   0.   ... 35.95 28.26  5.  ]
 [55.31 55.46  0.   ...  5.04 18.89 72.99]
 [10.22 21.53 27.73 ...  3.96 14.05 37.98]]
```

In [50]:

```python
# add column names (feature names) for the predicate matrix of the real value.
classes_features = pd.DataFrame(data = predicate_matrix, columns = predicates_with_feature_names)

# add row index for predicate matrix of the real value.
classes_features.index = classes_with_animal_names
print('Shape of classes_features:\n', classes_features.shape, '\n')
print('Data of classes_features:\n', classes_features)
```

```
Shape of classes_features:
 (50, 85)
```

Data of classes_features:

	black	white	blue	brown	gray	orange	red	yellow	\
antelope	-1.00	-1.00	-1.00	-1.00	12.34	0.00	0.00	0.00	
grizzly+bear	39.25	1.39	0.00	74.14	3.75	0.00	0.00	0.00	
killer+whale	83.40	64.79	0.00	0.00	1.25	0.00	0.00	0.00	
beaver	19.38	0.00	0.00	87.81	7.50	0.00	0.00	0.00	
dalmatian	69.58	73.33	0.00	6.39	0.00	0.00	0.00	0.00	
persian+cat	19.38	50.09	29.44	8.98	38.19	0.00	0.00	0.00	
horse	44.90	42.91	4.44	69.41	35.94	0.00	0.00	0.00	
german+shepherd	43.54	15.88	5.00	54.16	26.82	3.12	2.50	0.38	
blue+whale	12.92	4.38	67.08	7.50	25.60	0.00	0.00	0.00	
siamese+cat	56.21	23.51	12.22	32.69	38.13	0.00	0.00	0.00	
skunk	87.99	85.35	0.00	0.00	0.00	0.00	0.00	0.00	
mole	39.05	0.00	0.00	51.33	34.91	0.00	0.00	0.00	
tiger	42.47	30.12	3.75	20.00	2.50	72.91	5.00	17.18	
hippopotamus	4.77	0.00	0.00	18.61	81.49	0.00	0.00	0.00	
leopard	40.88	19.44	0.00	31.33	2.50	20.42	2.50	24.58	
moose	10.24	6.25	0.00	91.20	11.81	0.00	3.57	0.00	
spider+monkey	36.04	6.77	0.00	55.21	34.48	0.00	0.00	0.00	
humpback+whale	24.01	5.92	31.10	8.75	59.43	0.00	0.00	0.00	
elephant	2.50	3.75	0.00	15.23	83.97	0.00	0.00	0.00	
gorilla	63.37	1.79	7.14	45.51	17.01	8.48	3.57	0.00	
ox	43.99	27.68	12.21	50.80	36.31	5.91	3.75	0.00	
fox	0.00	1.56	0.00	51.25	11.17	48.92	40.89	3.75	
sheep	32.36	89.09	0.00	8.41	19.38	0.00	0.00	0.00	
seal	81.96	29.31	1.56	35.00	44.50	0.00	0.00	0.00	
chimpanzee	47.51	9.93	0.00	69.10	13.05	7.14	0.00	2.50	
hamster	41.38	39.71	0.00	62.76	37.38	17.50	2.50	10.00	
squirrel	10.56	13.19	0.00	64.51	72.67	0.00	3.12	0.00	
rhinoceros	9.75	7.50	7.50	19.90	63.96	0.00	0.00	0.00	
rabbit	26.49	64.32	0.00	47.12	39.71	0.00	0.00	0.00	
bat	91.55	1.39	0.00	54.76	27.64	0.00	0.00	0.00	
giraffe	6.11	11.87	0.00	32.21	0.00	24.70	1.39	48.43	
wolf	43.74	23.96	0.00	58.07	53.94	0.00	0.00	0.00	
chihuahua	32.63	10.00	0.00	64.79	22.63	2.50	0.12	0.25	
rat	50.13	33.78	0.00	46.23	40.42	0.00	0.00	0.00	
weasel	30.57	14.11	0.00	62.48	30.34	0.00	0.00	2.50	
otter	46.81	0.00	0.00	44.86	16.25	0.00	0.00	0.00	
buffalo	45.37	0.00	0.00	61.05	9.90	0.00	0.00	0.00	
