Lab Assignment 2: Classification

Peter Jacobs, Daniel Marchese

Introduction

The purpose of this assignment was to gain experience with classification techniques using the feature vectors generated in the previous lab assignment. As a group, we chose to implement two of the classification methods from scratch (one per person). These two methods were K-Nearest-Neighbors Classification, and the Naïve Bayes Classifier.

5-Fold Cross Validation

In this report, averages are an aggregation of information from the 5 folds of cross-validation. Cross-Validation was run on a data matrix consisting of rows of documents (each row includes topics, and a feature vector of counts for the document). Due to a bug in the previous project, **this data matrix contains 18,103 documents instead of the original 21,000**. In our cross validation method, the documents with non-null topics in the data matrix (9814), were split into 5 roughly equal groups using randomization. All combinations of 4 groups comprising training data and 1 group comprising test data are considered. (In the kNN classifier, the inclusion of documents with non-null topics (18,103 documents total) is also considered).

Vector Preprocessing

Due to differences in the requirements for feature vectors from the previous project, some pre-processing had to take place that was common to both of the classifiers. In both instances, the vectors had to be "paired-down" to 256 features in order to speed up computation time. In order to prevent the creation of null-vectors, an approach was used that purely considered the document frequency for each term, and picked the 256 terms with the highest document frequency. This resulted in very few null vectors (less than 20).

Design and Implementation

The project is structured as a simple nested python package structure. The two different classification methods were implemented and tested using completely separate files in the same package. The only code common to the two classifiers was the cross-validation logic in *crossValidation.py*.

The project consists of several pieces, and the following discusses the design decisions and implementation details of the two classifiers.

K Nearest Neighbors Classifier

The kNN classifier consists of two main files: a driver program which drives the logic of the classifier, and the classifier itself.

knndriver.py

This is the driver program for the kNN classifier code. It was implemented to have several customization points so that various facets of the classifier could be tested.

classifiers/nearestneighbors.py
This is the file that actually contains the logic for performing the kNN classification. It is minimally configurable to prevent any errors resulting from complexity.

Further Feature Vector Processing
In addition to the steps taken beforehand, the kNN classifier makes two additional modifications to the data matrix. First, the data format for the vector was changed from a dictionary mapping words to frequencies, to a standard column vector (thus preserving order) with the features ordered alphabetically. This step also eliminated documents with a null vector. Additionally, documents with more than one topic were duplicated so that each vector could only have one topic.

Training The Classifier
Since a kNN classifier is not "trained", this step simply consisted of indexing the input in a way that made it easier to calculate distances.

Classification
The classification step is also very straightforward. Originally the finding of the nearest neighbors was implemented with a linear-time algorithm, however it was re-implemented as sorting the entire dataset based on the Euclidean distance from the vector question (python's sorting is implemented in C, so it ends up being faster than simple iteration). To break ties in the most common classification, a random class was chosen.

Results
As it turns out, pairing-down the feature vector did not change accuracy that much, the biggest win was in the speed of classification. Two different classification rounds were run for each type of feature vector (full versus paired-down). The first run was done only considering vectors which had a topic; the second run was done considering all the documents (lack of a topic being considered its own class). Using 5-fold cross-validation, the accuracy was shown not to vary between these two approaches for both types of vector. A complete showing of results can be found in *knnresults.txt* inside the data folder.

	1000 w/o NULL	1000 w/ NULL	256 w/o NULL	256 w/ NULL
Avg. online cost	0.066 seconds	0.104 seconds	0.056 seconds	0.093 seconds
Avg. Accuracy	76.6%	74.0%	75.7%	72.4%

Discussion
There are a couple important notes from the results. First is that the accuracy acts exactly as one would expect when removing features from a feature vector (although not statistically significant enough to say anything definitive). The big win occurs in the performance improvement when pairing-down the feature vector. This is expected because the number of calculations decreases for each classification run. Although the differences appear small, when multiplied over thousands of points, they become significant (about 30 seconds per segmentation during cross-validation). In the end, I think the tradeoff of losing accuracy is worth it when it comes to pairing-down the feature vectors for the large gains in performance.

