EE 511 – Homework 4 Solutions

1. **Perceptron problem:** There are two versions of this problem, due to a typo in an earlier version of the homework assignment.

The earlier homework asks to work with the loss function:

$$\frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - \hat{y}_i|$$

where $\mathcal{M}$ is the set of misclassified examples and $T$ is the number of training examples. The perceptron classifier predicts as $\hat{y} = \text{sign}(w^T x)$. The difficulty of this loss function lies in the fact that sign is not easily differentiable.

(a) We’ll tackle the revised homework problem first. It aims to optimize the loss function:

$$\frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - w^T x_i|$$

To find the solution, we write out the loss function as follows (to get rid of the absolute value):

$$L(w) = \frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - w^T x_i|$$

$$= \frac{1}{T} \left( \sum_{i \in \mathcal{M}_p} p(y_i - w^T x_i) + \sum_{i \in \mathcal{M}_n} n(w^T x_i - y_i) \right)$$

where $\mathcal{M}_p$ is the set of positive samples that are misclassified $\{i : y_i = 1, \hat{y}_i = -1\}$ and $\mathcal{M}_n$ is the set of negative samples that are misclassified.

Taking the derivative wrt to $w$:

$$\frac{dL(w)}{dw} = \frac{1}{T} \left( \sum_{i \in \mathcal{M}_p} -px_i + \sum_{i \in \mathcal{M}_n} nx_i \right)$$

To do the perceptron update, we take a one-sample approximation of the above gradient. The update is in the negative direction of the approximate gradient, so the pseudocode is:
initialize \( w_0 = 0 \)
for \( i = 1 \ldots T \)
\[
\hat{y}_i = \text{sign}(w_{i-1}'x_i)
\]
if \( y_i = 1, \hat{y}_i = -1: \) \( w_i = w_{i-1} + \frac{1}{T}px_i \)
if \( y_i = -1, \hat{y}_i = 1: \) \( w_i = w_{i-1} - \frac{1}{T}nx_i \)
end i
return \( w \)

Note that the above loss function based on absolute values \( |y - w'x| \) is identical to the hinge loss function presented in class: \(-yw'x\) (up to a constant scaling factor of 2). We are not taking any approximations in the derivation of the loss function. The only approximation is the stochastic gradient descent step where we evaluate the gradient at one sample (in an online fashion).

(b) Now, let us return to the earlier loss function:

\[
L(w) = \frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - \hat{y}_i| = \frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - \text{sign}(w'x_i)|
\]

Without loss of generality, let us first redefine \( y \in \{-1, 1\} \) to take on values \( y \in \{0, 1\} \), since this will make the math prettier (we will only be scaling the objective by a factor of 2). Accordingly, we redefine the \( \hat{y} = \text{sign}(w'x) \) to output in the same range: \( \hat{y} = \text{sign}(w'x) + 1 \). Now, we can use the sigmoid function as a replacement for the above:

\[
\hat{y} = \text{sign}(w'x) + 1 \approx \sigma(w'x)
\]

Sigmoid function \( \sigma \) can be seen as a curved approximation of the sign step function. We will use it because it has a nice derivative.

\[
\sigma(g(t)) = \frac{1}{1 + e^{-g(t)}}
\]

\[
\frac{\partial \sigma(g(t))}{\partial t} = \sigma(g(t))(1 - \sigma(g(t)))\frac{\partial g(t)}{\partial t}
\]

Now let’s plug in the sigmoid to our loss function:

\[
L(w) = \frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - (\text{sign}(w'x_i) + 1)| \approx \frac{1}{T} \sum_{i \in \mathcal{M}} C(y_i, \hat{y}_i)|y_i - \sigma(w'x)|
\]