zebra	85.04	85.04	0.00	0.00	0.00	0.00	0.00	0.00	
giant+panda	76.85	72.33	0.00	5.00	4.38	0.00	0.00	0.00	
deer	0.00	20.34	0.00	75.85	5.92	0.00	0.00	0.00	
bobcat	16.13	9.44	0.00	38.39	2.50	37.08	5.00	22.83	
pig	21.52	25.04	0.00	35.13	26.62	0.00	6.25	3.75	
lion	1.88	1.25	0.00	43.91	0.00	13.12	0.00	34.38	
mouse	18.37	55.35	0.00	32.53	49.72	0.00	0.00	0.00	
polar+bear	10.00	95.62	2.50	3.12	12.50	0.00	0.00	2.50	
collie	10.13	41.37	0.00	47.27	3.75	8.00	0.50	0.00	
walrus	18.84	4.82	0.00	67.59	44.27	0.00	0.00	0.00	
raccoon	63.57	43.10	0.00	17.29	54.51	0.00	0.00	0.00	
cow	55.31	55.46	0.00	58.48	15.50	1.49	0.15	0.00	
dolphin	10.22	21.53	27.73	0.33	60.82	0.00	0.00	0.16	

	patches	spots	...	water	tree	cave	fierce	timid	\
antelope	16.11	9.19	...	0.00	0.00	1.23	10.49	39.24	
grizzly+bear	1.25	0.00	...	7.64	9.79	53.14	61.80	12.50	
killer+whale	68.49	32.69	...	79.49	0.00	0.00	38.27	9.77	
beaver	0.00	7.50	...	65.62	0.00	0.00	3.75	31.88	
dalmatian	37.08	100.00	...	1.25	6.25	0.00	9.38	31.67	
persian+cat	17.93	6.25	...	1.25	2.64	0.00	13.98	43.69	
horse	22.29	15.80	...	2.22	0.00	0.00	15.51	35.39	
german+shepherd	48.78	11.59	...	3.75	0.00	2.50	57.44	10.00	
blue+whale	15.31	23.75	...	76.61	0.00	0.00	7.50	44.58	
siamese+cat	35.83	6.94	...	2.22	10.00	1.11	35.98	28.82	
skunk	6.46	1.25	...	4.03	3.26	10.00	17.29	46.11	
mole	5.62	0.00	...	0.00	10.00	9.72	16.88	29.47	
tiger	8.26	6.08	...	4.06	6.39	20.14	83.81	3.75	
hippopotamus	0.00	0.00	...	46.78	0.00	0.00	19.17	29.58	
leopard	26.11	74.97	...	2.50	25.37	10.56	70.30	0.00	
moose	12.50	12.50	...	1.25	0.00	2.50	8.75	40.42	
spider+monkey	0.00	0.00	...	0.00	76.01	1.25	17.71	23.68	
humpback+whale	18.72	6.88	...	89.95	0.00	0.00	3.75	60.72	
elephant	0.00	1.25	...	1.25	0.00	0.00	20.63	39.36	
gorilla	0.00	0.00	...	1.25	50.29	14.12	46.23	12.13	
ox	19.03	15.62	...	2.50	0.00	1.25	20.16	33.65	
fox	1.56	0.00	...	0.00	1.25	4.06	44.90	5.00	
sheep	0.00	0.00	...	0.00	0.00	0.00	0.00	54.38	
seal	17.19	23.12	...	78.92	0.00	5.00	13.75	35.31	
chimpanzee	0.62	0.00	...	1.25	81.36	10.00	23.10	29.64	
hamster	52.77	8.50	...	0.00	5.00	2.50	8.75	40.69	
squirrel	9.79	2.50	...	1.25	70.99	1.25	7.92	38.16	
rhinoceros	2.95	1.39	...	13.54	0.00	0.00	42.50	13.33	
rabbit	26.04	9.26	...	0.00	0.00	7.27	4.17	61.21	
bat	1.25	1.25	...	1.39	56.97	80.62	32.57	8.98	
giraffe	47.52	77.08	...	0.00	9.88	0.00	1.11	45.36	
wolf	19.92	0.00	...	0.00	0.00	30.49	75.75	0.00	
chihuahua	17.10	11.79	...	1.25	9.72	0.00	33.03	36.73	
rat	10.62	1.25	...	0.00	1.25	20.47	59.44	11.81	
weasel	16.61	8.75	...	5.42	12.50	10.00	59.37	3.75	
otter	0.00	0.00	...	78.75	0.00	0.00	6.25	33.47	
