CSE 5160 Project Fall 2021 - Letter Classification

Spencer Wallace
Computer Science and Engineering
California State University of San Bernardino
San Bernardino, CA 92407
spencerdwallace@gmail.com

December 1, 2021

Abstract

The purpose of this project was to compare the accuracy and run-time for the Iterative Bayes (Also known as Recursive Bayes) and K-Nearest Neighbor (KNN) algorithms. The ensuing models were used for letter classification using a data set with 20,000 instances. Along with their accuracy and run-time, the effectiveness of parallelizing the algorithms was also observed. The parallel methods used were MPI and openMP, analysis was done using 10 total test runs for each algorithm and method, with outliers (Z-score over 3) discarded. The results of this project show that the Iterative Bayes algorithm is much more efficient than KNN, and may also be more resistant to noise in the data. Overall, the KNN algorithm was more effective for prediction and found more usefulness in parallel methods.

1 Iterative Bayes and K-Nearest Neighbors

The algorithms used for letter classification were Iterative Bayes and K-Nearest Neighbors (KNN). Iterative Bayes stems from the naive Bayes algorithm, and uses Bayes theorem for probabilities to predict the output for an instance. KNN uses euclidean distance to find the closest neighbors to the current instance, and classifies the current instance as the majority output of its K-nearest neighbors [6].

1.1 Iterative Bayes

The 'training' portion for Iterative Bayes, as well as determining the most probable output, is the same as naive Bayes. Where Iterative Bayes differs is the step after completing all classifications for the test set. After all instances in the test set have been predicted, the instances predicted correctly are now used to update the probabilities from training [2]. Once the probabilities have been updated, classification occurs again on test instances which weren't predicted correctly; and this continues until no new instances are correctly predicted.

1.1.1 Iterative Bayes: Training

```
/*****************************************************************************************************************************************
prb[] stores the total occurences of each output
pGivenC[][][] stores the total occurences for each possible attribute value for each attribute for each output
1st dim = outputs, 2nd dim = attribute, 3rd dim = attribute value
inst[][] contains the dataset - 1st dim = instance, 2nd dim = dimension of input vector (0 = output, 1-16 = attributes)
print is a flag used to print the total occurences of each output, mainly used to verify data set was read from file correctly
****************************************************************************************************************************************/
void findProbClass(int startIndex, int endIndex, int prb[], int  pGivenC[][DIM][ATTRIBUTERANGE], int inst[][DIMWITHY], int print)
{
  initializeProb(prb, pGivenC); //sets each index of the arrays to 0

  for(int xDim = 0; xDim < DIM; xDim++) //DIM is the number of attributes in each instance (num of dimensions in x-input not including the output)
    {
      for(int instanceNum = startIndex; instanceNum < endIndex; instanceNum++)
        {
              int outputValue = inst[instanceNum][0];
              int attributeValue = inst[instanceNum][xDim+1] - 1;
              if(xDim == 0) //output dimension
                prb[outputValue]++;
              pGivenC[outputValue][xDim][attributeValue]++;
        }
    }

  if(print)
    {
      for(int i = 0; i < CLASSIFIERS; i++)//CLASSIFIERS is the amount of possible output values
        {
          printf("Total num of %d is %d.\n", i, prb[i]);
        }
    }

}
```

The training is done in the findProbClass function, which calculates the total occurrences of each possible classification (output) as well as the total occurrences of each possible attribute value for each attribute, given an output.

1.1.2 Iterative Bayes: Testing

```
/**************************************************************************************************************************
res[] is used as a flag for whether or not an instance has been correctly predicted
prb, pGivenC, and inst are the same as they were for findProbClass function
*************************************************************************************************************************/
double naiveBayes(int startInd, int endInd, int res[], int inst[][DIMWITHY], int prb[], int pGivenC[][DIM][ATTRIBUTERANGE])
{
  int accuracy = 0;
  int updateProbabilities[TESTINSTANCES]; //stores instances number for test data that is correctly predicted

  for(int instance = startInd; instance < endInd; instance++) //loop through each test instance
    {

      if(res[instance-TRAININGINSTANCES] == 0) //test instance hasn't been correctly predicted
        {
          int priorP = 0;
          int highOutput = 0; double highestProbability = -1;
          for(int currentOutput = 0; currentOutput < CLASSIFIERS; currentOutput++) //loop through each possible classification
            {
              double totalOutputOccurences = (double)prb[currentOutput];
              double currentOutputProbability = 1;
              double pGc = totalOutputOccurences/TRAININGINSTANCES; //probability of the current ouput

              for(int xDim = 0; xDim < DIM; xDim++) //loop through each dimension
                {
                  int currentAttributeValue = inst[instance][xDim+1] - 1;
                  double xGivenC;
//^ stores probability of current attribute value (currentAttributeValue) at the current dimension (xDim), given the classification value (currentOutput)

                  xGivenC = (double)pGivenC[currentOutput][xDim][currentAttributeValue]/totalOutputOccurences;
                  xGivenC = xGivenC * xGivenC; //square to give weight to higher probabilities
                  currentOutputProbability = currentOutputProbability*xGivenC;
                }
              currentOutputProbability = currentOutputProbability * pGc;
              if(currentOutputProbability > highestProbability) //check if probability given current output is the highest
                {
                  highOutput = currentOutput;
                  highestProbability = currentOutputProbability;
                }
            }
          if(highOutput == inst[instance][0]) //predicted correctly
            {
              updateProbabilities[accuracy] = instance;
              res[instance-TRAININGINSTANCES] = 1; //mark as correctly predicted
              accuracy++;
            }
        }
    }

  updateProb(prb, pGivenC, updateProbabilities, inst, (int)accuracy); //iterative bayes function

  return accuracy;
}
```

