Digit Recognition

Using Perceptron & SVMs

5.

1. Implemented the 4 versions of perceptron algorithm.
2. Simple Perceptron – Perceptron.java (*trainPerceptron()* method)
3. Averaged Perceptron – Perceptron.java (*trainAveragedPerceptron()* method)
4. Kernel Perceptron – KernelPerceptron.java (*trainKernelPerceptron()* with appropriate kernel type, kernel parameters (d/σ) as parameters)
   1. *polynomialKernel()*
   2. *gaussianKernel()*
5. Averaged Kernel Perceptron – KernelPerceptron.java (*trainAveragedKernelPerceptron()* with appropriate kernel type, kernel parameters (d/σ) as parameters)
   1. *polynomialKernel()*
   2. *gaussianKernel()*
6. Validating the exercise 2 problem.
   1. Perceptron algorithm does not converge on this dataset. Ran for different epoch values (3, 5, 10).   
      For epoch 3:   
      Not Converged! ~ in Simple Perceptron  
      W: -0.55 -0.83 0.00 0.00  
        
      For epoch 5:  
      Not Converged! ~ in Simple Perceptron

W: -0.55 -0.83 0.00 0.00

For epoch 10:  
Not Converged! ~ in Simple Perceptron

W: -0.55 -0.83 0.00 0.00

* 1. Smallest polynomial degree at which the dataset converges:  
     For d=1; Epochs 5;

Not Converged! ~ in Kernel Perceptron (POLYNOMIAL)

α = 3 0 4 0 4 0 4 0  
  
For d=2; Epochs 5;

Converged ~ in Kernel Perceptron (POLYNOMIAL) @epoch: 3 With Alpha:

α = 1 0 1 0 1 0 1 0

So, for degree 2 in polynomial kernel perceptron the data converges.

1. Downloaded the training & testing sets from the web link. Then divided training data set into two different files. One as *digits\_development.dat*  with first 1000 examples from the training dataset and the remaining 2823 examples into *digits\_training.dat.*

Now, use *TrainingDataSeparator.java* program (from the package classification in the given source) to split the files into different class files.

Eg:- Digit0.tra, Digit1.tra, …. Digit9.tra – Training Data

Digit0.dev, Digit1.dev, …. Digit9.dev – Development Data

Where each file will have the sign for its class as 1 and the rest as -1. Also saved the *optdigits.tes* for testing.

1. Now, using our validation/development data let’s find the optimal values for the unknown constants in our perceptron models. i.e.
   1. The maximum number of epochs (T) to run in each perceptron to check for convergence
   2. Degree (d) to be taken for the polynomial in Kernel Perceptrons
   3. Sigma (σ) to be taken for the Gaussian Kernel in kernel perceptron.

Note: All of the Kernels & Weight vectors are normalized for the experiments below.

**To Find Best Maximum Epoch count:**

Train the linear perceptron on training data for different epoch counts – {1, 5, 10, 20} and check the overall accuracy on the development data. So the epoch which gives the best overall accuracy will be our T for the rest of the experiments.

Run *OptimalEpochs.java* program from the package *testing*. It will print the total accuracy result for each epoch value. They are aggregated below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Epoch – 1** | **Epoch – 5** | **Epoch – 10** | **Epoch – 20** |
| Accuracy: 88.0 %  Successful : 880  Total Count: 1000 | Accuracy: 90.8 %  Successful : 908  Total Count: 1000 | Accuracy: 87.9 %  Successful : 879  Total Count: 1000 | Accuracy: 89.6 %  Successful : 896  Total Count: 1000 |

So, from the table we can see that for Epoch = 5 we are getting the best overall accuracy. i.e. on all the classes combined.

**To Find Best Degree for Polynomial Kernel:**

This is same as the previous experiment. We will train on training data using the polynomial kernel perceptron for {2, 3, 4, 5, 6} values. And will test on development data to check for the overall accuracies (diagonal elements in a confusion matrix).

Run *OptimalDegree.java* program from the package *testing*. It will print the total accuracy result for each epoch value. They are aggregated below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **d = 2** | **d = 3** | **d = 4** | **d = 5** | **d = 6** |
| Accuracy: 94.7 %  Successful : 947  Total Count: 1000 | Accuracy: 96.5 %  Successful : 965  Total Count: 1000 | Accuracy: 97.0 %  Successful : 970  Total Count: 1000 | Accuracy: 98.3 %  Successful : 983  Total Count: 1000 | Accuracy: 97.4 %  Successful : 974  Total Count: 1000 |

So, from the table we can see that for at degree = 5 we are getting the best overall accuracy. i.e. on all the classes combined.