buffalo	2.08	0.00	...	9.38	0.00	0.00	26.99	19.88	
zebra	0.00	0.00	...	0.00	0.00	0.00	0.00	53.25	
giant+panda	84.95	24.17	...	1.25	49.12	14.40	6.25	65.84	
deer	30.34	48.08	...	0.00	0.00	0.00	0.00	67.49	
bobcat	7.50	33.55	...	0.00	24.90	26.20	71.95	0.00	
pig	41.47	21.23	...	3.87	0.00	3.75	25.48	27.85	
lion	0.00	0.00	...	0.00	2.50	13.75	70.47	4.38	
mouse	12.19	6.25	...	0.00	0.00	0.00	1.25	50.00	
polar+bear	0.00	0.00	...	45.90	0.00	17.50	70.35	3.75	
collie	37.00	9.09	...	0.00	0.00	0.00	5.25	43.09	
walrus	11.61	13.12	...	62.30	0.00	0.00	20.00	29.43	
raccoon	29.58	22.87	...	2.08	33.41	7.27	34.46	2.27	
cow	26.67	31.35	...	0.55	0.00	0.32	3.34	47.87	
dolphin	3.30	0.80	...	71.40	0.00	0.00	10.73	21.06	

	smart	group	solitary	nestspot	domestic
antelope	17.57	50.59	2.35	9.70	8.38
grizzly+bear	24.00	3.12	58.64	20.14	11.39
killer+whale	52.03	24.94	15.77	13.41	15.42
beaver	41.88	23.44	31.88	33.44	13.12
dalmatian	53.26	24.44	29.38	11.25	72.71
persian+cat	38.62	6.25	36.60	9.17	72.88
horse	37.28	36.47	16.78	14.62	59.33
german+shepherd	57.53	12.50	35.11	16.53	68.55
blue+whale	39.06	33.12	25.99	10.83	5.00
siamese+cat	52.90	3.33	47.54	17.22	83.55
skunk	12.50	2.50	47.85	18.19	8.89
mole	16.67	0.00	55.31	29.79	13.75
tiger	36.59	23.96	39.36	24.34	5.64
hippopotamus	8.75	20.60	25.14	2.31	0.00
leopard	37.16	11.25	33.84	22.08	6.25
moose	19.11	30.34	31.94	10.14	5.14
spider+monkey	55.00	39.06	9.38	31.67	12.78
humpback+whale	60.31	51.03	19.06	20.34	0.00
elephant	22.81	49.87	6.25	12.29	6.88
gorilla	56.19	54.50	7.50	44.17	18.93
ox	14.68	16.79	32.39	13.12	53.33
fox	79.26	18.75	48.18	27.88	4.38
sheep	10.62	73.69	0.00	7.50	30.68
seal	44.49	49.84	6.25	18.18	47.03
chimpanzee	84.36	74.51	8.75	41.93	36.62
hamster	18.37	16.98	35.00	36.60	71.44
squirrel	18.48	9.17	22.58	21.56	14.76
rhinoceros	10.59	22.85	25.17	3.75	2.50
rabbit	11.47	38.16	18.89	38.89	60.52
bat	32.30	62.25	19.97	34.91	5.56
giraffe	16.95	40.89	9.57	8.69	3.33
wolf	61.48	46.81	41.37	18.40	5.00
chihuahua	29.84	1.25	44.91	9.80	73.55
rat	24.99	18.94	40.87	23.12	16.69
weasel	36.56	13.06	26.26	10.14	3.75
otter	27.22	17.92	30.56	23.19	7.78
buffalo	10.00	52.99	8.80	9.38	2.31
zebra	22.63	80.60	1.25	19.09	9.94
giant+panda	37.30	28.10	55.38	38.19	29.58
deer	40.05	55.46	6.51	31.55	10.27
bobcat	42.89	13.47	59.47	20.28	3.75
pig	23.04	28.07	4.19	17.88	48.95
lion	25.70	56.17	11.88	43.12	3.75
mouse	12.42	23.96	6.88	23.26	27.64
polar+bear	16.25	13.75	48.75	19.38	5.00
collie	42.17	0.62	45.99	18.57	79.11
walrus	23.96	60.41	11.19	33.77	18.75
raccoon	48.68	13.01	35.95	28.26	5.00
cow	13.97	51.57	5.04	18.89	72.99
dolphin	60.38	49.62	3.96	14.05	37.98

[50 rows x 85 columns]

In [51]:

