Naive Bayes Classifier with Feature Selection

Jiakai Xu ax2155, Amy Qi xq2224

December 21, 2023

Contents

1 Introduction

1.1 Naive Bayes Classifier

The Naive Bayes classifier is a probabilistic machine learning model based on Bayes' theorem. It is widely used for classification tasks, particularly in natural language processing and document categorization. The term "naive" is employed because the model makes a strong assumption of independence among the features, which simplifies the computation and facilitates efficient training.

To maximize the probability that given observations X the correct is Y, we need to find the maximum value $P(Y|X)$ out of all classes.

1.2 Assumption about Our Data

Since the naive Bayes classifier has the assumption that our data follows some distribution, we assume that for each feature in our dataset, the values follow a normal distribution, defined by its mean and variance.

The reason why we choose normal distribution is that, by the Central Limit Theorem (mention it here because it's pretty cool ☺), the distribution of a normalized version of the sample mean converges to a standard normal distribution. This indicates that if we have a lot of data, it is safe to assume that the distribution resembles a normal distribution. In fact, many things in nature follow a normal distribution, including this classical Iris Dataset people use for classification programs.

In our program, we implemented both the data generator that generates data following a normal distribution and a CSV file reader to read input files. The reason why we choose to generate data instead of relying on the CSV reader is that we do not want to program to be IO-bound.

1.3 Feature Selection with Cross Validation

This Haskell program's primary contribution lies in its implementation of feature selection with cross-validation. The goal is to determine the most "predictive" feature among various options. The process involves training numerous models using different features and comparing their performances. To ensure that the test data remains untouched until the actual testing stage, a portion of the training data is set aside as validation data. This dataset serves as the "fake" test data for evaluating various models, and the method is called cross validation.

1.4 Pseudocode

for feature in features: **do**
  for train-validation split: **do**
    use training data (of this feature) to train the model
    use validation data (of this feature) to test the model
    get error rate
  end for
  average error rate for this feature
end for
choose the feature with the minimal averaged error rate

2 Sequential Version

2.1 Data Generation

This is the function we implemented to generate data. We plant a "best feature" in our data manually, by using this feature as a base feature, where different labels get different normal distributions. Then using this base feature, we generated other features by adding different noise vectors onto the base feature. In theory, if we implement our program correctly, the best feature returned should always be the base feature since it is the only feature vector that follows a strict normal distribution. A graph below with size of dataset=3 and number of features=5 is presented to demonstrate our idea.

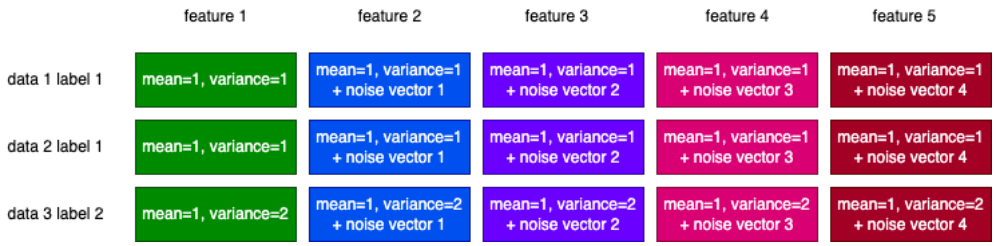


Figure 1: Generate Data

Here is the code to generate the dataset. We will parallelize it in the next section. Apart from this data generator, we also have a CSV reader to read files. This part of the code is omitted but you can find it in the appendix section.

```
generateDataset :: Int -> Int -> [(Double, Double)] -> [Double] -> Dataset
generateDataset totalSize maxValue featureParams noisesArray =
  let labels = concatMap (replicate (totalSize `div` maxValue))
   ↪  [1..maxValue]
      baseFeature = concatMap (uncurry (generateNormalFeature (totalSize
       ↪  `div` maxValue))) featureParams
      features = map (\noise -> zipWith (+) (generateNormalFeature totalSize
       ↪  0 noise) baseFeature) noisesArray
  in zipFeaturesToDataset labels features
```

2.2 Feature Selection

To conduct feature selection, we call evaluateFeature on each possible choice of features. We store the values returned by evaluateFeature, which are error rates, in a list, and find the minimum value. The feature corresponding to this value is the best feature since it results in the smallest error rates.

