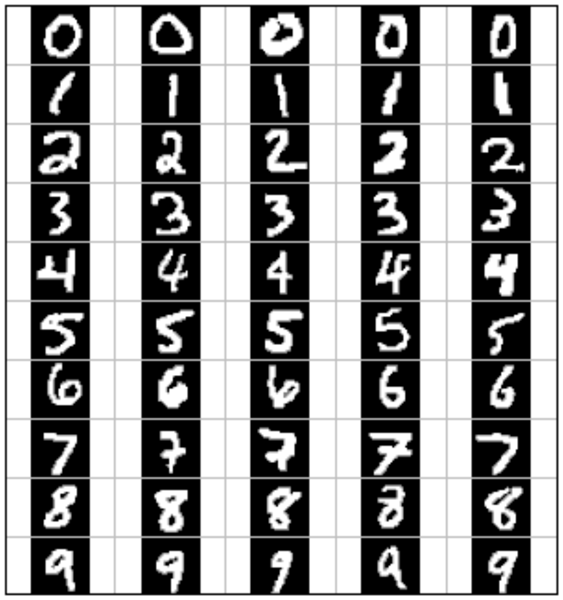
CS 686 – Data Mining
Project 1: Classification


Which Digit?

Which are Faces?

Due in phases, for details see below & Canvas
ZeroR: Sep 12, midnight
FirstFeature: Sep 15, midnight
Q3, basic Naïve Bayes: Sep 19, midnight
Final: September 22, midnight

Introduction

Note: this project is adapted from those of Berkeley's AI class and Hal Daume's machine learning class. We appreciate their making the projects available for others.

In this project, you will design three classifiers: "ZeroR," "First Feature Classifier," and a Naive Bayes classifier. You will test your classifiers on two image data sets: a set of scanned handwritten digit images and a set of face images in which edges have already been detected. Even with simple features, your Naïve Bayes classifier will be able to do quite well on these tasks when given enough training data.

Optical character recognition (OCR) is the task of extracting text from image sources. The first 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).

Face detection is the task of localizing faces within video or still images. The faces can be at any location and vary in size. There are many applications for face detection, including human computer interaction and surveillance. You will attempt a simplified face detection task in which your system is presented with an image that has been pre-processed by an edge detection algorithm. The task is to determine whether the edge image is a face or not. There are several systems in use that perform quite well at the face detection task. One good system is the Face Detector by Schneiderman and Kanade. You can even try it out on your own photos in this demo.

The code for this project includes the following files and data, available as a zip file. Explanation of what you will write is below.

Data file: data.zip – includes the digit and face data

Files you will edit and turn in

zeroR.py	- The location where you will write your zeroR classifier.
firstFeatureClassifier.py	- The location where you will write your firstFeature classifier.
naiveBayes.py	- The location where you will write your naive Bayes classifier.
dataClassifier.py	- The wrapper code that will call your classifiers. You can also write your enhanced feature extractor here (for extra credit). You will also use this code to analyze the behavior of your classifier.

Files you should read but NOT edit

`classificationMethod.py`	Abstract super class for the classifiers you will write. (You **should** read this file carefully to see how the infrastructure is set up.)
`samples.py`	I/O code to read in the classification data.
`util.py`	Code defining some useful tools. You may be familiar with some of these by now, and they will save you a lot of time.
`mostFrequent.py`	A simple baseline classifier that just labels every instance as the most frequent class.

What to submit: You will fill in portions of zeroR.py, firstFeatureClassifier.py, `naiveBayes.py`, and `dataClassifier.py` (only) during the assignment, and submit them. Despite the staged nature, please turn in everything at the end even if you turned it in earlier.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. You are responsible for testing your own code but I provide some guidelines below.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. Instead, contact the course staff if you are having trouble.

Getting Started

To try out the classification pipeline, run `dataClassifier.py` from the command line. This will classify the digit data using the default classifier (`predictOne`) which blindly classifies every example with a "1.".

```
python dataClassifier.py
```

Obviously this is not a very useful classifier, since it doesn't actually learn anything! We'll slowly build up to more intelligent classifiers.

You can learn more about the possible command line options by running:

```
python dataClassifier.py -h
```

We have defined some simple features for you. Later you can try to design some better features. Our simple feature set includes one feature for each pixel location, which can take values 0 or 1 (off or on). The features are encoded as a `Counter` where keys are feature locations (represented as (column,row)) and values are 0 or 1. The face recognition data set has value 1 only for those pixels identified by a Canny edge detector.

Implementation Note: You'll find it easiest to hard-code the binary feature assumption. If you do, make sure you don't include any non-binary features later on (in the extra credit). Or, you can write you code more generally, to handle arbitrary feature values, though this will probably involve a preliminary pass through the training set to find all possible values (and you'll need an "unknown" option in case you encounter a value in the test data that you never saw during training).

