Guidelines – Read Carefully! You are provided a zip file containing a single folder named xyz007 containing the following folders and files.

xyz007
xyz007/digitdata
xyz007/digitdata/trainingimages
xyz007/digitdata/traininglabels
xyz007/digitdata/validationimages
xyz007/digitdata/validationlabels
xyz007/mp440.py
xyz007/main.py

You should first rename the folder name xyz007 to be your NETID. If you are working in a team, the folder should be named with both group members' NETIDs, in the format of NETID1_NETID2. The folder name letters can be either upper or lower case. When you are ready to submit, remove any extra files (e.g., some python interpreters will create .pyc files) that are not required and zip the entire folder. The zip file should also be named with your NETID as NETID.zip (or NETID1_NETID2.zip for groups with two members). You may create additional python files though it should not be necessary; all your code can bit fit into `mp440.py`. This MP also requires the submission of a report in PDF format. Please name your PDF file as report.pdf and put it under the main submission folder, i.e.,

NETID/report.pdf or NETID1_NETID2/report.pdf

You are required to write your program adhering to Python 2.7 standards. In particular, DO NOT use Python 3.x. Beside the default libraries supplied in the standard Python distribution, you may use ONLY numpy and matplotlib libraries (i.e., among the extra libraries installed after the standard python libraries).

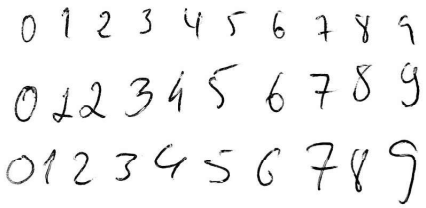
As mentioned in class, you may form groups of up to two students. Only a single student should submit the solution.

Problem 1 [100 points]. Naive Bayes classifier for recognizing hand written digits.

Acknowledgement: This project is based on one created by Dan Klein and John DeNero that was given as part of the programming assignments of Berkeley's CS188 course (`http://inst.eecs.berkeley.edu/~cs188/sp11/projects/classification/classification.html`). It is a simplified version that only asks you to work on the Naive Bayes classifier for hand written digits.

Description: In this project, you will design a naive Bayes classifier, mainly focusing on the feature design. You will test your classifiers on a set of scanned handwritten digit images. Even with simple features, your classifiers will be able to do reasonable well on these tasks given enough training data. Optical character recognition (OCR) is the task of extracting text from image sources. The data set on which you will run your classifiers is a collection of handwritten numerical digits (0-9). This is a very commercially useful technology, similar to the technique used by the US post office to route mail by zip codes. There are systems that can perform with over 99% classification accuracy (see LeNet-5 for an example system in action).

The digit data files (e.g., `xyz007/digitdata/trainingimages`) contain preprocessed digits stored as blobs of 28×28 ASC-II texts (using only ' ', '+', '#' characters). You may open the data files to get an idea of the raw data. For each digit data file, there is also an accompanying file containing the labels for the given data. In `main.py`, some functions are already provided for loading the data.



Which Digit?

An example of the data for the digit 3.

a) Implement a functioning Naive Bayes classifier (50 points). The first task is the implementation of a functioning Naive Bayes classifier.

a.1 (5 points) To start, you are to first implement a basic feature extractor

`extract_basic_features(`*digit_data*`,` *width*`,` *height*`)`,

which processes a single digit image data (as list of list, see `main.py` for more details) of given width and height and return a list of features extracted from the given image. For this function, the feature are the pixels in the image. More specifically, you are to create one feature for each pixel located at a given coordinate (*row*, *column*). You can retrieve the pixel value at the given coordinate via accessing *digit_data*[*row*][*column*]. Each feature will be a binary feature, i.e., if the pixel is in the background (' ' in the data file), then the feature will be set to False. Otherwise the feature should be set to True. You should return a binary list of length $row \times column$, i.e., the list is an unfolding of the 2D array row by row.

a.2 (5 points) Given the large number of features, if we use the standard multiplicative representation of joint probability, i.e.,

$$P(y \mid x_1, \ldots, x_n) \propto \prod_i P(x_i \mid y) P(y), \tag{1}$$

there could be some drawbacks. What are the drawbacks?

a.3 (10 points) We instead opt to take the natural logrithm on both sides of (1) and use

$$\log P(y \mid x_1, \ldots, x_n) \propto \log P(y) + \sum_i \log P(x_i \mid y) \tag{2}$$

for our computation. To compute this, you need to implement the function

`compute_statistics(`*data*`,` *label*`,` *width*`,` *height*`,` *feature_extractor*`,` *percentage*`)`.