In our sequential implementation, we used list comprehension to achieve similar functionality of a loop in imperative programming. This part will be refactored later to enable parallelism.

```
findBestFeature :: Int -> Dataset -> (Int, ErrorRate)
findBestFeature k dataset =
  let numFeatures = length (snd (head dataset))
      errorRates = map (\idx -> (idx, evaluateFeature k idx dataset)) [0 ..
      ↪  numFeatures - 1]
   in minimumBy (comparing snd) errorRates

evaluateFeature :: Int -> Int -> Dataset -> ErrorRate
evaluateFeature k featureIndex dataset =
  let featureOnly = extractFeature dataset featureIndex
   in kFoldCrossValidation k featureOnly
```

2.3 K-fold Cross Validation

Similarly, in the sequential implementation, we used list comprehension to iterate through each possible way of train-validation split. In each iteration, we train and evaluate the model, then we take the average of all models trained on this feature.

```
kFoldCrossValidation :: Int -> Dataset -> ErrorRate
kFoldCrossValidation k dataset =
  let errorRates = map (\i -> trainAndValidate (splitData i k dataset)) [0
  ↪  .. k - 1]
  in averageErrorRates errorRates

trainAndValidate :: ([LabeledFeatures], [LabeledFeatures]) -> ErrorRate
trainAndValidate tvPair =
  let trainingData = fst tvPair
      validationData = snd tvPair
      model = trainModel trainingData
      predicted = predict model (extractFeatures validationData)
   in calculateErrorRate predicted (extractLabels validationData)
```

3 Parallel Version

Below are some strategies we used for the parallel version of our program, but we certainly experimented with a lot more strategies. In the next section Performance Evaluation we will compare the performance of different strategies. Also, we have attached the code for all strategies we have tried at the end of this report.

3.1 Data Generation

Here is a parallelized version we wrote for data generation. However, this version is not actually used in our final implementation because it triggers too many sparks, and a lot of them are garbage collected (See Table 2 in Appendix for the statistics). Without this parallelized data generator, our program runs faster. Therefore we stick to the sequential generator in the previous section.

```
generateDatasetParallel :: Int -> Int -> [(Double, Double)] -> [Double] ->
↪  Dataset
generateDatasetParallel totalSize maxValue featureParams noisesArray =
  let labels = concat $ parMap rpar (replicate (totalSize `div` maxValue))
  ↪  [1..maxValue]
      baseFeature = concat $ parMap rpar (uncurry (generateNormalFeature
      ↪  (totalSize `div` maxValue))) featureParams
      features = parMap rpar (\noise -> zipWith (+) (generateNormalFeature
      ↪  totalSize 0 noise) baseFeature) noisesArray
  in plainDatasetParallel labels features
```

3.2 Feature Selection

We refactored the code for the sequential version to use parMap in feature selection. More experiments using other strategies can be found in the next section.

```
findBestFeature :: Int -> Dataset -> (Int, ErrorRate)
findBestFeature k dataset =
  let numFeatures = length (snd (head dataset))
      errorRates = parMap rpar  (\idx -> (idx, evaluateFeature k idx
      ↪  dataset)) [0 .. numFeatures - 1]
      -- multiple implementations other than parMap
      -- details in section 4 Performance Evaluation
   in minimumBy (comparing snd) errorRates
```

3.3 K-fold Cross Validation

We refactored the code for the sequential version to use parListChunk in k-fold cross validation. More experiments using other strategies can be found in the next section.

```
kFoldCrossValidationPLC :: Int -> Dataset -> ErrorRate
kFoldCrossValidationPLC k dataset =
  let chunkSize = ceiling ((fromIntegral k / 4) :: Double)
      errorRates = runEval $
         parListChunk chunkSize rpar $ map (\i -> trainAndValidate
          ↪  (splitData i k dataset)) [0 .. k - 1]
         -- multiple implementations other than parListChunk
      -- details in section 4 Performance Evaluation
  in averageErrorRates errorRates
```

4 Performance Evaluation

We tested the correctness of our classifier using the Iris Dataset with the CSV reader, and achieved a test error rate of less than 1%, which should guarantee the soundness of our program. However, the size of the Iris Dataset is just 150, which is not so interesting for parallel programming. Thus, we only focus on the data generator in performance evaluation.