ZeroR

Question 1 (10 points) Your first implementation task will be to implement the missing functionality in `zeroR` (for "zero rules" – learning without any conditions). This actually will "learn" something simple: Upon receiving training data, it will simply remember which label y in Y is the most common. It will then always predict this label for future data. Test your code on the default training data (hint: you should always guess 1):

```
python dataClassifier.py -c zeroR
```

You can test on the faces data with the `-d faces` option. This is “P1: Q1” on Canvas and should be fully functional when you turn it in. You are permitted to turn in only zeroR.py for this first due date.

FirstFeatureClassifier

Question 2 (15 points) Your second implementation task will be to implement the missing functionality in firstFeatureClassifier.py. This will do something slightly non-trivial. It looks at the first feature (i.e., `X[0]`) and uses this to make a prediction. Based on the training data, it computes what is the most common class for the case when `X[0] == 0` and the most common class for the case when `X[0] == 1`. Upon receiving a test point, it checks the value of `X[0]` and returns the corresponding class. Test your code on the default training data:

```
python dataClassifier.py -c firstFeatureClf
```

This is “P1: Q2” on Canvas and should be fully functional when you turn it in (for the first 7 points), though if you find bugs later you can update it for the final submission (to get the other 8 points; if it was right the first time you’ll get the 8 pts for free!). You are permitted to turn in only firstFeatureClassifier.py for this second due date.

Naive Bayes

A skeleton implementation of a naive Bayes classifier is provided for you in `naiveBayes.py`. You will fill in the following functions in stages: `justTrain`, `trainAndTune`, `calculateLogJointProbabilities` and `findHighOddsFeatures`.

Theory Review

As discussed in class, a naive Bayes classifier models a joint distribution over a label *Y* and a set of observed random variables, or *features*, (X_1, X_2,..., X_d), using the assumption that the full joint distribution can be factored as follows (features are conditionally independent given the label):

$$P(X_1, \ldots, X_d, Y) = P(Y) \prod_i P(X_i|Y)$$

To classify a datum, we can find the most probable label given the feature values for each variable, using Bayes theorem:

$$\begin{aligned} P(y|\mathbf{x}) &= \frac{P(x_1, x_2, \ldots, x_d|y)P(y)}{P(x_1, x_2, \ldots, x_d)} \\ &= \frac{P(y)\prod_{i=1}^{d} P(x_i|y)}{P(x_1, x_2, \ldots, x_d)} \end{aligned}$$

$$\begin{aligned} \mathtt{argmax}_{y \in Y} P(y|\mathbf{x}) &= \mathtt{argmax}_{y \in Y} \frac{P(y)\prod_{i=1}^{d} P(x_i|y)}{P(x_1, x_2, \ldots, x_d)} \\ &= \mathtt{argmax}_{y \in Y} P(y) \prod_{i=1}^{d} P(x_i|y) \end{aligned}$$

Because multiplying many probabilities together often results in underflow, we will instead compute ***log probabilities*** which have the same argmax:

$$\begin{aligned} \mathtt{argmax}_{y \in Y} \mathtt{log} P(y|\mathbf{x}) &= \mathtt{argmax}_{y \in Y} \mathtt{log} P(y, \mathbf{x}) \\ &= \mathtt{argmax}_{y \in Y} \{\mathtt{log} P(y) + \sum_{i=1}^{d} \mathtt{log} P(x_i|y)\} \end{aligned}$$

To compute logarithms, use `math.log()`, a built-in Python function.

Parameter Estimation

Our naive Bayes model has several parameters to estimate. One parameter is the ***prior distribution*** over labels (digits, or face/not face), $\hat{P}(y)$ where we use a “hat” (^) over a probability to denote our model’s estimate of that probability. We can estimate P(Y) directly from the training data:

$$\hat{P}(y) = c(y)/N$$

where c(y) is the number of training instances with label y and N is the total number of training instances.

The other parameters to estimate are the ***conditional probabilities*** of our features given each label y: $\hat{P}(x_i|y)$. We do this for each possible feature value ($x_i \in \{0,1\}$).

$$\hat{P}(X_i = x_i|Y = y) = \frac{c(x_i, y)}{\sum_{x'_i \in \{0,1\}} c(x'_i, y)}$$

where $c(x_i, y)$ is the number of times variable X_i took value x_i in the training examples of label y.

Question 3 (40 points) Implement `justTrain` (20 points) and `calculateLogJointProbabilities` (20 points) in naiveBayes.py**.** This is a baseline Naïve Bayes without smoothing, and ignores the validation set. Test your classifier with:

```
python dataClassifier.py -c naiveBayes
```

This is "P1: Q3" on Canvas. Try to get as much of it done as you can before this due date, I will check that you have written at least a possibly buggy version (to get 15 pts), which you can update for the final due date (to get the remaining 25 pts). You may turn in only naiveBayes.py for this due date.

Hints and observations:

- The method `calculateLogJointProbabilities` uses the conditional probability tables constructed by `justTrain or trainAndTune` (depending on whether smoothing is used) to compute the log posterior probability for each label y given a feature vector. The comments of the method describe the data structures of the input and output.
- You can add code to the `analysis` method in `dataClassifier.py` to explore the mistakes that your classifier is making. This is optional.
- To run on the face recognition dataset, use `-d faces` (optional but I will test this).

