

Deep Learning

Assignment 1

The objective of this assignment is to learn about simple data curation practices, and familiarize you with some of the data we'll be reusing later.

This notebook uses the [notMNIST](#) dataset to be used with python experiments. This dataset is designed to look like the classic [MNIST](#) dataset, while looking a little more like real data: it's a harder task, and the data is a lot less 'clean' than MNIST.

In [2]:

```
# These are all the modules we'll be using later. Make sure you can import them
# before proceeding further.
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
import os
import sys
import tarfile
from IPython.display import display, Image
from scipy import ndimage
from sklearn.linear_model import LogisticRegression
from six.moves.urllib.request import urlretrieve
from six.moves import cPickle as pickle

# Config the matplotlib backend as plotting inline in IPython
%matplotlib inline
```

First, we'll download the dataset to our local machine. The data consists of characters rendered in a variety of fonts on a 28x28 image. The labels are limited to 'A' through 'J' (10 classes). The training set has about 500k and the testset 19000 labelled examples. Given these sizes, it should be possible to train models quickly on any machine.

In [3]:

```
url = 'http://commondatastorage.googleapis.com/books1000/'
last_percent_reported = None

def download_progress_hook(count, blockSize, totalSize):
    """A hook to report the progress of a download. This is mostly intended for
users with
slow internet connections. Reports every 1% change in download progress.
    """
    global last_percent_reported
    percent = int(count * blockSize * 100 / totalSize)

    if last_percent_reported != percent:
        if percent % 5 == 0:
            sys.stdout.write("%s%%" % percent)
            sys.stdout.flush()
        else:
            sys.stdout.write(".")
            sys.stdout.flush()
        last_percent_reported = percent
```

```
def maybe_download(filename, expected_bytes, force=False):
    """Download a file if not present, and make sure it's the right size."""
    if force or not os.path.exists(filename):
        print('Attempting to download:', filename)
        filename, _ = urlretrieve(url + filename, filename,
            reporthook=download_progress_hook)
        print('\nDownload Complete!')
        statinfo = os.stat(filename)
        if statinfo.st_size == expected_bytes:
            print('Found and verified', filename)
        else:
            raise Exception(
                'Failed to verify ' + filename + '. Can you get to it with a browser?')
    return filename
```

```
train_filename = maybe_download('notMNIST_large.tar.gz', 247336696)
test_filename = maybe_download('notMNIST_small.tar.gz', 8458043)
```

```
Found and verified notMNIST_large.tar.gz
```

```
Found and verified notMNIST_small.tar.gz
```

Extract the dataset from the compressed .tar.gz file. This should give you a set of directories, labelled A through J.

In [4]:

```
num_classes = 10
np.random.seed(133)
```

```
def maybe_extract(filename, force=False):
    root = os.path.splitext(os.path.splitext(filename)[0])[0] # remove .tar.gz
    if os.path.isdir(root) and not force:
        # You may override by setting force=True.
        print('%s already present - Skipping extraction of %s.' % (root, filename))
    else:
        print('Extracting data for %s. This may take a while. Please wait.' % root)
        tar = tarfile.open(filename)
        sys.stdout.flush()
        tar.extractall()
        tar.close()
        data_folders = [
            os.path.join(root, d) for d in sorted(os.listdir(root))
            if os.path.isdir(os.path.join(root, d))]
        if len(data_folders) != num_classes:
            raise Exception(
                'Expected %d folders, one per class. Found %d instead.' % (
                    num_classes, len(data_folders)))
        print(data_folders)
    return data_folders
```

```
train_folders = maybe_extract(train_filename)
```

```
test_folders = maybe_extract(test_filename)
```

```
notMNIST_large already present - Skipping extraction of notMNIST_large.tar.gz.
```


```
['notMNIST_large/A', 'notMNIST_large/B', 'notMNIST_large/C', 'notMNIST_large/D',
'notMNIST_large/E', 'notMNIST_large/F', 'notMNIST_large/G', 'notMNIST_large/H',
'notMNIST_large/I', 'notMNIST_large/J']
notMNIST_small already present - Skipping extraction of notMNIST_small.tar.gz.
['notMNIST_small/A', 'notMNIST_small/B', 'notMNIST_small/C', 'notMNIST_small/D',
'notMNIST_small/E', 'notMNIST_small/F', 'notMNIST_small/G', 'notMNIST_small/H',
'notMNIST_small/I', 'notMNIST_small/J']
```

Problem 1

Let's take a peek at some of the data to make sure it looks sensible. Each exemplar should be an image of a character A through J rendered in a different font. Display a sample of the images that we just downloaded. Hint: you can use the package IPython.display.

