CSCI 447 Project 1 Paper

Kellen Hurley KELLEN.HURLEY@STUDENT.MONTANA.EDU
Department of Computer Science
Montana State University
Bozeman, MT 59717-3880, USA

Alex Ellingsen ALEXANDER.ELLINGSEN@STUDENT.MONTANA.EDU
Department of Computer Science
Montana State University
Bozeman, MT 59717-3880, USA

Editor: Kellen Hurley, Alex Ellingsen

Abstract

In this paper, we train and test a version of the Naive Bayes learning algorithm on five data sets that vary in size, dimension, and number of classes. During the training and testing process, we use 10-fold cross-validation while also ensuring that the data is stratified across the folds. We test the model twice on each dataset; once on the unaltered dataset (aside from discretization and data imputation), and once on an altered version of the dataset where 10% of the features are shuffled between examples in order to introduce noise into the data.

Keywords: Naive Bayes, discretization, data imputation, cross-validation

1 Problem Statement

Naive Bayes classifiers are generative classifiers that operate according to principles from Bayesian decision theory. They are named as such because they make the "naive" assumption that all of the features are conditionally independent given the class. This is a bold assumption, as many datasets in the real world do not reflect this behavior. However, this assumption is also powerful in the sense that it reduces the time complexity of the algorithm from exponential time to linear time. It is also important to note that Naive Bayes is a very simple model in which the number of parameters scales linearly with the number of features. Given these facts, our goal in this paper is to test Naive Bayes on five datasets: *breast-cancer-wisconsin*, *glass*, *house-votes-84*, *iris*, and *soybean-small* in order understand how Naive Bayes performs on many different kinds of datasets in which this conditional independence assumption may or may not hold. Moreover, we also wish to understand how Naive Bayes performs on noisy data, as this may give us an indication as to whether or not it has a propensity to overfit the data. After an analysis of the datasets listed above, we have determined that the *soybean-small* and the *glass* datasets will provide a significantly greater challenge for the model as compared to the other three. The *glass* dataset is composed of nine continuous attributes. Due to the limitations imposed by Naive Bayes, these values must be discretized in order for the algorithm to accurately predict a class. However, this will cause issues as binning continuous features results in loss of specificity and detail. This

loss of accuracy will significantly reduce the model's ability to accurately predict classes of our test data set. On top of this, the *glass* dataset has extremely high dimension after one-hot encoding, containing over *939* dimensions. The *soybean-small* dataset will provide a different challenge, as the conditional independence assumption is especially inconsistent. There are three attributes in particular that stand out: precip, temp, and hail. These three phenomena are often times interlinked and heavily influence the values of the others, which leads us to conclude that these features are likely conditionally dependent. It is also worth noting that this dataset is relatively high-dimensional compared with some of the others. Thus, we propose two hypotheses. We first hypothesize that Naive Bayes will perform worse on the *glass* and *soybean-small* datasets in statistically significant ways when compared with the others. Secondly, we hypothesize that Naive Bayes will perform substantially worse on the noisy data because of the model's simplicity. Extending this idea, we predict that Naive Bayes will likely have a propensity to underfit the data.

2 Experimental Approach and Program Design

In order to test our hypotheses, we designed and implemented a version of the Naive Bayes learning algorithm to run on the five datasets. However, before the algorithm can be run on any of these datasets, there are some considerations that must be taken into account. The first is that all five of the datasets have imperfections that need to be addressed. These imperfections include missing values and continuous and categorical features. To fix missing values, we replaced them with a different value depending on whether the column it was found in was categorical or numeric. If it was categorical, the value would be selected from any of the unique values that the column held, picked randomly using the distribution of values in the column as weights. If the column was numeric, the missing value was simply replaced with the mean of that column. To deal with continuous features, we discretized them by performing binning using the Freedman-Diaconis rule. Then, we used one-hot encoding on both the newly binned features and the categorical features to allow our model to easily interpret these columns. The second consideration is how to ensure that multiple different partitions of the datasets are used for training and testing since the algorithm could train very well on one partition of the data, but poorly for the others. In order to address this we implemented 10-fold cross-validation. We split each data set into ten stratified folds, meaning each partition has approximately 10% of the data. The distribution of classes in each fold was modeled as closely as possible to the distribution of classes in the overall dataset. This allowed us to ensure each fold was representative of the entire data set and could be an accurate predictor of performance. We iterated through these folds ten times, with nine of the folds being used for training, and the last one being used as a testing set. Each iteration provided different training and testing sets, with each fold acting as the testing set once. After the data has been fully pre-processed, the algorithm can now train and test on the five datasets. Our training algorithm differs slightly from the typical Naive Bayes algorithm because it implements something known as *Laplace smoothing* (also known as additive smoothing). In order to understand why this decision was made, consider the typical Naive Bayes classifier:

For each class $c_i \in C$, calculate:

$$C(x) = P(C = c_i) \cdot \prod_{j=1}^{d} P(x_{A_j} = a_k | c_i)$$

Then return:

$$class(x) = \underset{c_i \in C}{argmax}\, C(x)$$

where $P(C = c_i)$ is the prior probability of the particular class c_i, and $P(x_{A_j} = a_k|c_i)$ is the conditional probability of the feature A_j of example x equalling value a_k given class c_i. Now consider a scenario where we want to classify an example x from the test set where a particular $x_{A_j} = a_k$, but for no example x_{train} in the training set is there an instance where $A_j = a_k$. Then $P(x_{A_j} = a_k|c_i) = 0$ for all classes $c_i \in C$. This implies that $C(x) = 0$ for all classes c_i, and the classification is now completely random even though the other conditional probabilities may still strongly indicate once class over another. This is where Laplace smoothing comes in. Laplace smoothing eliminates the possibility that any of the conditional probabilities is zero by making the following alterations:

For each class $c_i \in C$, calculate:

$$C(x) = P(C = c_i) \cdot \prod_{j=1}^{d} \frac{\#\{x_{A_j} = a_k \wedge x \in c_i\} + 1}{N_{c_i} + d}$$

Then return:

$$class(x) = \underset{c_i \in C}{argmax}\, C(x)$$

where $\#\{x_{A_j} = a_k \wedge x \in c_i\}$ is the number of times $x_{A_j} = a_k$ *and* $x \in c_i$, N_{c_i} is the number of examples in c_i, and d is the number of features. Notice that the numerator $\#\{x_{A_j} = a_k \wedge x \in c_i\} + 1 \neq 0$ because $\#\{x_{A_j} = a_k \wedge x \in c_i\} \geq 0$. Thus, even if a particular feature value in the test set is never seen in the training set, the product term will not go to zero. With that decision addressed, let us go over the actual steps of the training algorithm. This was somewhat explained in the previous explanation, but to be completely explicit, the algorithm is as follows:

Step 1. Calculate the prior probability for each class $c_i \in C$

$$P(C = c_i) = \frac{\#\{x \in c_i\}}{N}$$

where N is the total number of examples in the training set. Then save these calculations to some data structure.

Step 2. For each feature A_j in the training set, calculate:

$$\frac{\#\{x_{A_j} = a_k \wedge x \in c_i\} + 1}{N_{c_i} + d}$$

for each class $c_i \in C$ and each value a_k that A_j takes on in the training set. Then save these calculations to some data structure.

This is how the the algorithm runs in theory, however let's now discuss how this works in our program. The class in our program relevant to the training algorithm is called *NaiveBayesModel*. It takes a pandas dataframe called *dataframe* representing the training set as input to its constructor, and it contains four methods called *class_probability()*, *feature_prob_given_class()*, *load_data()*, and *train_model()*. *load_data()* iterates through the "class" column of the training dataframe, adding each new class to a list called *classes*. While iterating through this column, it also updates a data field called *number_of_classes* representing the number of unique classes in the training dataframe, and a dictionary called *number_of_examples_in_class* which maps each class to the number of examples present in that class. Next, *load_data()* iterates through each column and adds each feature name to a list called *features*, and updates a field called *number_of_features*. Lastly, it takes the length of the training dataframe and saves it to a field called *number_of_examples*. The purpose of this method is to save general information about the dataset so that the other methods can access it in constant time during the actual training. The method *class_probability()* calculates the prior probability of a given class by accessing the *number_of_examples_in_class* dictionary and obtaining the appropriate value, and dividing that by the value stored in the field *number_of_examples*. *feature_prob_given_class()* takes in a feature name, feature value, and class, and calculates the conditional probability that the given feature equals the given value given the class. It does this by iterating through the rows of the training dataframe and checking if the value in the column corresponding to the feature name is the feature value, as well as checking if the entry in the "class" column equals the class passed to the method. If both of these conditions are true, then it add/assigns 1 to a local variable *feature_val_occurs_in_class*. After going through all rows of the dataframe, the method returns:

$$\frac{feature\_val\_occurs\_in\_class + 1}{number\_of\_examples\_in\_class[class] + number\_of\_features}$$

This expression is the same as the Laplace smoothing step discussed earlier. Lastly, the *train_model()* method calculates the prior probability of each class by iterating through the *classes* list and calling the *class_probability()* method on each class, saving these probabilities to a dictionary *prior_prob_of_classes*. Then, it calculates all of the conditional probabilities for each feature and its values by running through each entry in the dataframe and each class in the *classes* list and passing the feature name, feature value, and each class to the *feature_prob_given_class()* method. These probabilities are saved to the *conditional_prob_of_features* dictionary. That's the entire training process. In order to test the model created by *NaiveBayesModel*, we have a class called *ModelTest*. *ModelTest* inherits from *NaiveBayesModel* and is the class that is used to actually do both the training and testing steps. We separated *NaiveBayesModel* and *ModelTest* because we felt it would be more organized to split the training and testing methods into two files. *ModelTest* has four methods, *classify_one()*, *classify_all()*, *zero_one_loss()*, and *recall()*. The *classify_one()* method takes in a dictionary *feature_vector* that maps feature names to feature values. *classify_one()* first instantiates a dictionary called *results_per_class* that will be used to map each class to the result generated by $C(x)$ on that class, where x is the example being