Again we can decompose the loss function into two parts:

\[
L(w) = \frac{1}{T} \left[ \sum_{i \in \mathcal{M}_p} p(y_i - \sigma(w'x_i)) + \sum_{i \in \mathcal{M}_n} n(\sigma(w'x_i) - y_i) \right]
\]
Taking the derivative wrt $w$,

\[
\frac{dL(w)}{dw} = \frac{1}{T} \left( \sum_{i \in M_p} -px_i\sigma(w'x_i)(1 - \sigma(w'x_i)) + \sum_{i \in M_n} nx_i\sigma(w'x_i)(1 - \sigma(w'x_i)) \right)
\]  

(5)

In this case, our perceptron algorithm will be similar in structure to the pseudocode above, except for a step size that varies with $w$:

initialize $w_0 = 0$

for $i = 1 \ldots T$

\[
\hat{y}_i = \text{sign}(w'_{i-1}x_i)
\]

if $y_i = 1, \hat{y}_i = -1$: $w_i = w_{i-1} + \frac{1}{T}p\sigma(w'_{i-1}x_i)(1 - \sigma(w'_{i-1}x_i))x_i$

if $y_i = -1, \hat{y}_i = 1$: $w_i = w_{i-1} - \frac{1}{T}n\sigma(w'_{i-1}x_i)(1 - \sigma(w'_{i-1}x_i))x_i$

end $i$

return $w$

This update has an interesting property. If the current $w$ is not confident (i.e. $w'x \approx 0$ for some sample $x$), then the update is most aggressive ($\sigma(t)(1 - \sigma(t))$ is maximized at $t = 0$). On the other hand, if $w$ is quite confident (whether confidently correct or wrong), the update is smaller. It is not exactly clear whether this is a good property or not.

2. KNN Computation reduction:

Baseline

Using the full data set with no dimensionality reduction and no approximations in computation, most people got about 94.42% accuracy for the 1-NN case and about 94.02% for the 5-NN case. (There should be some variation in the 5-NN case since it depends on how you break ties).

Computation Reduction

Feature selection/dimensionality reduction was a popular approach. This has the added benefit that it may improve accuracy, although of course it may also worsen accuracy. Some methods tried were:

- PCA (determining the transform on the training set and applying it to the training and testing data prior to NN). The selected number of dimensions $p$ ranged from 10 to 128 and a slight improvement in accuracy was reported (by one person) over the baseline when 32-128 dimensions were used (and worse accuracy when fewer were used). If 128 dimensions are used, this reduces the terms in the distance computations by half, but it also adds some computations to compute the PCA transform (on the training set) and then the PCA representation of each data point. In general the computation of the PCA
transform is at least $O(nd^2 + d^3)$ (the $nd^2$ term is for the computation of the scatter matrix, and the $d^3$ term for the computation of the eigenvalues). The computation of the transformed data points is $O(ndp)$ where $p$ is the reduced number of dimensions, since it requires $p$ dot products of length $d$ vectors for every data point. So, for relatively large data sets it’s still a good idea since the number of distance terms needed for NN is $O(n^2p)$. Furthermore, if we look only at “test” time computation (neglecting the computation of the PCA vectors and the PCA transforms of the training points), as one person did, we can say the following: if $p = 128$, then for each test point we add $128 \times 256$ multiplications to compute the PCA representation (performed once per test point), but we save 128 multiplications in each distance computation (performed roughly 7000 times per test point). Therefore this method provides reasonably good computational savings and, empirically, good accuracy.

- Fisher multiclass LDA (again, determining the transform on the training set, applying it to the training and testing data prior to NN). In contrast to PCA, this makes use of the class labels available for the training data so it is a smart thing to try. The person who did this chose to reduce the dimensionality to just 9 dimensions, with still very good accuracy (about 89% for both 1-NN and 5-NN; in contrast, PCA with $p = 10$ reported about 90%). The analysis of computational savings is the same as for PCA when we consider only the testing computations, i.e., if we remove half the dimensions we add $128 \times 256$ multiplications per test point but save 128 multiplications in each distance computation.