In [5]:

```
#display single image as requested
Image(train_folders[2] + '/' + os.listdir(train_folders[3])[0])
```

Out[5]: 

Now let's load the data in a more manageable format. Since, depending on your computer setup you might not be able to fit it all in memory, we'll load each class into a separate dataset, store them on disk and curate them independently. Later we'll merge them into a single dataset of manageable size.

We'll convert the entire dataset into a 3D array (image index, x, y) of floating point values, normalized to have approximately zero mean and standard deviation ~0.5 to make training easier down the road.

A few images might not be readable, we'll just skip them.

In [6]:

```
image_size = 28 # Pixel width and height.
pixel_depth = 255.0 # Number of levels per pixel.

def load_letter(folder, min_num_images):
    """Load the data for a single letter label."""
    image_files = os.listdir(folder)
    dataset = np.ndarray(shape=(len(image_files), image_size, image_size),
                        dtype=np.float32)

    print(folder)
    num_images = 0
    for image in image_files:
        image_file = os.path.join(folder, image)
        try:
            image_data = (ndimage.imread(image_file).astype(float) -
                          pixel_depth / 2) / pixel_depth
            if image_data.shape != (image_size, image_size):
                raise Exception('Unexpected image shape: %s' % str(image_data.shape))
            dataset[num_images, :, :] = image_data
            num_images = num_images + 1
        except IOError as e:
            print('Could not read:', image_file, ':', e, '- it\'s ok, skipping.')

    dataset = dataset[0:num_images, :, :]
```

```

if num_images < min_num_images:
    raise Exception('Many fewer images than expected: %d < %d' %
                    (num_images, min_num_images))

print('Full dataset tensor:', dataset.shape)
print('Mean:', np.mean(dataset))
print('Standard deviation:', np.std(dataset))
return dataset

def maybe_pickle(data_folders, min_num_images_per_class, force=False):
    dataset_names = []
    for folder in data_folders:
        set_filename = folder + '.pickle'
        dataset_names.append(set_filename)
        if os.path.exists(set_filename) and not force:
            # You may override by setting force=True.
            print('%s already present - Skipping pickling.' % set_filename)
        else:
            print('Pickling %s.' % set_filename)
            dataset = load_letter(folder, min_num_images_per_class)
            try:
                with open(set_filename, 'wb') as f:
                    pickle.dump(dataset, f, pickle.HIGHEST_PROTOCOL)
            except Exception as e:
                print('Unable to save data to', set_filename, ':', e)

    return dataset_names

train_datasets = maybe_pickle(train_folders, 45000)
test_datasets = maybe_pickle(test_folders, 1800)
print('finished')

notMNIST_large/A.pickle already present - Skipping pickling.
notMNIST_large/B.pickle already present - Skipping pickling.

```

Now let's load the data in a more manageable format. Since, depending on your computer setup you might not be able to fit it all in memory, we'll load each class into a separate dataset, store them on disk and curate them independently. Later we'll merge them into a single dataset of manageable size.

We'll convert the entire dataset into a 3D array (image index, x, y) of floating point values, normalized to have approximately zero mean and standard deviation ~0.5 to make training easier down the road.

A few images might not be readable, we'll just skip them.

In [6]:

```

image_size = 28 # Pixel width and height.
pixel_depth = 255.0 # Number of levels per pixel.

def load_letter(folder, min_num_images):
    """Load the data for a single letter label."""
    image_files = os.listdir(folder)
    dataset = np.ndarray(shape=(len(image_files), image_size, image_size),
                        dtype=np.float32)

    print(folder)