All tests below are run in the following environment setup:
Apple M2 Chip (4 Efficiency Cores, 8 Performance Cores, Single Threaded), 32GB RAM

4.1 Different Strategies

Consider the algorithm described in Section 1.4, there are two loops where we can apply different paralleling strategies: one is the outer loop where we iterate through different features, and the other one is the inner loop where we iterate through different ways for train-validation split (i.e. different folds). For different features, we tried parMap and parBuffer. For different validation folds, we tried parMap, parListChunk and parBuffer. The reason why we did not implement parListChunk on the outer loop is that in the setting below, we only have 5 features, and parListChunk behaves very similar to parMap.

```
Training data Size: 1,000,000
Test Data Size: 100
Number of Labels: 5
Number of Features: 5
Number of Folds: 10
Number of Cores:8
```

# of Features=m	# of Folds=n	converted	overflowed	dud	GC'd	fizzled	speed	conversion%
	parMap (n)	98	0	0	4	8	10.819	89.1%
parMap(m)	parListChunk (4)	68	0	0	4	8	10.823	85.0%
	parBuffer ($n/2$)	71	0	0	2	37	11.181	64.5%
	parMap (n)	94	0	5	3	8	11.584	85.5%
parBuffer($m/2$)	parListChunk (4)	64	0	5	2	9	11.334	80.0%
	parBuffer ($n/2$)	68	0	5	1	36	11.63	61.8%

Table 1: N=8 Result with Different Strategies

As demonstrated in Table 1, we achieved high conversion rates and few sparks ended up in overflowed/dud/GC'd/fizzled. That means with the current settings, our parallel implementation is reasonable. Also, note that the parMap, parListChunk combination archives a very high conversion rate and high speed, and this will be the combination we use for the rest of the tests conducted on other parameters.

Here we show the ThreadScope output using this optimal combination. For more ThreadScope outputs, see the Appendix.