- Removing uninformative (low-entropy) features: it was observed that some of the pixels were nearly constant in all the images. Several people tried reducing the dimensionality by removing the pixels with the lowest entropy across the images. Results were reported with 1/8 to 1/4 of pixels removed with no reduction in accuracy from the baseline case. (Lower accuracy was reported when only the top 50 highest entropy pixels were used). Reducing the number of pixels by 1/4 saves 1/4 the computation, so this is a simple and accurate method as well, and can be combined with other methods as some people did.

Several computation reduction approaches were tried during the NN search, often on top of dimensionality reduction. They were:

- partial-distance computation (several variations). The basic version works like this: compute the distance by adding distance terms for each dimension as usual, but check after each new dimension term has been added to see if the sum so far exceeds that of the minimum distance training point; if it does, stop the computation and go onto the next training point. This has the potential to reduce the number of dimension terms you have to compute but it does add some > comparisons and “if” statements. One variation on this was to always start with a base of $d$ dimension terms without checking, and then proceed with the usual term-by-term partial-distance procedure as
more terms are added. This has the benefit over the basic version that it doesn’t spend time checking the sum at the beginning, when few terms have been added and it’s unlikely that the sum would yet exceed the minimum. A similar variation was to check after every $m$ new terms, rather than after every term; another was to only do one check during the sum, after $l$ terms. The value of $d, m$ or $l$ to minimize computations was selected by running experiments on a validation set (either the second half of the test set, which was un-used otherwise, or splitting the training set). I am not sure how they measured “computations”; presumably they counted distance terms computed as well as “if” comparisons. Since these partial-distance methods don’t do any approximations in finding the nearest neighbors, accuracy was not affected. Computational savings of $65 - 75\%$ was generally reported (measured by the average number of dimension terms saved per distance computation).

- condensing. The person reported a reduction in the training set by nearly 90% and about 170 test samples misclassified. (I am not sure if this was out of the “full” provided test set, as seemed to be implied—if so, the result is very good, about 8% error).

- tree search. One way this was implemented was by having one “prototype” for each of the 10 classes (the average of the training samples in each class). For each test sample, the closest $C$ classes (by distance to the prototype) were found and then NN search we performed only on the samples in those classes. So if 9 of the classes are retained, this saves 10% if the computation; if 5 of the classes are retained, this saves roughly half the computation. In general this yielded good results: with 5 classes (50% savings) the reported error rates were only $1 - 2\%$ above the reported baselines. One variant on this method was to split each class randomly into $C$ groups and compute prototypes for each of the $10 \cdot C$ groups, then perform nearest neighbor only on the prototypes. This reduces the computation dramatically $C$ is small—there are only $10 \cdot C$ distance computations per test sample rather than 7000. The results also were not too terrible considering the dramatic savings: with $C = 120$ (a savings of more than 80%), the reported error rates were 11% for 1-NN and 14% for 5-NN.

- multi-edit. Two people implemented this. One reported good results, with 20% reduction in the size of the training set and only 2 – 3% reduction in accuracy. The other reported bad results with 50% reduction in the size of the training set. From this it would seem that this method works well when a small amount of training data is removed. Therefore, as with the entropy method, it may work well when a small amount of data is removed and it is then combined with one of the other computation reduction methods.

- geometrical trick involving bounding Euclidean distance below in terms of the $l_1$ norm. This allowed computation of the $l_1$ norm first (which does not involve multiplications), and then Euclidean distance only if the $l_1$ norm exceeded a bound. Like the partial-distance methods, it is a bit tricky to analyze for computational savings (because some
operations are substituted for others). The person who did this reported that 15% of the time the test saved computation of the Euclidean distance for 1-NN, and 11% of the time for 5-NN. The person further combined it with partial distance. Since these are exact methods (no approximation), the accuracy was the same as the baseline.

3. **NN Decision Regions:**
For the standard NN case, you can draw bisecting lines between the green and red dots closest to each other, as in the Voronoi regions for a vector signal in white noise detection problem or a quantization problem. The decision region is given by the union of all these, so you only need to worry about the boundaries that are at the edge of the red/green regions. The default case is illustrated by the solid line in the figure at the end.