The classification occurs in the naiveBayes function, which calculates the probability for all outputs, given the current instance, and predicts the current instance as the output with the highest probability. At the end of this function, the updateProb function is called to continue with the iterative method.

1.1.3 Iterative Bayes: Update Probabilities

```
/*********************************************************************************************************
update[] holds the instance numbers for the instances predicted correctly
*********************************************************************************************************/
void updateProb(int prb[], int pGivenC[][DIM][ATTRIBUTERANGE], int update[], int inst[][DIMWITHY], int size)
{
  for(int instance = 0; instance < size; instance++) //loop through each instance that was correctly predicted
    {
      int currentInstance = update[instance];
      int output = inst[currentInstance][0];
      for(int dim = 0; dim < DIM; dim++) //loop through each dimension
        {
          int value = inst[currentInstance][dim+1] - 1;
          pGivenC[output][dim][value]++;
          if(dim == 0)
            prb[output]++;
        }
    }
}
```

The updateProb function uses the correctly predicted instances to update the totals found in the training phase; or the findProbClass function. These updated totals are then used in the next iteration of the naiveBayes function

1.2 K-Nearest Neighbors

In this implementation the training and classification section of KNN occur in the same function. To find the K-nearest neighbors, each test instance must be compared against every training instance, calculating the distance between these instances. It is worth noting that for this algorithm the plurality output of the K-nearest neighbors is used, rather than majority, to classify the test instance.

1.2.1 KNN: Training and Classification

```
double KNN(int startInd, int endInd, int inst[][DIMWITHY])
{
  double accuracy = 0;
  double neighborDist[TESTINSTANCES][NUMNEIGHBORS]; //stores distance of K nearest neighbors
  int neighborOutput[TESTINSTANCES][NUMNEIGHBORS]; //stores output value of each neighbor
  for(int test = startInd; test < endInd; test++) //loop through all test instances
    {

      for(int instance = 0; instance < TRAININGINSTANCES; instance++) //loop through all training instances
        {
          double furthestNeighbor = 0;
          double distance = 0;
          for(int dim = 0; dim < DIM; dim++) //loop through each dimension
            {
              double difference = ( (double)inst[instance][dim+1] - (double)inst[test][dim+1] );
              distance += difference * difference;
            }
          distance = sqrt(distance);

          if(instance < NUMNEIGHBORS) //for first K num of training instances
            {
              neighborDist[test-startInd][instance] = distance;
              neighborOutput[test-startInd][instance] = inst[instance][0];
            }
          else
            {
              int indexFN = 0; //index of the furthest neighbor, neighbor to be replaced
              furthestNeighbor = neighborDist[test-startInd][indexFN];
              for(int i = 1; i < NUMNEIGHBORS; i++)
                {
                  if(neighborDist[test-startInd][i] < furthestNeighbor)
                    {
                      furthestNeighbor = neighborDist[test-startInd][i];
                      indexFN = i;
                    }
                }
              if(distance < furthestNeighbor)
                {
                  neighborDist[test-startInd][indexFN] = distance;
                  neighborOutput[test-startInd][indexFN] = inst[instance][0];
                }
            }
        }
      int majOutput = neighborOutput[test-startInd][0];
      int mostOccur = 0;

      for(int i = 0; i < NUMNEIGHBORS-1; i++)//find majority output of neighbors
        {
          int numOccur;
          for(int s = i; s < NUMNEIGHBORS; s++)
            {
              if(neighborOutput[test-startInd][i] == neighborOutput[test-startInd][s])
                numOccur++;
            }
          if(numOccur > mostOccur)
            {
              mostOccur = numOccur;
              majOutput = neighborOutput[test-startInd][i];
            }
        }
      if(majOutput  == inst[test][0]) //correctly predicted
          accuracy++;
    }

  return accuracy;
}
```

In the KNN function the distance between the current instance and every instance in the training set is calculated. The K least distances are stored in an array along with their outputs in a separate array. After iterating over the entire training set, the majority output among these K neighbors is found and is used to classify the instance.