In the function, *data* is the list of raw digit images and *label* is the list of accompanying labels. *width* and *height* are the dimensions of the digit figures. *feature_extractor* is the function that will be used to extract features, which can be *extract_basic_features*. The argument *percentage* specifies that only the first *percentage*% of the training data set should be used. In the function `compute_statistics`, you should first extract the features (on only the specified *percentage*% of the training data) and then compute the prior and conditional probabilities. You will need to create some global variable to hold the computed statistics. Because some $P(x_i \mid y)$, you cannot take log of these easily. In such cases you may use Laplace smoothing to avoid doing $\log 0$ (see the Berkeley link provided above on how to do this). Use the same smoothing parameter k for all the features.

a.4 (10 points) With a trained model, you can now make predictions using it. To do this, you are to implement two functions,

`classify(`*data*`,` *width*`,` *height*`,` *feature_extractor*`)`,

whose arguments have the same meaning as the similar-named arguments in `compute_statistics`, which in turn calls the function

`compute_class(`*features*`)`

that computes the class for the feature vector (list) for a single digit image.

a.5 (10 points) Evaluation the effectiveness of the basic Naive Bayes classifier. For this task, you should train your classifier using 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and finally 100% of the training data (i.e., from 500 examples to 5000 examples). For each trained model, you should then evaluate the accuracy of your classifier on the validation data (i.e., the data loaded from the files named "validation*"). To compute the accuracy, simply divide the number of correctly predicted label by the total number of validation examples. Summarize the result in your report.

a.6 (10 points) What is the k that you decide to use in the end? Why? How does your classifier do? Do you think the result is reasonably good? Why or why not?

b) Enhancing features (50 points).

b.1 (30 points) You are now to come up with three new sets of features (you are not limited to binary features). Each feature set should have some rationale for being included, which should be clearly stated in your report. Note that after you come up with a feature set design, you cannot simply apply it to a different region of the image data and call that a new feature set. The new features should be implemented in the function

`extract_advanced_features(`*digit_data*`,` *width*`,` *height*`)`.

Make sure that, if you need to make adjustments to the functions you have implemented, they should continue to work for part a) of this assignment. Again, report your classifier performance on the validation set using 10%, . . . , 100% of the examples (using all three features simultaneously).

Keep in mind that your features should not take excessive time to compute. In particular, a training/classification run using all training and validation data should not take more than 2.5 minutes on a single core of a 3GHz commodity CPU.

b.2 (10 points) Report the performance of combining the basic features (those extracted from `extract_basic_features`) and the new feature sets that you just designed. Try the classifier on 10%, . . . , 100% of the examples and report your classifier performance on the validation data set. Compare the result with using only the basic feature set and using only the three new feature sets.

b.3 (10 points) Implement a final feature extractor function

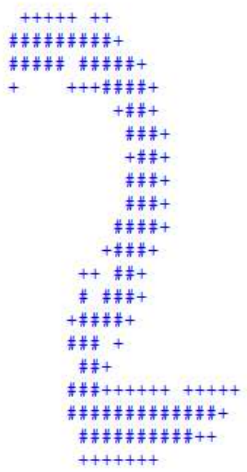
`extract_final_features(`*digit_data*`,` *width*`,` *height*`)`,

which we train on all the training data and then use the resulting classifier to test on a set of test data (not given to you). We will verify that the classifier achieves similar performance on the validation data set and on the test data set. For full score, your classifier should achieve at least 75% accuracy on the test data.

b.4 (7 bonus points) For implementations that achieves at least 80% accuracy on the test data, the top 10 teams will be awarded 7–4–4–4–2–2–2–2–2–2 bonus points with better performing implementation given higher score.

For the basic implementation, running the included `main.py` will yield something that looks like the following.

```
Printing digit example #2473 with label: 2
```



```
Correct prediction: 0.63
```

