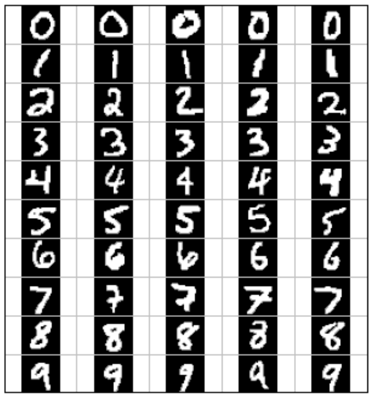
Mar. 09, 2016

Naive Bayes Classifier

– Byung-Jun Lee, Jung-Pyo Hong –

1 Introduction



Which Digit?

Which are Faces?

In this project, you will design a naive Bayes classifier 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 classifiers 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 post office to route mail by zip codes. There are systems that can perform with over 99% classification accuracy.
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.

2 Project Instruction

2.1 Naive Bayes Classifier

2.1.1 Theory

A naive Bayes classifier models a joint distribution over a label Y and a set of observed random variables, or features, $\{F_1, F_2, \dots F_n\}$, using the assumption that the full joint distribution can be factored as follows (features are conditionally independent given the label):

$$P(F_1 \dots F_n, Y) = P(Y) \prod_i P(F_i|Y)$$

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

$$
\begin{aligned}
P(y|f_1,\ldots,f_m) &= \frac{P(f_1,\ldots,f_m|y)P(y)}{P(f_1,\ldots,f_m)} \\
&= \frac{P(y)\prod_{i=1}^{m} P(f_i|y)}{P(f_1,\ldots,f_m)} \\
\text{arg max}_y P(y|f_1,\ldots,f_m) &= \text{arg max}_y \frac{P(y)\prod_{i=1}^{m} P(f_i|y)}{P(f_1,\ldots,f_m)} \\
&= \text{arg max}_y P(y)\prod_{i=1}^{m} P(f_i|y)
\end{aligned}
$$

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

$$
\begin{aligned}
\text{arg max}_y \log P(y|f_1,\ldots,f_m) &= \text{arg max}_y \log P(y,f_1,\ldots,f_m) \\
&= \text{arg max}_y \left(\log P(y) + \sum_{i=1}^{m} \log P(f_1|y)\right)
\end{aligned}
$$

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

2.1.2 Parameter Estimation

Our naive Bayes model has several parameters to estimate. One parameter is the prior distribution over labels (digits, or face/not-face), $P(Y)$.
We can estimate $P(Y)$ directly from the training data:

$$\hat{P}(y) = \frac{c(y)}{n}$$

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: $P(F_i|Y = y)$. We do this for each possible feature value ($f_i \in 0, 1$).

$$\hat{P}(F_i = f_i|Y = y) = \frac{c(f_i, y)}{\sum_{f_i} c(f_i, y)}$$

where $c(f_i, y)$ is the number of times pixel F_i took value f_i in the training examples of label y.

2.1.3 Smoothing

Your current parameter estimates are unsmoothed, that is, you are using the empirical estimates for the parameters $P(f_i|y)$. 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:

$$P(F_i = f_i|Y = y) = \frac{c(F_i = f_i, Y = y) + k}{\sum_{f_i} (c(F_i = f_i, Y = y) + k)}$$

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 $P(Y)$.

2.2 What to Do

You will fill in portions of **naiveBayse.py** during the assignment. You should use **Python 2.7** when scripting your code. You will fill in the *trainAndTune* function, the *calculateLogJointProbabilities* function.
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.
The method *calculateLogJointProbabilities* uses the conditional probability tables constructed by *trainAndTune* 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.

2.3 How to Run the Code

To try out the classification pipeline, run **dataClassifier.py** from the command line. This will classify the digit data using the default classifier (mostFrequent) which blindly classifies every example with the most frequent label.

```
python dataClassifier.py
```

To activate the naive bayes classifier, use -c naiveBayes:

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

Without the option --autotune, kgrid is given to 1 as default. In this case, you can change the smoothing parameter k with -k. --autotune option makes the classifier to be trained with different values of k.

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

To run on the face recognition dataset, use -d faces.

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

2.4 Some Hints

- The *trainAndTune* method calls the method classify to evaluate performance for a given k. classify in turn calls *calculateLogJointProbabilities* which uses the conditional probability tables computed in the current iteration in *trainAndTune*. Make sure you understand that - this would help you code the solution.
- You can add code to the analysis method in **dataClassifier.py** to explore the mistakes that your classifier is making.
- 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.
- 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.

Acknowledgement: This assignment is based closely on the one created by Dan Klein and John DeNero that was given as part of the programming assignments of Berkeley's CS188 course. We thank Dan and John for sharing it.