**To Find Best Sigma for Gaussian Kernel:**

This is same as the previous experiment. We will train on training data using the Gaussian Kernel perceptron for {0.5, 2, 3, 5, 10} values. And will test on development data to check for the overall accuracies.

Run *OptimalSigma.java* program from the package *testing*. It will print the total accuracy result for each epoch value. They are as below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **σ = 0.5** | **σ = 2** | **σ = 3** | **σ = 5** | **σ = 10** |
| Accuracy: 65.9 %  Successful : 659  Total Count: 1000 | Accuracy: 97.1 %  Successful : 971  Total Count: 1000 | Accuracy: 97.2 %  Successful : 972  Total Count: 1000 | Accuracy: 97.2 %  Successful : 972  Total Count: 1000 | Accuracy: 98.2 %  Successful : 982  Total Count: 1000 |

So, from the table we can see that for at sigma = 10 we are getting the best overall accuracy. i.e. on all the classes combined.

**NOTE (2):** I have observed that in testing on test data with σ = 0.5, kernel values are becoming 0. This is zero for all the kernel values between multiple feature vectors. So, the discriminant function value would be 0! Thus it is getting misclassified completely without knowing if it belongs to +1 class or -1 class. This is due to the very small value given by the kernel (eg: e-4050 which goes beyond the bounds of a double and becomes Zero, as the kernel is in e-(||x-y||2)/2\*σ\*σ).

**OBSERVATIONS**

From the above experiments we concluded that the data we are working on behaves well for out models with maximum epochs count as 5, degree of polynomial as 5 & sigma for Gaussian kernel as 10. We will use these values for the rest of the experiments/tests we conduct.

**Overall Performance/Accuracy on Test Data:**

These values are aggregated from the Confusion matrices which are discussed below. (Observed from the Confusion Matrix diagonal)

|  |  |
| --- | --- |
| **Linear Perceptron** | **Average Perceptron** |
| Successful Predictions: 1648  Total Examples: 1797  Accuracy: 91.71 % | Successful Predictions: 1680  Total Examples: 1797  Accuracy: 93.49 % |

|  |  |
| --- | --- |
| **Polynomial Kernel Perceptron** | **Averaged Poly. Kernel Perceptron** |
| Successful Predictions: 1744  Total Examples: 1797  Accuracy: 97.05 % | Successful Predictions: 1753  Total Examples: 1797  Accuracy: 97.55 % |

|  |  |
| --- | --- |
| **Gaussian Kernel Perceptron** | **Averaged Gaussian Kernel Perceptron** |
| Successful Predictions: 1735  Total Examples: 1797  Accuracy: 96.55 % (for σ = 10) | Successful Predictions: 1738  Total Examples: 1797  Accuracy: 96.71 % (for σ = 10) |

By observing all performances of different Perceptron Models we can see that **Averaged Polynomial Kernel Perceptron** performs the best with accuracy 97.55%. Then the second best performer is the **Polynomial Kernel Perceptron** with accuracy 97.05%.

**Comparisons on Timings & Support Vectors:**

Using the values from the above experiments i.e. T = 5; d = 5; σ = 10. I got the following results for timings on learning all the models (i.e. from 0 to 9) for each of the type of Perceptron. This is done using the *System.currentTimeMillis()* method while training in each model.

|  |  |
| --- | --- |
| Perceptron Type | Timing to get trained in all models |
| Simple Perceptron | Total Time: 606 ms |
| Averaged Perceptron | Total Time: 947 ms |
| Polynomial Kernel Perceptron | Total Time: 34144 ms |
| Gaussian Kernel Perceptron | Total Time: 366621 ms |
| Averaged Polynomial Kernel Perceptron | Total Time: 48280 ms |
| Averaged Gaussian Kernel Perceptron | Total Time: 375317 ms |

We can observe that training times for Gaussian are really high, almost > 360 seconds. i.e. more than 5 minutes. And Simple perceptron just took 0.6 seconds to train on the whole data. Gaussian takes longer time because the computation time in calculating the kernel is large, it has to calculate reciprocal to a power (diff of X, Y and squared) of *e* (e-(||x-y||2)/2\*σ\*σ)*.* Where as in polynomial kernels it just have to N x N multiplication and its sum, whose computation time is low. So, Gaussian Perceptrons take longer time to learn and predict as well.