```

```

num_images = 0
for image in image_files:
    image_file = os.path.join(folder, image)
    try:
        image_data = (ndimage.imread(image_file).astype(float) -
                       pixel_depth / 2) / pixel_depth
        if image_data.shape != (image_size, image_size):
            raise Exception('Unexpected image shape: %s' % str(image_data.shape))
        dataset[num_images, :, :] = image_data
        num_images = num_images + 1
    except IOError as e:
        print('Could not read:', image_file, ':', e, '- it\'s ok, skipping.')

dataset = dataset[0:num_images, :, :]
if num_images < min_num_images:
    raise Exception('Many fewer images than expected: %d < %d' %
                    (num_images, min_num_images))

print('Full dataset tensor:', dataset.shape)
print('Mean:', np.mean(dataset))
print('Standard deviation:', np.std(dataset))
return dataset

def maybe_pickle(data_folders, min_num_images_per_class, force=False):
    dataset_names = []
    for folder in data_folders:
        set_filename = folder + '.pickle'
        dataset_names.append(set_filename)
        if os.path.exists(set_filename) and not force:
            # You may override by setting force=True.
            print('%s already present - Skipping pickling.' % set_filename)
        else:
            print('Pickling %s.' % set_filename)
            dataset = load_letter(folder, min_num_images_per_class)
            try:
                with open(set_filename, 'wb') as f:
                    pickle.dump(dataset, f, pickle.HIGHEST_PROTOCOL)
            except Exception as e:
                print('Unable to save data to', set_filename, ':', e)
    return dataset_names

train_datasets = maybe_pickle(train_folders, 45000)
test_datasets = maybe_pickle(test_folders, 1800)
print('finished')

notMNIST_large/A.pickle already present - Skipping pickling.
notMNIST_large/B.pickle already present - Skipping pickling.
.....
notMNIST_small/J.pickle already present - Skipping pickling.
finished

```

Problem 2

Let's verify that the data still looks good. Displaying a sample of the labels and images from the ndarray.
Hint: you can use matplotlib.pyplot.

In [7]:

```
#now going to show some labels
%matplotlib inline
print(train_datasets)
with open(train_datasets[0], 'rb') as f:
    letter_set = pickle.load(f)
plt.imshow(letter_set[0])
plt.title("Char a")

['notMNIST_large/A.pickle', 'notMNIST_large/B.pickle',
 'notMNIST_large/C.pickle', 'notMNIST_large/D.pickle', 'notMNIST_large/E.pickle',
 'notMNIST_large/F.pickle', 'notMNIST_large/G.pickle', 'notMNIST_large/H.pickle',
 'notMNIST_large/I.pickle', 'notMNIST_large/J.pickle']
```

out[7]:

<matplotlib.text.Text at 0x7effd4abff10>



Problem 3

Another check: we expect the data to be balanced across classes. Verify that.

In [8]:

```
#function will show name of letter based on label
def letter(i):
    return 'ABCDEFGHIJ'[i]
```

In [9]:

```
#time to check how many records in each label (letter)
for i in range(0,10):
    with open(train_datasets[i], 'rb') as f:
        letter_set = pickle.load(f)
    print('Size of data set for class ' + letter(i) + ': ' +
          str(len(letter_set)))
```

```
Size of data set for class A: 52909
Size of data set for class B: 52911
Size of data set for class C: 52912
Size of data set for class D: 52911
Size of data set for class E: 52912
Size of data set for class F: 52912
Size of data set for class G: 52912
Size of data set for class H: 52912
Size of data set for class I: 52912
Size of data set for class J: 52911
```

Merge and prune the training data as needed. Depending on your computer setup, you might not be able to fit it all in memory, and you can tune `train_size` as needed. The labels will be stored into a separate array of integers 0 through 9.

Also create a validation dataset for hyperparameter tuning.