For the case where $\lambda_{RG} < \lambda_{GR}$, the decision region for red gets smaller. To understand how the regions will change, consider the two class case with diagonal Gaussians, which corresponds to finding the nearest-neighbor of the two class means. The decision rule is to decide class $G$ when:

$$\log \left( \frac{p(x|G)}{p(x|R)} \right) > \log \frac{\lambda_{GR}}{\lambda_{RG}} + \log \frac{p(R)}{p(G)}$$

$$-|x - \mu_G|^2/(2\sigma^2) + |x - \mu_G|^2/(2\sigma^2) > \log \frac{\lambda_{GR}}{\lambda_{RG}} + \log \frac{p(R)}{p(G)} + 0.5(|\mu_g|^2 + |\mu_R|^2)$$

$$x^t(\mu_G - \mu_R)/\sigma^2 > \log \frac{\lambda_{GR}}{\lambda_{RG}} + \log \frac{p(R)}{p(G)} + 0.5(|\mu_g|^2 + |\mu_R|^2)$$

where $K_i$ correspond to the three additive terms on the right side of the equation and the only term impacted by a change in costs is $K_1 = \sigma^2 \log \frac{\lambda_{GR}}{\lambda_{RG}}$. This will be a constant shift in the decision boundary, which will be parallel to the original boundary. The impact for the NN decision rule is illustrated by the dashed line in the attached figure. Note that if the shift is big enough, then some red samples could end up in the green region.

4. **PCA:** Let $x^m, m = 1 \ldots M$ be the $M$ original data vectors of dimension $d$. Let $e_j, j = 1 \ldots d$ be the set of eigenvectors obtained from covariance matrix of the original data. The set $e_j, j = 1 \ldots d$ form an orthonormal basis and the matrix

$$E = \begin{bmatrix} e_1 & e_2 & \cdots & e_d \end{bmatrix}$$

is the PCA transform matrix, which maps original data vectors into new data vectors as $y^m = E'x^m$. Let $v$ be the mean vector of the new data. Note that $v = \frac{1}{M} \sum_m y^m = \frac{1}{M} \sum_m E'x^m = E' \frac{1}{M} \sum_m x^m = E'u$ where $u$ is the mean of the original data.
(a) To find the sample covariance matrix of the new datapoints $y^m$, simply write out the form of the covariance in terms of $e$ and $x$.

$$
\Sigma_Y = \frac{1}{M} \sum_{m=1}^{M} (y^m - v) (y^m - v)' \\
= \frac{1}{M} \sum_m (E'x^m - E'u)(E'x^m - E'u)'
$$

$$
= \frac{1}{M} \sum_m E'(x^m - u)(x^m - u)' E

= E' \frac{1}{M} \sum_m (x^m - u)(x^m - u)' E

= E'CE
$$

where $C$ is the sample covariance matrix of the original data $x^m, m = 1..M$.

The eigendecomposition of $C$ is $C = E\Lambda E'$ where $E$ is the eigenvector matrix and $\Lambda$ is a diagonal matrix of eigenvalues. Thus $\Sigma_Y = E'(E\Lambda E')E = \Lambda$.

(b) Would you say that it is true that examples can be correlated in one basis but not another? Yes, as shown by the PCA matrix above. ($x^m$ need not have diagonal covariance, but $y^m$ will.)

Does your answer depend on the underlying data distribution? Since the question is about the specific examples, then the reference is to the sample correlation (not true correlation), so the answer is NO since the distribution that generated $x^m$ doesn’t play a role. As long as we can compute a full rank sample covariance matrix, then we can transform the data in such a way that the new sample covariance matrix is diagonal. However, this does not mean that the transform will cause new samples (generated by the same distribution) to be uncorrelated. That would depend on the underlying data distribution, as well as how many samples you have to estimate the covariance.