Assignment 3

Mukul Sati [msati3@gatech.edu]

April 4, 2016

1 KNN

Note — I have not tested on the office dataset.

1.1 Methodology and notation

I randomly select 20% of data as test (TtD) and considered the rest as training data (TD). I select subsets of p% ($p \in P = \{20, 50, 80, 100\}$) from TD and generate 5 training sets for each of these subsets. The set of training sets is $TSS = \{TS_1, TS_2, TS_3, TS_4, TS_5\}$. It is these TS_i's that I do n-fold cross validation on ($n \in N = \{2, 5, 10, |TS_i|\}$). Note that $|TS_i|$ stands for the cardinality of TS_i and that $|TS_i|$-fold cross validation corresponds to leave one out cross validation.

Notationally, I carry out several cross validation runs. Each run uses a value of k ($k \in K$) for the KNN (the set K used is mentioned for each dataset in the following sections) and carries out n-fold cross validation on a particular TS generated using a particular percent subset of the TD. $R[p_a, t_b, n_c, k_d]$ refers to the run using the a^{th} indexed element of P, the b^{th} indexed element of TSS, the c^{th} indexed element of N and the and the d^{th} indexed element of K. For conciseness, in the remainder of the writeup, I often used the term "cross validation batch" to refer to sets of cross validation runs as determined by conditions on elements of P, T, N, K. For example, the n-fold cross validation batch is the set $B = \{R[p_a, t_b, n_c, k_d] \mid n_c = n, p_a \in P, t_b \in T, k_d \in K\}$. Cross validation batches are also often expressed by abusing Python's splicing notation for indexing. Thus, $B = R[:, :, n_c == n, :]$ or even more shorthand, $B = R[:, :, n, :]$.

Metric Learning: I use the metric-learn module (https://github.com/all-umass/metric-learn) as mentioned by a TA on piazza. I use the LMNN algorithm to learn a Mahalanobis distance metric.

Stability: Each run in the cross validation batch $R[p, :, n, k]$ has an average cross validation error. This is different for each cross validation run, even with the same p due to differing TS_i. Looking at the average cross validation error for each run as samples of a random variable, the stability (here tacitly assumed as the inverse of the variance) of the cross validation error gives some insight on the homogeneity of the dataset and I comment on this stability for each of the datasets. Intuitively, one would also expect the variance to decrease with increasing values of p, and thus, the stability to increase with increasing subset sizes used for cross validation runs. I see if this holds.

Selecting k: The training errors of each batch for a fixed k ($R[:, :, :, k]$) cannot simply be computed by averaging the training error of each run in the batch, as the runs are heterogeneous (are obtained from different values of p and n). Instead, I use the box-and-whisker's plot as suggested by a TA to eyeball a good k for the data-set.

Testing: Now, I train my KNN model using the "optimal" value of k using the entire TD, learning the suitable distance metric and use the trained model for computing the error on TtD. I compare the this with the cross validated errors, determining which cross validation split (value of $n \in N$) had given an error that is most correlated with the error on the test data. The most correlated n also gives some subtle insight into the distribution of data for the dataset. Note that for Wine, I use the 20% split of the data (TtD) for testing, but for MNIST and office datasets, I use the separate testing data. In these two cases, I don't split the training data into test and train as the separate testing data implies no need for hold out testing from the training set itself (thus TD = all of training data, TtD = {} for MNIST and office datasets).

1.2 Results

1.2.1 Wine dataset

Stability of Cross Validation scores within a TS: The cross validation scores for the same sized TS ($R[a, :, c, d]$) are similar to each other, as expected. There is very little variation here, with a maximum variation in score of 0.004. For the same sized TS, cross validation results are stable.

CV scores variation with TS size: The scores are smaller for smaller sized TD (and thus TS), and greater for larger sized TD's, as is expected due to larger size of training data being available for larger TD's. As stated for the same sized TD's but using different values of n and k are observed to be close, suggesting that the size of the TS is the most important factor in determining the score. The box plots described for the selection of best K can be used to see the variation of CV scores with TS size as well. With increasing TS size, the CV scores become less varying.

Best K and associated CV error: The best k is eyeballed using the box and whisker's plot. Fig 1 has a couple of them for particular values of k. I have decided on using k=3 as the best choice. This choice leads to higher scores and relatively lower variances as the performance degrades quickly for $k > 3$. I feel that k = 1 would be a good choice as well.