In [10]:

```
def make_arrays(nb_rows, img_size):
    if nb_rows:
        dataset = np.ndarray((nb_rows, img_size, img_size), dtype=np.float32)
        labels = np.ndarray(nb_rows, dtype=np.int32)
    else:
        dataset, labels = None, None
    return dataset, labels

def merge_datasets(pickle_files, train_size, valid_size=0):
    num_classes = len(pickle_files)
    valid_dataset, valid_labels = make_arrays(valid_size, image_size)
    train_dataset, train_labels = make_arrays(train_size, image_size)
    vsize_per_class = valid_size // num_classes
    tsize_per_class = train_size // num_classes

    start_v, start_t = 0, 0
    end_v, end_t = vsize_per_class, tsize_per_class
    end_l = vsize_per_class + tsize_per_class
    for label, pickle_file in enumerate(pickle_files):
        try:
            with open(pickle_file, 'rb') as f:
                letter_set = pickle.load(f)
                # let's shuffle the letters to have random validation and training set
                np.random.shuffle(letter_set)
                if valid_dataset is not None:
                    valid_letter = letter_set[:vsize_per_class, :, :]
                    valid_dataset[start_v:end_v, :, :] = valid_letter
                    valid_labels[start_v:end_v] = label
                    start_v += vsize_per_class
                    end_v += vsize_per_class

                    train_letter = letter_set[vsize_per_class:end_l, :, :]
                    train_dataset[start_t:end_t, :, :] = train_letter
                    train_labels[start_t:end_t] = label
                    start_t += tsize_per_class
                    end_t += tsize_per_class
        except Exception as e:
            print('Unable to process data from', pickle_file, ':', e)
            raise

    return valid_dataset, valid_labels, train_dataset, train_labels

train_size = 200000
valid_size = 10000
test_size = 10000

valid_dataset, valid_labels, train_dataset, train_labels = merge_datasets(
    train_datasets, train_size, valid_size)
_, _, test_dataset, test_labels = merge_datasets(test_datasets, test_size)
```

```
print('Training:', train_dataset.shape, train_labels.shape)
print('Validation:', valid_dataset.shape, valid_labels.shape)
print('Testing:', test_dataset.shape, test_labels.shape)
```

```
Training: (200000, 28, 28) (200000,)
Validation: (10000, 28, 28) (10000,)
Testing: (10000, 28, 28) (10000,)
```

Next, we'll randomize the data. It's important to have the labels well shuffled for the training and test distributions to match.

In [11]:

```
def randomize(dataset, labels):
    permutation = np.random.permutation(labels.shape[0])
    shuffled_dataset = dataset[permutation,:,:]
    shuffled_labels = labels[permutation]
    return shuffled_dataset, shuffled_labels
train_dataset, train_labels = randomize(train_dataset, train_labels)
test_dataset, test_labels = randomize(test_dataset, test_labels)
valid_dataset, valid_labels = randomize(valid_dataset, valid_labels)
```

Problem 4

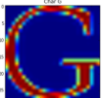
Convince yourself that the data is still good after shuffling!

In [12]:

```
#show some random letter
%matplotlib inline
sample_idx = np.random.randint(0, len(train_dataset))
plt.imshow(train_dataset[sample_idx])
plt.title("Char " + letter(train_labels[sample_idx]))
#for i in range(0,10):
#     print('Size of data set for class ' + letter(i) + ': ' +
str(len(train_dataset)))
```

Out[12]:

```
<matplotlib.text.Text at 0x7effd49b8c50>
```



Finally, let's save the data for later reuse:

In [13]:

```
pickle_file = 'notMNIST.pickle'
try:
    f = open(pickle_file, 'wb')
    save = {
        'train_dataset': train_dataset,
        'train_labels': train_labels,
        'valid_dataset': valid_dataset,
        'valid_labels': valid_labels,
        'test_dataset': test_dataset,
        'test_labels': test_labels, }
```



```

    pickle.dump(save, f, pickle.HIGHEST_PROTOCOL)
    f.close()
except Exception as e:
    print('Unable to save data to', pickle_file, ':', e)
    raise

```

In [14]:

```

statinfo = os.stat(pickle_file)
print('Compressed pickle size:', statinfo.st_size)

Compressed pickle size: 690800441

```

Problem 5

By construction, this dataset might contain a lot of overlapping samples, including training data that's also contained in the validation and test set! Overlap between training and test can skew the results if you expect to use your model in an environment where there is never an overlap, but are actually ok if you expect to see training samples recur when you use it. Measure how much overlap there is between training, validation and test samples.

Optional questions:

- What about near duplicates between datasets? (images that are almost identical)
- Create a sanitized validation and test set, and compare your accuracy on those in subsequent assignments.

In [16]:

```

#lets use md5 hasing to measure the overlap between something
from hashlib import md5
#prepare image hashes
%time set_valid_dataset = set([ md5(x).hexdigest() for x in valid_dataset])
%time set_test_dataset = set([ md5(x).hexdigest() for x in test_dataset])
%time set_train_dataset = set([ md5(x).hexdigest() for x in train_dataset])

#measure overlaps and print them
overlap_test_valid = set_test_dataset - set_valid_dataset
print('overlap test valid: ' + str(len(overlap_test_valid)))

overlap_train_valid = set_train_dataset - set_valid_dataset
print ('overlap train valid: ' + str(len(overlap_train_valid)))

overlap_train_test = set_train_dataset - set_test_dataset
print ('overlap train test: ' + str(len(overlap_train_test)))

CPU times: user 116 ms, sys: 0 ns, total: 116 ms
Wall time: 135 ms
CPU times: user 112 ms, sys: 0 ns, total: 112 ms
Wall time: 113 ms
CPU times: user 2.26 s, sys: 0 ns, total: 2.26 s
Wall time: 2.26 s
overlap test valid: 9710
overlap train valid: 186240
overlap train test: 186083