```python
# run k-means on the data, with k = 10.
kmeans = KMeans(n_clusters = 10, init = 'k-means++',  n_init = 10)
kmeans.fit(predicate_matrix, classes_with_animal_names)
```

Out[51]: `KMeans(n_clusters=10)`

In [52]:

```python
# label all animals
print('labels for animals:\n',kmeans.labels_)
```

```
labels for animals:
 [6 0 3 9 4 4 6 4 3 4 1 1 5 8 5 2 7 3 8 7 2 5 2 3 7 1 1 8 1 1 6 5 4 1 1 9 2
 6 7 6 5 2 5 1 0 4 3 1 2 3]
```

In [53]:

```python
# cluster animals to their labels

clustering = {i:[] for i in range(0,10)}

for i,j in enumerate(classes_with_animal_names):
    clustering[kmeans.labels_[i]].append(j)

for i in range(0,10):
    print('label =',i,'\n')
    print('animals:',clustering[i],'\n')
```

```
label = 0

animals: ['grizzly+bear', 'polar+bear']

label = 1

animals: ['skunk', 'mole', 'hamster', 'squirrel', 'rabbit', 'bat', 'rat', 'weasel', 'mouse', 'raccoon']

label = 2

animals: ['moose', 'ox', 'sheep', 'buffalo', 'pig', 'cow']

label = 3

animals: ['killer+whale', 'blue+whale', 'humpback+whale', 'seal', 'walrus', 'dolphin']

label = 4

animals: ['dalmatian', 'persian+cat', 'german+shepherd', 'siamese+cat', 'chihuahua', 'collie']

label = 5

animals: ['tiger', 'leopard', 'fox', 'wolf', 'bobcat', 'lion']

label = 6

animals: ['antelope', 'horse', 'giraffe', 'zebra', 'deer']

label = 7

animals: ['spider+monkey', 'gorilla', 'chimpanzee', 'giant+panda']

label = 8

animals: ['hippopotamus', 'elephant', 'rhinoceros']

label = 9

animals: ['beaver', 'otter']
```

Conclusion for (a):

Does the clustering make sense?

--- Yes, it makes some sense.

--- From the clustering above, we can see that pets in label 7 are grouped close together as well as other labels.

--- However, it is not perfect. we need to figure out optimal k or find other algorithms to cluster these animals better.

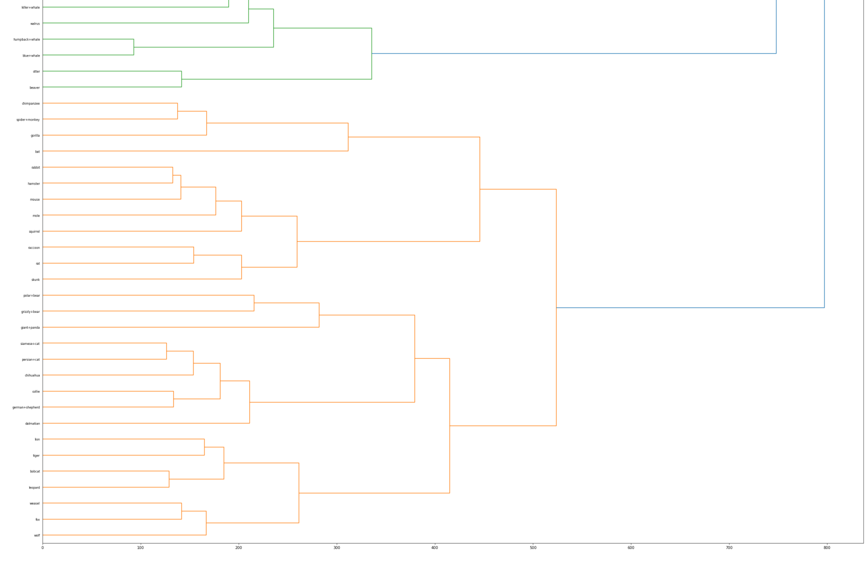
(b) Now hierarchically cluster this data, using scipy.cluster.hierarchy.linkage. Choose Ward's method, and plot the resulting tree using the dendrogram method, setting the orientation parameter to 'right' and labeling each leaf with the corresponding animal name. You will run into a problem: the plot is too cramped because the default figure size is so small. To make it larger, preface your code with the following:

```
from pylab import rcParams
rcParams['figure.figsize'] = 5, 10
```

(or try a different size if this doesn't seem quite right). Does the hierarchical clustering seem sensible to you?

In [55]:

```python
# use scipy.cluster.hierarchy.linkage to hierarchically cluster this data.
hierarchically_cluster_link = linkage(predicate_matrix, method = 'ward');

# use dendrogram method to plot the resulting tree.
dendrogram(hierarchically_cluster_link, orientation = "right", labels = classes_with_animal_names)

# resize the figure
rcParams['figure.figsize'] = [40,40]
```



Conclusion for (b):

Does the hierarchical clustering seem sensible to you?

--- Yes, it seems sensible to me.

--- For choosing the linkage, Ward's method is the sensible default. It groups based on reducing the sum of squared distances of each observation from the average observation in a cluster.

(c) Turn in an iPython notebook with a transcript of all this experimentation.

See above for all this experimentation