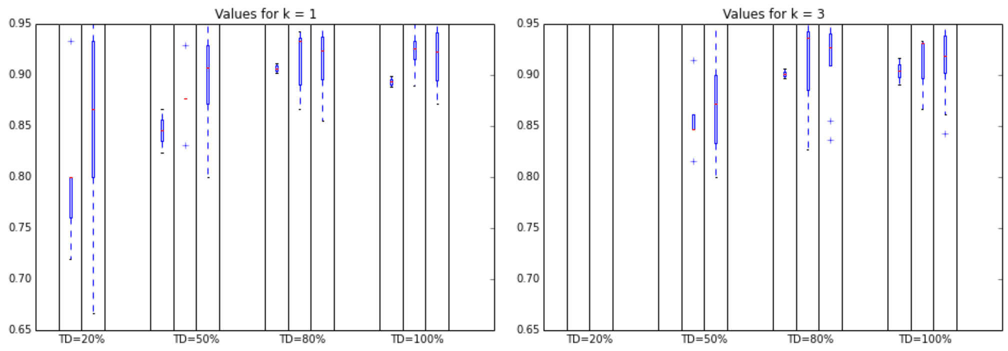

Figure 1: The box and whisker plots corresponding to k=1(left) and k=3(right) for Wine dataset. Note that some data vectors are missing and the reason for this is explained in the discussion. The three close subplots correspond to 2,5 and 10 fold cross validation respectively.

Testing vs CV: Using 3 as the KNN k, I get an accuracy of 91.1% on the testing data. This is closely correlated with the 5 fold cross validation, although the small size of the dataset means that the results are not without significant noise.

1.2.2 MNIST dataset

For MNIST, I used PCA to reduce dimensionality, retaining enough principal component vectors that explain 90% of the variance in the data. I feel this gives me a good balance between a performant algorithm and execution time.

Most of the observations from the wine dataset carry over to MNIST.

1.3 Discussion

For small dataset such as wine, especially when using a smaller subsampled dataset, k-fold cross validation is expected to be extremely noisy, more so for smaller k. In some extreme cases, it may be possible that a k-fold training set does not contain any data for one class, and thus, no testing data would be classified to that class. Alternatively, there could be lesser samples in training for a class than the value of 'k' and, in this case, the LMNN algorithm of metric-learn errors out — in this case, I've capped the range of k's tried to the minimum number of training data-points for any class.

2 HMM

2.1 Methodology

Modeling: The training data can be used to constuct the HMM model for the robot's walks. Each square in the 4 × 4 world corresponds to a HMM state. It is known that the transition probabilities for the non Von-Neumann neighborhood states (if the states are arranged in a 4 × 4 matrix) is zero. The transition probabilities from a particular state to its Von-Neumann neighbors and to the state itself (to account for black cells) can be computed using the training data by frequency counting of the transitions. Notationally:

$$a_{ij} = \frac{\#i \rightarrow j}{\sum_{k=1}^{n} \#i \rightarrow k} \tag{1}$$

Here we use some of the notation from [1] for the matrix of state transition probabilities, and $\#i \rightarrow j$ stands for the number of times a transition from state i is made to state j. Thus, the transition matrix can be determined — this is known to be sparse and thus, the computations can be made computationally efficient.

The observation at each state if one of the colors $\{r, g, b, y\}$. The observation can be modeled as a discrete random variable and the probability distribution for each state can be found by a similar frequency counting of the training data as for the transition matrix computation. The initial state probabilities is computable similarly, thus completing the description of the HMM for the experiment.

Most probable states: Using just the color observation of each step of a testing random walk, the learned model is used to infer the state transitions that best explain the walk. This problem displays optimal substructure, and the Viterbi algorithm provides a dynamic programming solution for it.

2.2 Results

Learned Model: The start state probabilities, the state transition probabilities and the observation symbol probabilities at each state was learned using training data as described above. I only visualize the observation probabilities that are learned as described below.
Observation Probabilities: The observation probabilities may be visualized in a grid for the training data. This is shown in Fig. 2. Note that the observation propabilities match closely the sample output of the world as given in the assignment statement.

Hidden State Inference: The average score for the testing data for my trained HMM is 81.35%. The error is thus around 19%.

2.3 Discussion:

The use of sparsity in the problem allows for a memory efficient implementation. However, this also makes the implementation difficult to parallelize, and seeing an important rationale given by the authors in the paper recently discussed in class [2] for the use of dense approximations to sparse networks in action was a key takeaway for me.

I also noticed propagated probability values reaching extremely low values due to repeated multiplication, and observed the need for a more robust algorithm than the one explained in [?] that works by appropriately scaling the probabilities at each step of the recursion / corresponding dynamic programming implementation.

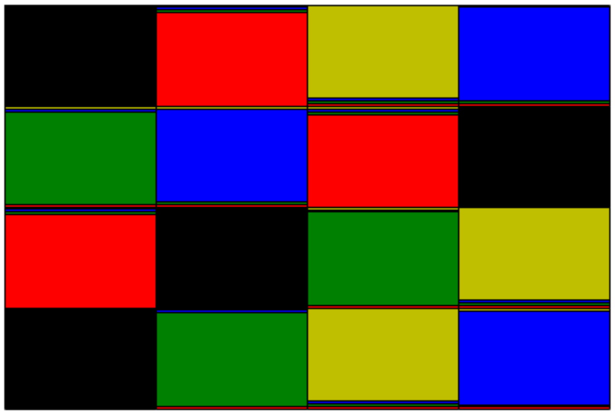

Figure 2: The observation probabilities visualized in a grid for the training data

References

[1] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[2] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.