```

In [18]:

```

#here is how we build sanized data
import matplotlib.pyplot as plt
from six.moves import cPickle as pickle
from hashlib import md5
with
open('/home/maxkhk/Documents/Udacity/DeepLearningCourse/SourceCode/tensorflow/e
xamples/udacity/notMNIST.pickle', 'rb') as f:
    datasets = pickle.load(f)
import numpy as np

#we use it later
def letter(i):
    return 'ABCDEFGHIJ'[i]

#merge all separate datasets from previous problems
all_labels = np.concatenate((datasets['train_labels'],
                                datasets['valid_labels'],
                                datasets['test_labels']))

all_data = np.concatenate((datasets['train_dataset'],
                                datasets['valid_dataset'],
                                datasets['test_dataset']))

#we will store unique values here. Notice - the order is preserved
sanitized_data = []
sanitized_labels = []

#lets illimnate dupes

#store of existing hashes
hashes = set()
#go over each image
for i in range(0, len(all_labels)):
    #check if its hash is in list hashes
    if not md5(all_data[i]).hexdigest() in hashes:
        #record its hash and copy to new dataset
        sanitized_data.append(all_data[i])
        sanitized_labels.append(all_labels[i])
        hashes.add(md5(all_data[i]).hexdigest())

#now change the lists to numpy ndarrays with stack command (it still preserves
original order)
sanitized_data = np.stack(sanitized_data)
sanitized_labels = np.stack(sanitized_labels)
#shuffle to ensure that original not-sanitized data and sanitized data is in
different order
def randomize(dataset, labels):
    permutation = np.random.permutation(labels.shape[0])
    shuffled_dataset = dataset[permutation, :, :]
    shuffled_labels = labels[permutation]

```

```

    return shuffled_dataset, shuffled_labels
sanitized_data, sanitized_labels = randomize(sanitized_data, sanitized_labels)

#checking that data is still fine
%matplotlib inline
sample_idx = np.random.randint(0, len(sanitized_labels))
plt.imshow(sanitized_data[sample_idx])
plt.title("char " + letter(sanitized_labels[sample_idx]))

#now split this all data to validation, train, test datasets as in prior problems
valid_dataset = sanitized_data[:10000]
valid_labels = sanitized_labels[:10000]
test_dataset = sanitized_data[10000:20000]
test_labels = sanitized_labels[10000:20000]
train_dataset = sanitized_data[20000:]
train_labels = sanitized_labels[20000:]

#let's dump it somewhere
pickle_file =
'/home/maxkhk/Documents/Udacity/DeepLearningCourse/SourceCode/tensorflow/examples/udacity/notMNIST_Sanitized.pickle'

try:
    f = open(pickle_file, 'wb')
    save = {
        'train_dataset': train_dataset,
        'train_labels': train_labels,
        'valid_dataset': valid_dataset,
        'valid_labels': valid_labels,
        'test_dataset': test_dataset,
        'test_labels': test_labels,
    }
    pickle.dump(save, f, pickle.HIGHEST_PROTOCOL)
    f.close()
except Exception as e:
    print('Unable to save data to', pickle_file, ':', e)
    raise

```



Problem 6

Let's get an idea of what an off-the-shelf classifier can give you on this data. It's always good to check that there is something to learn, and that it's a problem that is not so trivial that a canned solution solves it.

Train a simple model on this data using 50, 100, 1000 and 5000 training samples. Hint: you can use the LogisticRegression model from `sklearn.linear_model`.

Optional question: train an off-the-shelf model on all the data!