No of support vectors means the non-zero values of αiti for a class models in Kernel Perceptrons. Support vectors would be same for both Normal & Averaged models because the value is just divided by an integer count variable. So, support vector count does not change. Figure is in the next page.

|  |  |  |
| --- | --- | --- |
| Digits | No of Support Vectors in Polynomial Kernel Perceptron & Averaged Polynomial Kernel Perceptron | No of Support Vectors in Gaussian Kernel Perceptron & Averaged Gaussian Kernel Perceptron |
| 0 | 22 | 26 |
| 1 | 103 | 80 |
| 2 | 46 | 41 |
| 3 | 88 | 80 |
| 4 | 45 | 56 |
| 5 | 50 | 59 |
| 6 | 46 | 38 |
| 7 | 42 | 33 |
| 8 | 145 | 87 |
| 9 | 141 | 107 |

**Confusion Matrices:**

Following are the confusion matrices for all 6 types for perceptrons with appropriate headings. The last row & column are the respective sums of each row/column for comparison. This would give brief accuracies for each digit predictions.







\* Accuracy comparisons for different models are given above (page 3)

By observing the confusion matrices we can see that the following digits are the hardest ones to classify in each model:

|  |  |
| --- | --- |
| Perceptron Type | Least Accuracy Digits |
| Linear Perceptron | 1 – 80 % 8 – 84 % |
| Average Linear Perceptron | 8 – 85 % |
| Polynomial Kernel Perceptron | 4 – 95 % 8 – 95.4 % |
| Average Polynomial Kernel Perceptron | 3 – 94.54 % |
| Gaussian Kernel Perceptron | 3 – 90.16 % |
| Average Gaussian Kernel Perceptron | 3 – 90.16 % |

So, from all the above observations we can see that 8 & 3 seems to be the more prevalent one which are misclassified in all the models.

1. I have used SVM Light package to run the SVM algorithms on the data. First to get the needed data to run SVMs we need sparse data formatted input & test files. So, run *SvmFileGeneration.java* which takes each *Digit<i>.tra* file and converts into sparse format. We need sparse files for test data as well. So, first run *TrainingDataSeparator.java* to get *Digit<i>.tes* files, which are inputs to *SvmFileGeneration.java.* Thus we can generate sparse data files from our data files.

Now, let us run *training.sh* Shell script to generate model files for each digit. Then run *classify.sh* shell script to classify test files. Comment or uncomment code depending on our usage of linear / Gaussian SVMs inside these shell script files.

From our observations previously let us take σ = 10 i.e. take ɣ = 0.005 (1/2\*σ2). In polynomial kernel take degree as 5 & option c = 1.

**Overall Accuracy comparison:**

Linear SVM does not need any exclusive parameters except the input data file & output model file.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linear SVM | Polynomial SVM (d=5) | Gaussian SVM (ɣ = 0.005) |
| Successful Predictions | 1665 | 1770 | 1755 |
| Total Examples | 1797 | 1797 | 1797 |
| Accuracy | 92.65 % | 98.50 % | 97.66 % |

We can see that Polynomial SVM got the best accuracy overall. Even in comparison with all the perceptron models, this exceeds.

**Learning Timings:**

The timing for learning / generating model files in SVMs are as below.

|  |  |
| --- | --- |
| **Linear SVM** | 1 sec |
| **Polynomial SVM (d=5)** | 1.40 sec |
| **Gaussian SVM (ɣ = 0.005)** | 9.46 sec |

In comparison with any Linear perceptron model, Linear SVM seems to take almost equal amount of time. Except this, all other models of perceptron takes a lot longer time than SVM models. But there is a phenomenal change in Gaussian learning speeds, it almost took 5 min in perceptron but here it is just less than 10 seconds!

**Support Vectors Count:**

Count of support vectors for each digit in each type of SVM:

|  |  |  |  |
| --- | --- | --- | --- |
| Digits | No of Support Vectors in Linear SVM | No of Support Vectors in Polynomial SVM | No of Support Vectors in Gaussian SVM |
| 0 | 124 | 62 | 2037 |
| 1 | 238 | 145 | 2038 |
| 2 | 202 | 120 | 2050 |
| 3 | 251 | 173 | 2087 |
| 4 | 212 | 126 | 2064 |
| 5 | 234 | 142 | 2086 |
| 6 | 155 | 99 | 2035 |
| 7 | 157 | 111 | 2051 |
| 8 | 395 | 242 | 2082 |
| 9 | 378 | 232 | 2084 |

**Confusion Matrices:** Following are the confusion matrices for the SVM models:





\* Accuracy comparisons for different models are given above (page 7)

By observing these matrices we look for hardest digit to classify:

|  |  |
| --- | --- |
| SVM Linear | 8 – 82.18% accuracy (least) |
| SVM Polynomial | 8 – 95.40% accuracy (least) |
| SVM Gaussian | 8 – 94.82% accuracy (least) |

Our observation with perceptron models also gave less accuracies for digit 8. Even here, in SVMs also 8 seems to a complex digit to classify. But accuracies are better in the Gaussian SVM.