Naive Bayes

To construct the Naive Bayes Classifier, the following features were used:
Probability of a cell being a blank pixel for a digit X - ‘ ‘
Probability of a cell being a grey pixel for a digit X - ‘+’
Probability of a cell being a dark pixel for a digit X - ‘#’
Probability of digit having Y amount of holes (i.e., 1 has no holes, 2 sometimes has 1 hole, 8 usually has two holes, etc.)

To build the classifier, we iterate through the training data line by line, and character by character, and depending on what character we find, we update the corresponding table that keeps track of the number of occurrences of that feature for that given digit. For the simple pixel features, there are three such tables, initialized to zero:

```
cellProbsGivenX0 = [[[0 for i in range(28)] for j in range(28)] for x in range(10)]
cellProbsGivenX1 = [[[0 for i in range(28)] for j in range(28)] for x in range(10)]
cellProbsGivenX2 = [[[0 for i in range(28)] for j in range(28)] for x in range(10)]
```

where X represents the digit, and i/j represent the cell coordinates.
Once all of the training data has been iterated through, every entry in these tables is divided by the number of total of occurrences of the respective digit. So for example, if in 50% of all 1’s, the very top left pixel was blank, then cellProbsGivenX0[1][0][0] would be equal to 0.5.

While iterating through the training file, we also build temporary matrices representing the digits, such that we can pass them into a function we created, `countHoles(digitGrid)`.

This method uses a simple DFS-search (with a hash map to keep track of visited cells), provided by our quick implementation: `DFSCell(digitGrid,map,i,j,prev)`.

We define a hole as any contiguous grouping of blank pixels that does NOT touch the external border of the grid (first/last rows and columns). Cells are connected either vertically or horizontally, but not diagonally.

In the Bayes equation, we found the denominator to be unnecessary, and thus focused only on the numerator of the equation, would be iteratively updated with the appropriate probabilities for each digit. The classifier would then classify a given digit based on whichever digit scored highest.

Initially, we only implemented two features, whether pixels were grey or dark ('+' or '#'), and we had 65% accuracy. After implementing the third feature of blank pixels, the accuracy jumped up to around 75%. After implementing the fourth feature of counting holes, the accuracy jumped up a WHOPPING 0.6% to 75.6%... And finally, after implementing laplacian smoothing for probability calculation (adding one to the number of each occurrence of features), the accuracy was improved to the final value of 78%.

Training the classifier takes on around 9 seconds, and classifying the given test data takes on around three seconds. The results of the classifier are the same with every run, because it is fully deterministic and there are no random elements.

While we were experimenting with just the classification, it was tedious to have to wait 9 whole seconds for every run (retraining the classifier each time), so we wrote some preprocessing code that gives us the option of training the classifier and then storing the probabilities in a file, such that during the next run, instead of retraining the classifier the probabilities can be simply read in from a file (called "bayesprobs.txt"). This addition is not necessary or super relevant to the assignment, so you can ignore it, and make sure that the boolean variable `preProcessed = False.`

Statistics:

--- 8.56300020218 seconds to proccess training data---

Total Accuracy: 77.9%

Accuracy of digit 0: 84.4444444444%
Accuracy of digit 1: 95.3703703704%
Accuracy of digit 2: 73.786407767%
Accuracy of digit 3: 79.0%
Accuracy of digit 4: 78.5046728972%
Accuracy of digit 5: 68.1318681319%
Accuracy of digit 6: 79.1208791209%
Accuracy of digit 7: 74.5283018868%
Accuracy of digit 8: 64.0776699029%
Accuracy of digit 9: 82.0%

--- 2.98000001907 seconds to classify test data---

--- 11.5430002213 seconds total---