In [2]:

```
from six.moves import cPickle as pickle
```

```
from sklearn.linear_model import LogisticRegressionCV
#load datasets
with
open('/home/maxkhk/Documents/Udacity/DeepLearningCourse/SourceCode/tensorflow/e
xamples/udacity/notMNIST_sanitized.pickle', 'rb') as f:
    datasets_sanitized = pickle.load(f)
with
open('/home/maxkhk/Documents/Udacity/DeepLearningCourse/SourceCode/tensorflow/e
xamples/udacity/notMNIST.pickle', 'rb') as f:
    datasets = pickle.load(f)

#here is our regressor which we will reuse
logregCV = LogisticRegressionCV()

#we first have to flatten our matrices with data to train logreg on them
flat_train_dataset_50 = [x.flatten() for x in datasets['train_dataset'][:50]]
flat_train_dataset_100 = [x.flatten() for x in datasets['train_dataset'][:100]]
flat_train_dataset_1000 = [x.flatten() for x in
datasets['train_dataset'][:1000]]
flat_train_dataset_5000 = [x.flatten() for x in
datasets['train_dataset'][:5000]]
#flat_train_dataset = [x.flatten() for x in datasets['train_dataset']]

flat_train_dataset_50_san = [x.flatten() for x in
datasets_sanitized['train_dataset'][:50]]
flat_train_dataset_100_san = [x.flatten() for x in
datasets_sanitized['train_dataset'][:100]]
flat_train_dataset_1000_san = [x.flatten() for x in
datasets_sanitized['train_dataset'][:1000]]
flat_train_dataset_5000_san = [x.flatten() for x in
datasets_sanitized['train_dataset'][:5000]]
#flat_train_dataset_san = [x.flatten() for x in
datasets_sanitized['train_dataset']]

#now we are going train our logreg on all these data variants
model_50 = logregCV.fit(flat_train_dataset_50, datasets['train_labels'][:50])
model_100 = logregCV.fit(flat_train_dataset_100,
datasets['train_labels'][:100])
model_1000 = logregCV.fit(flat_train_dataset_1000,
datasets['train_labels'][:1000])
model_5000 = logregCV.fit(flat_train_dataset_5000,
datasets['train_labels'][:5000])

model_50_sanitized = logregCV.fit(flat_train_dataset_50_san,
datasets_sanitized['train_labels'][:50])
model_100_sanitized = logregCV.fit(flat_train_dataset_100_san,
datasets_sanitized['train_labels'][:100])
model_1000_sanitized = logregCV.fit(flat_train_dataset_1000_san,
datasets_sanitized['train_labels'][:1000])
model_5000_sanitized = logregCV.fit(flat_train_dataset_5000_san,
datasets_sanitized['train_labels'][:5000])
```

```

#calc the scores and compare vs sanitized
from sklearn.cross_validation import cross_val_score

score_50 = cross_val_score(model_50, flat_train_dataset_50,
datasets['train_labels'][:50]).mean()
score_100 = cross_val_score(model_100, flat_train_dataset_100,
datasets['train_labels'][:100]).mean()
score_1000 = cross_val_score(model_1000, flat_train_dataset_1000,
datasets['train_labels'][:1000]).mean()
score_5000 = cross_val_score(model_5000, flat_train_dataset_5000,
datasets['train_labels'][:5000]).mean()

score_50_san = cross_val_score(model_50_sanitized, flat_train_dataset_50_san,
datasets_sanitized['train_labels'][:50]).mean()
score_100_san = cross_val_score(model_100_sanitized,
flat_train_dataset_100_san, datasets_sanitized['train_labels'][:100]).mean()
score_1000_san = cross_val_score(model_1000_sanitized,
flat_train_dataset_1000_san, datasets_sanitized['train_labels'][:1000]).mean()
score_5000_san = cross_val_score(model_5000_sanitized,
flat_train_dataset_5000_san, datasets_sanitized['train_labels'][:5000]).mean()

print('in sample cross validation scores ')
print(score_50)
print(score_50_san)
print(score_100)
print(score_100_san)
print(score_1000)
print(score_1000_san)
print(score_5000)
print(score_5000_san)

```

```

/home/maxkhk/anaconda/lib/python2.7/site-
packages/sklearn/cross_validation.py:417: Warning: The least populated class in
y has only 2 members, which is too few. The minimum number of labels for any
class cannot be less than n_folds=3.
  % (min_labels, self.n_folds)), Warning)
/home/maxkhk/anaconda/lib/python2.7/site-
packages/sklearn/cross_validation.py:417: Warning: The least populated class in
y has only 1 members, which is too few. The minimum number of labels for any
class cannot be less than n_folds=3.
  % (min_labels, self.n_folds)), Warning)
in sample cross validation scores
0.451348960637
0.40467571644
0.603122275501
0.637363834423
0.793045621829
0.792980048456
0.80599758878
0.810790242749

```