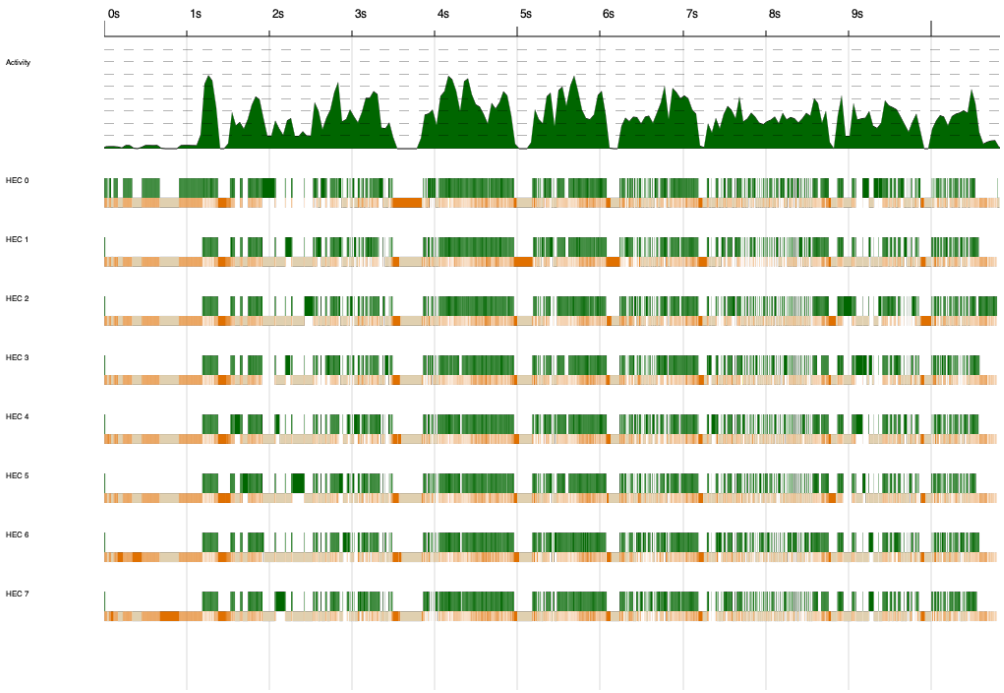


Figure 2: ThreadScope Output Using parMap, parListChunk

Note that in the first 1s we are still generating data using the sequential method, therefore no parallelism appears in that period.

4.2 Different Numbers of Cores

Here is a graph demonstrating the average speed-up ratio (compared with our sequential version) if we use different numbers of cores. This test is conducted using the best strategy combination chosen from the experiments above, and the best result we get is around 55% speed-up. The settings we use for this experiment is

```
Training data Size: 1,000,000
Test Data Size: 100
Number of Labels: 5
Number of Features: 5
Number of Folds: 10
Number of Cores: Various
```

We get a significant boost from using one core to using two cores, but not so significantly afterward, probably because of increasing overheads and the fact that we have a moderate level of parallelism.

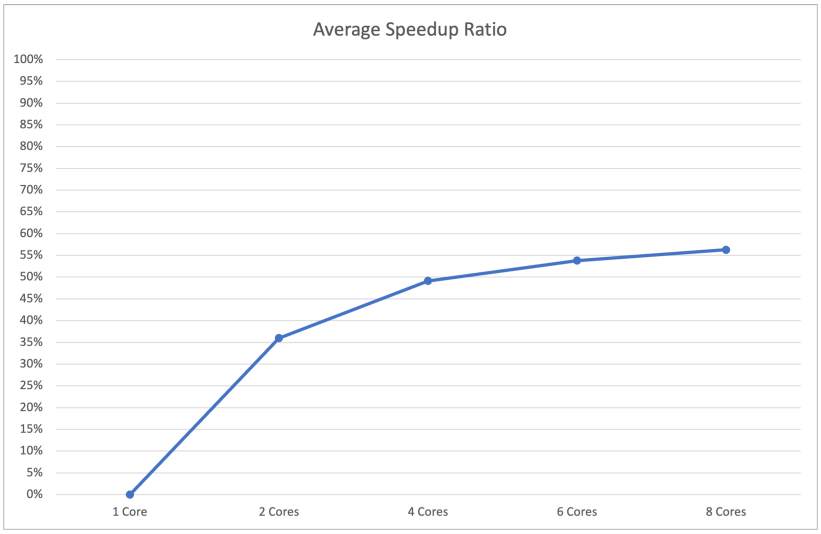


Figure 3: Speed-up Ratio with Different Numbers of Cores

4.3 Different Data Sizes

We calculate the mean and variance over the dataset for each model we train, the time complexity for this operation is $O(n)$. With the sequential implementation, increasing the size of the training data should result in a linear increase in runtime. Compared with the sequential implementation, our parallel version is slightly faster. This is shown in the experiment below. The setting we use for this experiment is

```
Training data Size: Various
Test Data Size: 100
Number of Labels: 5
Number of Features: 5
Number of Folds: 10
Number of Cores: 8
```

