CSE847 HW2

Javen W. Zamojcin
zamojci1@msu.edu

2024, October 7

1 Question 1: Support Vector Machines

1.1

The primary difference between hard-margin and soft-margin SVMs are how they handle noisy or non-linearly separable data. Hard-margin SVM assumes the data is linearly separable and is sensitive to outliers. Soft-margin SVM can handle non-linearly separable data or data that contains outliers. It does this by introducing a tolerance threshold for misclassified samples.

1.2

The three categories of samples based on the slack variable values are $\xi_i = 0$, meaning x_i is on the correct side of the margin, $0 < \xi_i < 1$, meaning x_i is on the correct side of the hyperplane but on the incorrect margin side, and $\xi_i \geq 1$, meaning x_i is on the incorrect side of the hyperplane or is misclassified. If removing a sample that was exactly on the margin, i.e., forming a support vector, the decision boundary could possibly change if this results in the margin shifting.

2 Question 2: AdaBoost

2.1

2.1.1

One such requirement for the weak classifiers is that they should perform better than random guessing. For binary classification, the expectation of accuracy for randomly guessing would be 50%. The weak classifiers must have an accuracy of more or less than 50%; if less than, the prediction sign may be flipped to achieve higher accuracy results in binary classification.

Additionally, each weak classifier should be focused on different portions of the data. Each weak classifier should have the capacity to adapt to different sample

weights based on their individual difficulty. And obviously the weak classifiers should produce binary output predictions.

2.1.2

If weak classifiers are used with accuracy's of less than 50%, AdaBoost will perform worse with each iteration, deteriorating performance over time by amplifying the errors, unless the weak classifier predictions are inverted. AdaBoost needs weak classifiers with accuracies greater than 50% to improve the strong classifier performance through boosting.

2.2

2.2.1

Calculating the weights without regularization, I produced the follow results shown in Table 1.

0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
0.06546537	0.06546537	0.06546537	0.06546537	0.06546537	0.06546537	0.15275252	0.15275252	0.15275252	0.06546537
0.06021019	0.06021019	0.06021019	0.06021019	0.06021019	0.06021019	0.16608484	0.16608484	0.16608484	0.06021019
0.06000037	0.06000037	0.06000037	0.06000037	0.06000037	0.06000037	0.16666565	0.16666565	0.16666565	0.06000037
0.06	0.06	0.06	0.06	0.06	0.06	0.16666667	0.16666667	0.16666667	0.06
0.06	0.06	0.06	0.06	0.06	0.06	0.16666667	0.16666667	0.16666667	0.06

Figure 1: Q2-P2 AdaBoost Weights

2.2.2

After the 4th iteration, we see that the weights have reached a convergence and we no longer see any updates in the 5th and 6th iterations. No, including these last two classifiers does not further improve the combined classifier performance.

3 Question 3: K-Nearest Neighbor Classifier

3.1

3.1.1

SVM does requires training from scratch for new training samples. SVM works by finding the global optimal hyperplane that circumscribes data into different classes. Adding a new training sample may change the margin and support vectors, hence requiring retraining from scratch.

Naive Bayes does not require training from scratch for new training samples.

Naive Bayes classifiers are probabilistic models with conditionally independent feature variables. When new training samples are provided, the prior probabilities and likelihoods can be updated independently.

K-Nearest Neighbors (KNN) does not require training from scratch for new training samples. KNN is a lazy learning algorithm that stores all training samples in memory and makes predictions considering all stored samples. If a new sample is add, it gets added to the global training sample bank and all future queries will consider this new sample for predictions.

3.1.2

The KNN classifier requires the most computational time for making new sample inferences: $O(n \cdot d)$, where d is the number of features as distance is calculated in the d-dimensional space. KNN stores all training samples, and for each new test sample, it computes the distance for every test-training sample combination to find the k nearest neighbors.

By contrast, once an SVM model is trained, prediction inferences only involve computing the dot product between the support vectors and the test sample. The support vectors are a much smaller subset of the training data that defines the decision boundary. Naive Bayes classifiers use pre-computed probabilities based on the training data for prediction inferences. This involves calculating the posterior probabilities for each class, not for each training sample.

3.2

3.2.1

For implementing the KNN algorithm, I first created a sub-routine for calculating the distance matrices, `calculate_distances()`. It accepts two matrices as parameters, one containing the testing image samples and the other containing training image samples. The distance matrix product has the dimensions of number of testing samples by number of training samples (e.g., (1000x784, 6000x784) $\longrightarrow$ 1000x6000). The distance to each training sample is calculated for each testing sample using an Euclidean distance formula modified for NumPy operations: $D(a, b) = \sqrt{\sum(a - b)^2}$.

After the distance matrix is calculated, `calculate_neighbors()` is called, returning the k nearest neighboring classes for each testing sample. `calculate_neighbors()` utilizes `numpy.argsort()` to first sort each training sample by least distance to each testing sample. The indices of the first k sorted training samples are then mapped to their respective class labels to produce the nearest neighbor list for each testing sample.

After the neighboring classes are calculated for each testing sample, `predict_classes()`

is called to predict the class for each testing sample. `predict_classes()` makes use of `numpy.unique()` and `numpy.argmax()`, to create a mapping of the count of each neighboring class occurrence, and then selecting the highest frequency class. Each testing sample has their list of nearest neighboring classes aggregated into the single, highest frequency class, becoming the prediction class for each testing sample.

Finally, the accuracy and error scores are calculated in `calculate_error()`, accepting the arrays of prediction classes and their respective actual classes, as parameters. Accuracy is calculated as the boolean sum of predictions that match their actual classes, divided by the total length of testing samples: $\frac{\sum(\hat{y}=y)}{count(y)}$. The error score simply being the complement of accuracy.

This entire process is repeated twice for each run for each k parameter, once to produce the training error rate and the second for the testing error rate. The difference between these two is that the training error rate is calculated by using the training sample set for both the training and testing samples, comparing training sample predictions against the actual training samples. The testing error uses the testing set for the testing samples, and makes predictions for the testing set.

3.2.2

From the plot shown in Figure 2, we see several interesting results. First, the testing error rate is very highest even at the lowest k-values of 0 and 9; this suggests the optimal k-value is likely somewhere in this range. Also, as expected we see at k-value of 0, a training error of 0.0%. As the value of K grows, we see an convergence of the training and testing errors, both to a very high error rate of around 85%.

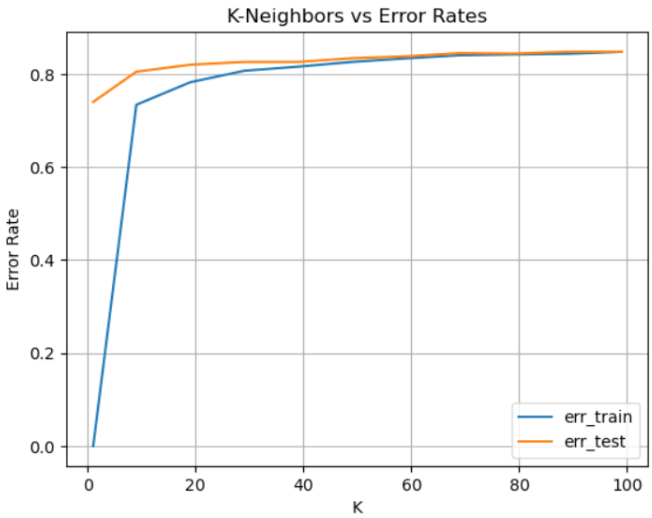

Figure 2: KNN Error Rates