Naïve Bayes Classifier
The Naive Bayes Classifier also contains two files: a driver program which runs the classifier on small input , and the classifier itself.

nbdriver.py
This file runs the classifier twice; once for both feature vector lengths.
To allow the grader to easily test the code, the original input to the classifier used to produce the results discussed below was reduced to 13% of this input using simple random sampling.
For each run, an array is printed. Each number in the array corresponds to accuracy on test data in 1 fold of the 5 folds of cross validation. The measure of accuracy will be discussed below. Note that the accuracy numbers presented in the nbdriver.py are significantly different from the accuracy average reported when the classifier was run on the entire data matrix. This could be due to having different subpopulations of documents in the sgml files; this possibility was not accounted for in the simple random sampling used to create the smaller input.

classifiers/bayes.py
This file contains all on and offline logic used by the classifier.

Scalability – Offline Cost
The Poisson distribution was used to model counts of features in documents. The Poisson distribution was chosen because it is a good model for rare events, and the occurrence of a word in a document in this data set tends to be uncommon.

There were two main aspects of the offline cost for this model: Estimating lambdas, the parameter used in the poisson distribution, and calculating prior probabilitity distributions.

- *Estimating Lambda's*

The online process requires that we be able to calculate the probability of observing a count for feature x in a document given **any** observed class.

Therefore, if we have n classes, we need n poisson probability models for any feature x. Therefore, we estimate n lambda's for every feature x.

An unbiased estimator for lambda is the mean; thus lambda for the count of feature x given class I is the mean of the counts of feature x in documents with class I.
Total lambda estimations is:

(1) #features * #classes

- *Calculating Class Priors*

Total Class Prior calculations:

(2) #classes

Because the number of features heavily influences the number of lambda estimations we have to do, we see approximately a factor of 4 timing decrease when we cut the number of features.

	1000 features (No Nulls)	256 features (No Nulls)
Avg. offline cost	3.067 seconds	.716 seconds

Online Cost

For a given tuple, the cost of making a prediction involves taking the set of values

(1) probability(data | class I)

where I ranges from 1 to the total number of classes, and finding the class where the numerator of this probability is maximized.

To calculate the numerator of this probability for class I, assuming feature counts are independent of one another, we take the product of

(2) probability(observed count for feature x| class I)

where x ranges from 1 to the total number of features, and we multiply this product by the class prior for class I. To obtain 2, we use the poisson distribution with the lambda for the count of feature x given class I.

The total number of online probability calculations involved in obtaining the numerator of (1) is equal to the total number of features. Because we have to calculate the numerator of (1) for each class, the total online work is theoretically approximately

(3) #features * #classes

With this in mind, we see a factor of 4 increase in performance with less features

	1000 features (No Nulls)	256 features (No Nulls)
Avg. online cost	.099 seconds	.025 seconds

Accuracy

The Naive Bayes Classifier attempted to predict the first topic for the document (if it was not null). Therefore, accuracy here refers to the percent of first topics that were predicted correctly by the classifier.

	1000 features (No Nulls)	256 features (No Nulls)
Avg. Accuracy	65.3%	61.8%

Possible explanation for change in accuracy

The smaller number of features will likely make the data matrix more sparse. The Naive Bayes Classifier is sensitive to very sparse data, because (1) goes to zero if just one word has zero total frequency when class I occurs. Even after adjusting for these zero probabilities by adding a small constant to each (2), the Naive Bayes Classifier is still adversely effected by significant reductions in features.

There is a clear tradeoff between accuracy and performance that one must consider when decreasing the number of features used by a Naive Bayes Classifier.

Credits

Each group member worked on implementing and testing one of the classifiers: Daniel Marchese worked on k nearest neighbors, and Peter Jacobs worked on the naïve bayes classifier. Both members wrote the section of the report corresponding to the classification method that they implemented.