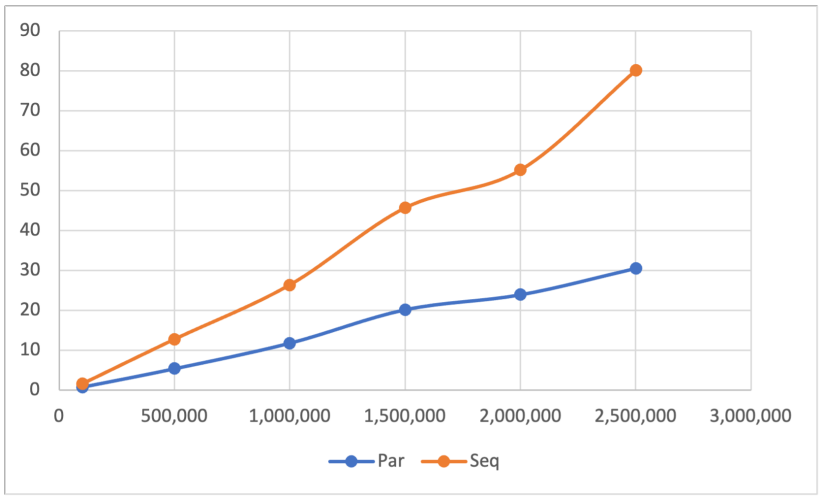


Figure 4: Runtime with Different Data Sizes

4.4 Different Numbers of Folds

In our pseudocode, the number of times the inner loop is executed is decided by the number of folds. In theory, as the number of folds grows, runtime should increase linearly. Compared with the sequential implementation, our parallel version is slightly faster. This is shown in the experiment below. The setting we use for this experiment is

```
Training data Size: 1,000,000
Test Data Size: 100
Number of Labels: 5
Number of Features: 5
Number of Folds: Various
Number of Cores: 8
```

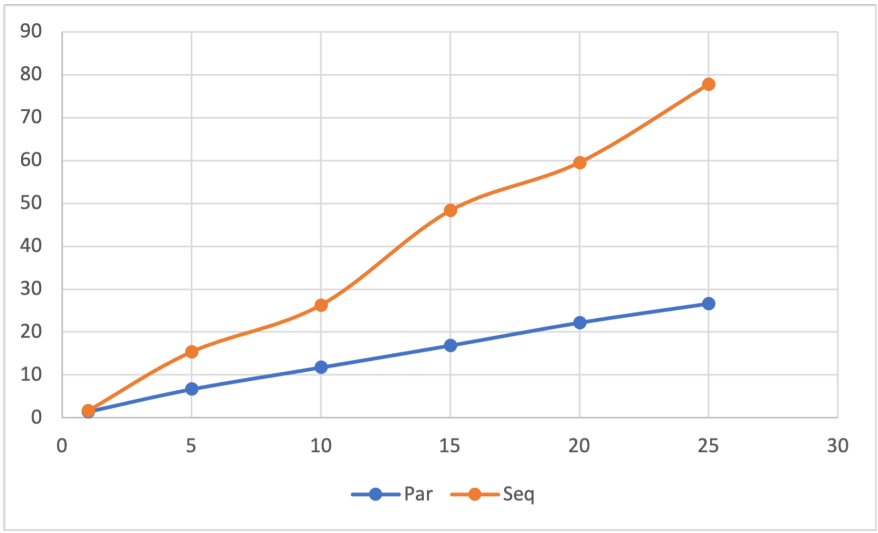


Figure 5: Runtime with Different Numbers of Folds

5 Future Works

Several future improvements we are considering include:

- Intermediate values.
  Although our algorithm does not store a lot of intermediate values thanks to its "naive" nature, meaning that features are independent, we do store some intermediate values such as the list of error rates to be averaged and the list of error rates from which we pick the minimum. One possible way to speed up our program is to get rid of these intermediate values.

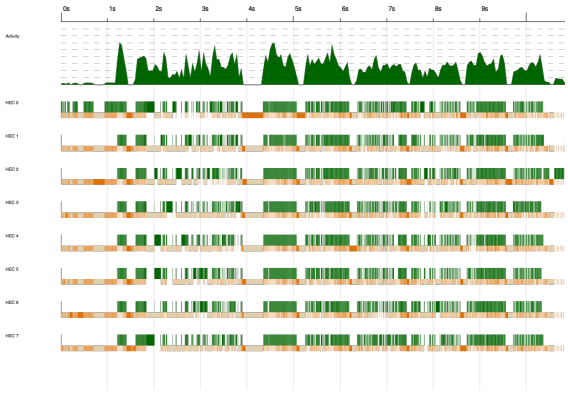
- Larger fold values/number of features
  We tried very large values for the data size (1 million), but we did not have statistics for cases where the number of folds or the number of feature values is huge. In fact, when we tried fold values = 1 million, the program didn't finish even after hours. We realize that with the computational resources we have on our laptops, running 1 million iterations of heavy computation might not be feasible. But with greater computational power, we'd like to see what large fold values/feature values could do to our program.