2 Explaining the Data

Example Instance: T, 2, 8, 3, 5, 1, 8, 13, 0, 6, 6, 10, 8, 0, 8, 0, 8

2.1 The Data Set

The data set used for the training and testing of the models can be found on the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/letter+recognition). This data set consists of 20,000 instances, each with 16 attributes and an output.

2.2 The Attributes

Referencing the example instance above, the first column of the instance is the output and the rest of the following columns are the attributes, or the independent variables. These attributes give different information about the pixels on the screen when observing a specific letter. This data was collected using 20 different fonts, and randomly distorting the images produced by these fonts to create 20,000 unique instances. More details on the attributes can be found using the link to the UCI machine learning repository.

2.3 Preprocessing

Very little preprocessing was required for this data. Each of the attribute values is normalized within a range of 0-15 by the author of the data set; there were also no errors in the data set [1].

3 Algorithm Comparison

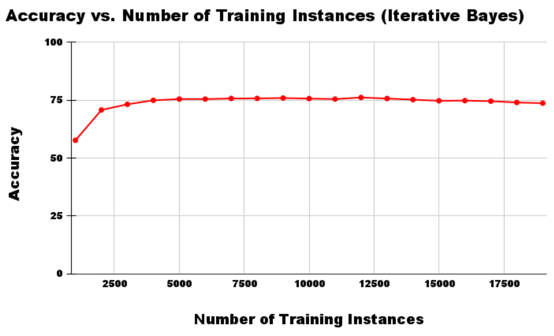
According to the author's of the data set used for these models, a Holland-Style adaptive Classifier obtained a best-accuracy of a little over 80% [1]. This can serve as a baseline for these models.

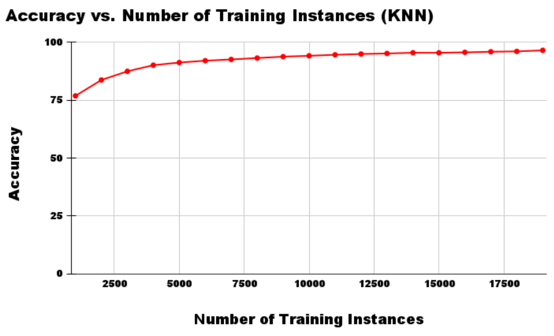
3.1 Accuracy

The Accuracy is determined over a single run, the number of training and test instances for each algorithm were chosen based on what generated the best accuracy; where both are divisible by 1,000 and the training set is not greater than 80% of the total data. The number of training instances used for the Iterative Bayes model was 12,000 and the remaining instances were used for testing. The accuracy for the Iterative Bayes model was 76.25%. The number of training instances used for KNN was 16,000 and again the remaining instances were used for testing; the value for K was 1. The accuracy for the KNN model was 95.65%

3.2 Analysis

Although the KNN model achieves a higher accuracy, there were a few significant observations while testing for optimal model parameters and run-times. For KNN, it was observed that increasing K to be more than 1 rapidly reduced the accuracy of the model, with K equal to 5 resulting in an accuracy of 30.275% and K equal to 10 resulting in 3.65%. This suggests that the KNN model is strongly affected by noise in the data, and it also follows that this data set possibly has a large amount of noise. The Iterative Bayes model is inherently affected by noise in the data, as all noisy data is used for calculating the probabilities, however we see an acceptable result compared to when the KNN model is influenced by noise. For very noisy data, the Iterative Bayes model may have better results.





When observing the accuracy vs number of training instances for both models, we find that for the KNN model the accuracy and number of training instances are directly related. For the Iterative Bayes model, it can be seen that at first the accuracy increases comparably to the KNN model. However, after using about 50% of the total data for training the accuracy begins to decline. These results demonstrate a key difference between Iterative Bayes and naive Bayes, being that Iterative Bayes benefits from using a smaller portion of the data as the initial training set. This allows the model to filter the most effective training instances as it iterates [2]. One proposal is that Iterative Bayes may be better than naive Bayes at classifying common cases, but it may also be affected by over-fitting resulting in unique instances being mispredicted at a higher rate than naive Bayes.

3.3 Run-times

The run-times are collected over 10 runs for each method. Test runs which had a Z-score over 3 were discarded and another run was performed. For Iterative Bayes the average sequential run-time was 75 milliseconds (ms), and for KNN the average sequential run-time was 4,678ms. This makes the sequential Iterative Bayes algorithm approximately 60 times faster than KNN for this data.