Smoothing

Your current parameter estimates are *unsmoothed*, that is, you are using the empirical estimates for the parameters $\hat{P}(x_i|y)$. These estimates are rarely adequate in real systems. Minimally, we need to make sure that no parameter ever receives an estimate of zero, but good smoothing can boost accuracy quite a bit by reducing overfitting.

In this project, we use *Laplace smoothing*, which adds *k* counts to every possible observation value:

$$\hat{P}(X_i = x_i|Y = y) = \frac{c(x_i, y) + k}{\sum_{x'_i \in \{0,1\}} (c(x'_i, y) + k)}$$

If k=0, the probabilities are unsmoothed. As k grows larger, the probabilities are smoothed more and more. You can use your validation set to determine a good value for k. **Note**: don't smooth P(Y).

Question 4 (20 points) Implement `trainAndTune` in [`naiveBayes.py`](naiveBayes.py). In `trainAndTune`, estimate conditional probabilities from the training data for each possible value of *k* given in the list `kgrid`. Evaluate accuracy on the held-out validation set for each *k* and choose the value with the highest validation accuracy. In case of ties, prefer the *lowest* value of *k*. Test your classifier with:

```
  python dataClassifier.py -c naiveBayes --autotune
```

Hints and observations:

- When trying different values of the smoothing parameter *k*, think about the number of times you scan the training data. Your code should save computation by avoiding redundant reading.
- To run your classifier with only one particular value of *k*, remove the `--autotune` option, which uses k=0 by default as in Question 3. You can change the default k with `–k`, which will call trainAndTune to do smoothing but it will not need to examine the validation set (if you implement it appropriately).
- Using a fixed value of *k=2* and 100 training examples, you should get a validation accuracy of about 69% and a test accuracy of 55%.
- Using `--autotune`, which tries different values of *k*, you should get a validation accuracy of about 74% and a test accuracy of 65%.
- Accuracies may vary slightly because of implementation details. For instance, ties are not deterministically broken in the `Counter.argMax()` method.
- To run on the face recognition dataset, use `-d faces` (optional).

Odds Ratios

One important tool in using classifiers in real domains is being able to inspect what they have learned. One way to inspect a naive Bayes model is to look at the most likely features for a given label.
Another, better, tool for understanding the parameters is to look at odds ratios. For each feature X and classes y_1, y_2, consider the odds ratio:

This ratio will be greater than one for features which cause belief in y_1 to increase relative to y_2.

The features that have the greatest impact at classification time are those with both a high probability (because they appear often in the data) and a high odds ratio (because they strongly bias one label versus another).

Question 5 (15 points) Fill in the function `findHighOddsFeatures(self, label1, label2)`. It should return a list of the 100 features with highest odds ratios for `label1` over `label2`. The option `-o` activates an odds ratio analysis. Use the options `-1 label1 -2 label2` to specify which labels to compare. Running the following command will show you the 100 pixels that best distinguish between a 3 and a 6.

```
  python dataClassifier.py -a -d digits -c naiveBayes -o -1 3 -2 6
```

Feature Design – Extra Credit

Building classifiers is only a small part of getting a good system working for a task. Indeed, the main difference between a good classification system and a bad one is usually not the classifier itself (e.g. decision trees vs. naive Bayes), but rather the quality of the features used. So far, we have used the simplest possible features: the identity of each pixel (being on/off).

To increase your classifier's accuracy further, you will need to extract more useful features from the data. The `EnhancedFeatureExtractorDigit` in `dataClassifier.py` is your new playground. When analyzing your classifiers' results, you should look at some of your errors and look for characteristics of the input that would give the classifier useful information about the label. You can add code to the `analysis` function in `dataClassifier.py` to inspect what your classifier is doing. For instance in the digit data, consider the number of separate, connected regions of white pixels, which varies by digit type. 1, 2, 3, 5, 7 tend to have one contiguous region of white space while the loops in 6, 8, 9 create more. The number of white regions in a 4 depends on the writer. This is an example of a feature that is not directly available to the classifier from the per-pixel information. If your feature extractor adds new features that encode these properties, the classifier will be able exploit them. Note that some features may require non-trivial computation to extract, so write efficient and correct code.

Extra Credit Question (up to 6 points) Add new features for the digit dataset in the `EnhancedFeatureExtractorDigit` function *in such a way that it works with your implementation of the naive Bayes classifier*: this means that for this part, you are restricted to features which can take a finite number of discrete values (and if you have assumed that features are binary valued, then you are restricted to binary features). Note that you can encode a feature which takes 3 values [1,2,3] by using 3 binary features, of which only one is on at the time, to indicate which of the three possibilities you have. In theory, features aren't conditionally independent as naive Bayes requires, but your classifier can still work well in practice. We will test your classifier with the following command:

```
python dataClassifier.py -d digits -c naiveBayes -f -a -t 1000
```

With the basic features (without the `-f` option), your optimal choice of smoothing parameter should yield 82% on the validation set with a test performance of 78%. You will receive 3 points for implementing new feature(s) which yield any improvement at all. You will receive 3 additional points if your new feature(s) give you a test performance greater than or equal to 84% with the above command.