CS5350: Machine Learning *Due November 17, 2011*

HW4: Dimensionality Reduction, Ensemble Methods, Multiclass/Ranking

1 Written Exercises

1. Linear PCA does an eigen-decomposition of the $D \times D$ covariance matrix of the data where D is the original dimensionality of the data. Kernel PCA (KPCA) which learns nonlinear projections is also based on doing an eigen-decomposition of the covariance matrix but this time the covariance matrix is defined in the feature space (ϕ) defined by the kernel. This feature space could even be infinite dimensional (e.g., if you use an RBF/Gaussian kernel). Explain how does KPCA get away with *not* computing this covariance matrix and doing its eigen-decomposition? What does it actually compute and does eigen-decomposition of? (*10 points*)

   KPCA is based on the intuition that, if a set of N points cannot be linearly separated in $d < N$ dimensions, then it nearly always is linearly separable in $d \geq N$ dimensions. Although we imagine some mapping $\Phi(\boldsymbol{x}_i) : \mathbb{R}^D \to \mathbb{R}^N$, it is undesireable, and possibly intractable to actually perform such computations in that feature space. Instead, we build an $N \times N$ matrix $\boldsymbol{K}$, which is the inner product space of the (probably intractable) feature space. What remains is to compute the $N \times 1$ eigenvector $\boldsymbol{v}_i$ for $\boldsymbol{K}\boldsymbol{v}_i = \lambda N \boldsymbol{v}_i, \forall i \in [1, N]$.

2. Dimensionality reduction methods take as input a set of points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ and learn a low-dimensional embedding (projection) of these points. Now suppose I give you a new point $\boldsymbol{x}_{N+1}$. Of the various dimensionality reduction methods we have seen in the class (PCA, Kernel PCA, LLE, ISOMAP), which ones allow you to compute the low-dimensional projection of the new point $\boldsymbol{x}_{N+1}$ (for each, explain why or why not). This problem is known as the *out-of-sample problem*. Note that including $\boldsymbol{x}_{N+1}$ in the original data and re-running dimensionality reduction on the whole data is not an option. (*10 points*)

3. For multiclass classification, All-vs-All (AVA) seems more computationally intensive at training time than One-vs-All (OVA) because it trains $\mathcal{O}(K^2)$ classifiers rather than $\mathcal{O}(K)$ classifiers. However, all of the K-many OVA classifiers are on the full data set of N examples, while the $\mathcal{O}(K^2)$ AVA classifiers are only on subsets of the data. Suppose that you have N data points, divided evenly into K classes (so that there are N/K examples per class).

   (a) Suppose that the training time for your binary classifier is linear in the number of examples it receives. What is the complexity of training OVA and AVA, as a function of N and K?

   Each of the $\binom{K}{2} \in O(K^2)$ AVA classifiers trains on $2\frac{N}{K}$ elements of data, since each classifier will recieve all elements of one class as positive examples and all elements of another class as negative examples. If we are taking linear time to train, each classifier takes $O(\frac{N}{K})$ time to train. We do this $O(K^2)$ times in total, so the AVA classifier takes $O(\frac{N}{K}K^2) = O(NK)$ time to train, assuming the classifiers are trained sequentially. OVA will train $O(K)$ classifiers, each of which requires N data. So, the total training time should be $O(NK)$ if it takes linear time to train them.

   (b) Suppose the training time is quadratic; then what is the complexity of AVA and OVA?

   If the training time is quadratic, then each component of the AVA model takes $O((\frac{N}{K})^2)$ time to train, for a total of $O((\frac{N}{K})^2K^2) = O(N^2)$, and the OVA takes $O(N^2K)$.

(*10 points*)

4. One issue with multiclass classification based methods such as OVA and AVA is that the test time complexity grows linearly or quadratically with K, the total number of classes. Moreover the multiclass classification error also worsens as K grows (because basically we are using a collection of binary classifiers and their individual errors would just add up).

   Now suppose we still want to construct a multiclass classification algorithm based on a set of binary classifiers but want to control the error such that it doesn't get worse than $\mathcal{O}(\log_2 K)$. How would you accomplish this (hint: what you want is minimizing the number of decisions you have to make at test time)? (*10 points*)

5. Define a ranking preference function ω that penalizes mispredictions *linearly* up to a threshold K. In other words, for $K = 20$, if I put the object that should be in position 5 in position 20, then I pay 15; if I put it in position 30, I only pay 20 because nothing costs more than $K = 20$. (*10 points*)

   A cost function ω should be symmetric, monotonic, and should satisfy the triangle inequality.

   $$\omega(i,j) = \begin{cases} |i-j| & : \quad |i-j| < K \\ K & : \quad \text{otherwise} \end{cases}$$

   We trivially can see that $\omega(i,j)$ will be symmetric, since the $|i-j| = |j-i| \forall i,j$. We trivially can see that it's monotonic, because both $\omega(i,j) = |i-j|$ and $\omega(i,j) = K$ are linear funcitons. We trivially can see that ω satisfies the triangle inequality, since $\omega(i,j)$ will be the distance between i and j, and we can trivially see that if $i < j < k$ or $i > j > k$, the difference between the distance between i and j and the distance between j and k can't ever be smaller than the distance between i and k.

2 Programming Exercises

1. Implement PCA, and Kernel PCA with RBF kernel. For the RBF kernel, there is a fairly decent heuristic for choosing the bandwidth parameter: compute the pairwise distances (Euclidean but don't square it) between all the points and take the *median* Euclidean distance as the bandwidth parameter.

   The shell for both PCA and Kernel PCA are provided (`PCA.m` and `KPCA.m`) and you will need to fill in the TODO parts. Once you complete implementing these (they are independent), you can test them on the MNIST digits dataset (provided in `mnist.mat`).

   The test scripts `test_pca.m` and `test_kpca.m` would test your implementations and will produce several plots, including the figures of reconstructed digits using the projected data. Observe (for both PCA and KPCA) how the reconstruct digits look like as you increase the number of dimensions d the data is projected on.

   Submit the completed codes for PCA and KPCA, and all the plots. Also comment on your observations about the reconstruction accuracies of digits as d varies. (*30 points*)

2. In this task, we will implement AdaBoost. A shell for AdaBoost is in `Boost.m`. You will need to fill in the `BoostTrain` and `BoostPredict` functions. See the comments in `Boost.m` for a description of what each function should do and how it should store its outputs. Note that in order to do boosting properly, you need to have binary classification algorithms that can accept weighted data. I have provided an implmentation of Perceptron that does so in `perceptron_weighted.m` (do not change that code! :)).

   The data for this part is in `data.mat`. When you do `load data.m`, you would see `trX,trY` for the training data and `teX,teY` for the test data.

Once you are done implementing `Boost.m`, you can run `test_adaboost.m` which will run a series of tests using different number of rounds for the boosting algorithm. To get a sense of whether your implementation is doing sensibly, here are some representative numbers: for boosting round 1, you should get about 62.5% training and 55.6% test accuracy; for round 3, you should get training accuracy of about 88.5% and 86.8% test accuracy. (*20 points*)