4 Parallel Methods

For the parallel methods, 8 cores (processes, threads) on an Intel i7 7700k were used. For both algorithms, and both methods, file read was done sequentially and the rest of the program ran in parallel. Like the sequential run-times, the run-times for the parallel methods were recorded over 10 test runs. The model of Bulk Synchronous Parallelism (BSP) was used to parallelize the sequential algorithms.

4.1 Iterative Bayes

For Iterative Bayes essentially all of the work occurs in the findProbClass and naiveBayes functions. The work for these functions is split evenly between all cores making the sequential algorithm a good fit for the BSP model [3].

4.1.1 MPI

Limitations of the MPI method were compounded by the iterative nature of Iterative Bayes. There are two main communication points in the parallel algorithm, the initial communication after first calculating the probabilities (training portion) and after updating the probabilities with the correctly predicted instances following each iteration of classifications. These communications cause additional synchronization costs which ultimately made the MPI method 52% slower than the openMP method, at an average run-time of 35ms; this gives a speedup of 53.33% compared to the sequential method.

4.1.2 openMP

Synchronization is also necessary for the openMP model, in the form of barriers and what openMP calls critical regions, these regions are used to ensure only one process enters the region at a time [4]. This is required to avoid 'race conditions', where multiple threads read the same variable, update the variable, and then 'race' to be the last thread to write their updated value back to memory [5]. Because of this, a critical region is needed during the training phase; specifically the instruction "pGivenC[outputValue][xDim][attributeValue]++;" from the figure in section 1.1.1. A critical region is also needed when updating the probabilities after the naiveBayes function from section 1.1.2. During testing it was found that making the entire function for updating probabilities (updateProb, section 1.1.3) a critical region increased overall speedup. The results of openMP gave a 70.67% speedup compared to the sequential method, with an average run-time of 22ms.

4.2 K-Nearest Neighbors

The sequential KNN algorithm is what you may call 'trivially parallel', meaning that parallelizing the algorithm is as simple as splitting the work up amongst multiple processes. Nearly all of the work occurs in the KNN function shown in section 1.2.1. Each test instance is completely independent of all other test instances, thus no communication or synchronization is required. Because there is almost no change from the sequential algorithm, and no complications with either method, the individual methods will not be explained in detail. Both methods achieved run-times close to the theoretical maximum speedup (87.5% for 8 cores) with MPI performing 82.75% faster than sequential (807ms) and openMP performing 82.21% faster than sequential (832ms).

5 Conclusion

The Iterative Bayes and K-Nearest Neighbor models provide results comparable or better than the baseline results from Frey and Slate [1]. Iterative Bayes offers a very efficient algorithm - when comparing the best case for Iterative Bayes (openMP) and the best case for KNN (MPI), the Iterative Bayes algorithm runs over 36 times faster than KNN. We see that KNN benefits greatly from parallelization decreasing the speedup offered from Iterative Bayes by approximately 40% when comparing the sequential and parallel times. Results for accuracy show that the KNN model performs much better than Iterative Bayes for this data. However, when adjusting K to be greater than 1, the accuracy quickly declines suggesting it is strongly influenced by noise in the data. With a larger data set (greater than 80,000 instances) I predict that we would see great improvements in accuracy for the Iterative Bayes method, and more resilience to noise in the KNN method.

6 References

[1] P. W. Frey and D. J. Slate (Machine Learning Vol 6 #2 March 91): "Letter Recognition Using Holland-style Adaptive Classifiers".

[2] Chang, H., Shuku, T., Theoni, K., Tempone, P., Luding, S., & Magnanimo, V. (2019). An iterative Bayesian filtering framework for fast and automated calibration of DEM models. Computer Methods in Applied Mechanics and Engineering, 350, 268–294. Retrieved November 29, 2021, from https://www.sciencedirect.com/science/article/pii/S0045782519300520.

[3] Bisseling RH, McColl WF (1993) Scientific computing on bulk synchronous parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[4] #pragma OMP critical. IBM. (n.d.). Retrieved December 1, 2021, from https://www.ibm.com/docs/en/xl-c-aix/13.1.2?topic=processing-pragma-omp-critical.

[5] Netzer, R. H., & Miller, B. P. (1992). What are race conditions? ACM Letters on Programming Languages and Systems, 1(1), 74–88. https://doi.org/10.1145/130616.130623

[6] Taunk, K., De, S., Verma, S., & Swetapadma, A. (2019). A brief review of nearest neighbor algorithm for learning and classification. 2019 International Conference on Intelligent Computing and Control Systems (ICCS). https://doi.org/10.1109/iccs45141.2019.9065747