☺

6 Appendix

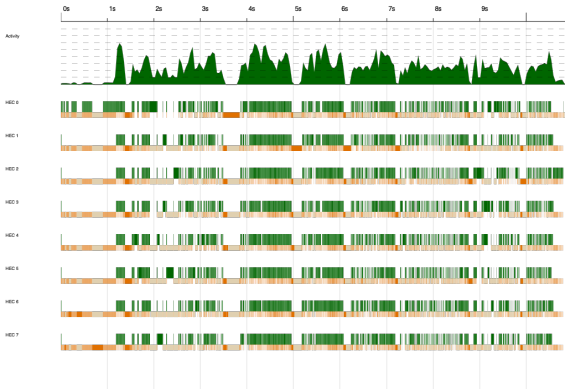
6.1 Figures

ThreadScope Outputs

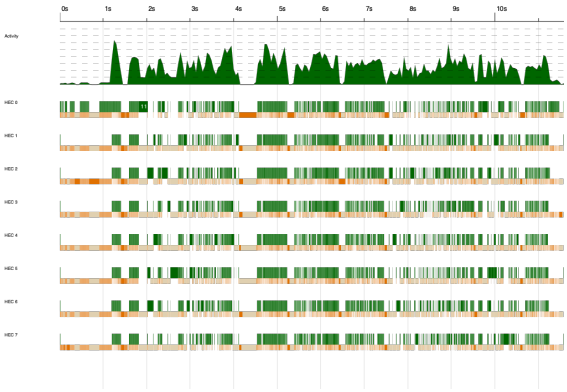


(a) outer: parMap; inner: parMap

(b) outer: parMap; inner: parBuffer

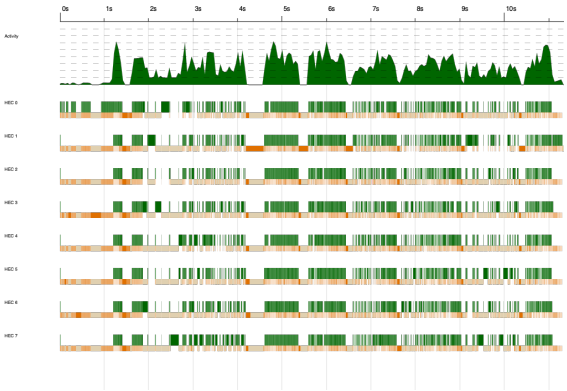


(a) outer: parMap; inner: parListChunk



(b) outer: parBuffer; inner: parMap

(a) outer: parBuffer; inner: parBuffer



(b) outer: parBuffer; inner: parListChunk

6.2 Table

Features=m	**Folds=n**	**converted**	**overflowed**	**dud**	**GC'd**	**fizzled**	**N=8**
	parMap (n)	22,133,824	13,118,334	0	21,909,356	2,667,753	84.496
parMap(m)	parListChunk (4)	22,130,818	11,799,704	0	23,087,003	1,673,023	72.331
	parBuffer ($n/2$)	22,785,809	13,571,071	0	19,983,393	4,439,267	72.398
	parMap (n)	22,592,144	13,559,339	5	23,422,609	1,115,330	86.809
parBuffer($m/2$)	parListChunk (4)	22,480,395	13,613,745	5	22,500,614	2,070,062	76.725
	parBuffer ($n/2$)	19,752,126	15,878,618	5	20,265,291	4,162,606	74.250

Table 2: N=8 Result with Different Strategies using the parallelized version of data generator

6.3 Code

Code can be found in our GitHub Repo.
Details on how to run our code can be found in the README file.