classified. Then for each class, it calculates $C(x)$. It does this by first instantiating a variable *repeated_product* and assigning it to one. Then it accesses *feature_vector* and passes the relevant information to the *conditional_prob_of_features* dictionary to obtain the conditional probability (with Laplace smoothing). Finally, it multiply/assigns *repeated_product* to the conditional probability. This process represents the repeated product portion of the classification function discussed earlier. After going through each feature:

$$repeated\_product = \prod_{j=1}^{d} \frac{\#\{x_{A_j} = a_k \wedge x \in c_i\} + 1}{N_{c_i} + d}$$

Then the method multiplies *repeated_product* by the prior probability of the class given by *prior_prob_of_classes* and we have now obtained $C(x)$ for a particular class. This process is repeated for each class and the results are saved to *results_per_class*. *classify_one()* then returns the class associated with the maximum value in *results_per_class*. The rest of the methods are straightforward, *classify_all()* simply iterates through each example in the test set and calls *classify_one()*. *zero_one_loss()* simply counts the number of misclassifications and returns it. Finally, *recall()* counts up the true positives, true negatives, false positives, and false negatives for each class and returns the recall. That wraps up the design of the testing process. In order to actually test the model on the two versions of each of the five datasets, we simply trained and tested a model for each of the ten different fold combinations for both versions of each dataset and returned the 0/1-loss and recall to evaluate the performance. The results of these tests can be seen in the next section.

3 Results from Experiments

Our model resulted in the following recall and 0/1 loss, separated by data set.

3.1 *breast-cancer-wisonsin*

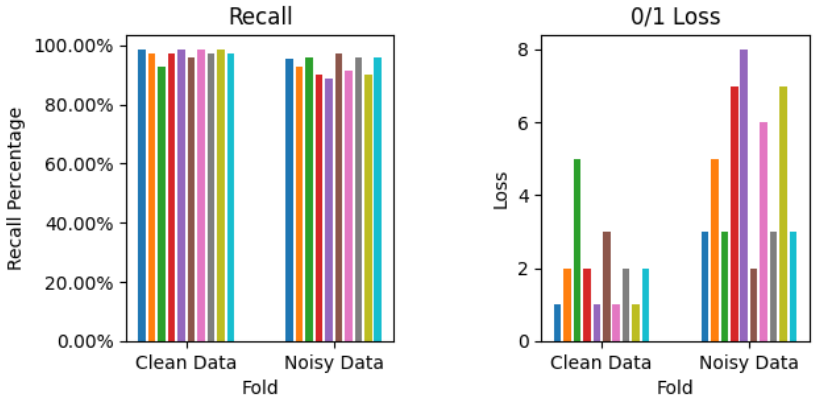

Figure 1: A graph of the recall and 0/1 loss on the *breast-cancer-wisconsin* data set

	Clean Data	Noisy Data
Recall	97.14 ± 1.78%	93.28 ± 3.08%
0/1 Loss (Average of 69.9 samples)	2.0	4.7

Table 1: A table displaying the average values of recall and 0/1 loss calculated using 10-fold cross-validation on the *breast-cancer-wisconsin* data set

3.2 *iris*

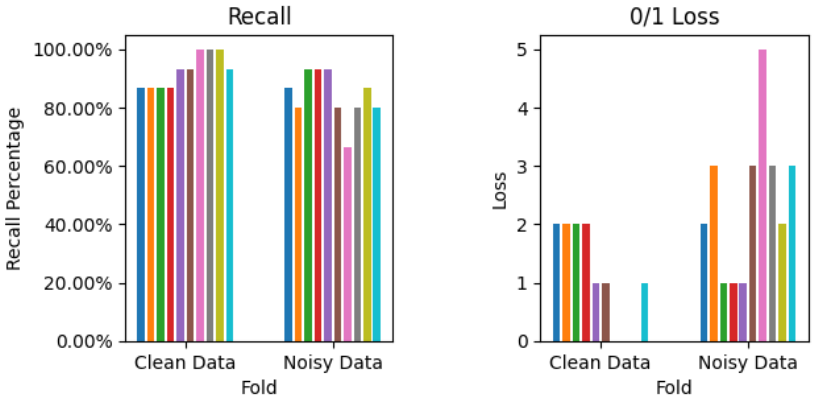

Figure 2: A graph of the recall and 0/1 loss on the *iris* data set

	Clean Data	Noisy Data
Recall	92.67 ± 5.84%	84.00 ± 8.43%
0/1 Loss (Average of 15 examples)	1.1	2.4

Table 2: A table displaying the average values of recall and 0/1 loss calculated using 10-fold cross-validation on the *iris* data set

3.3 *house-votes-84*

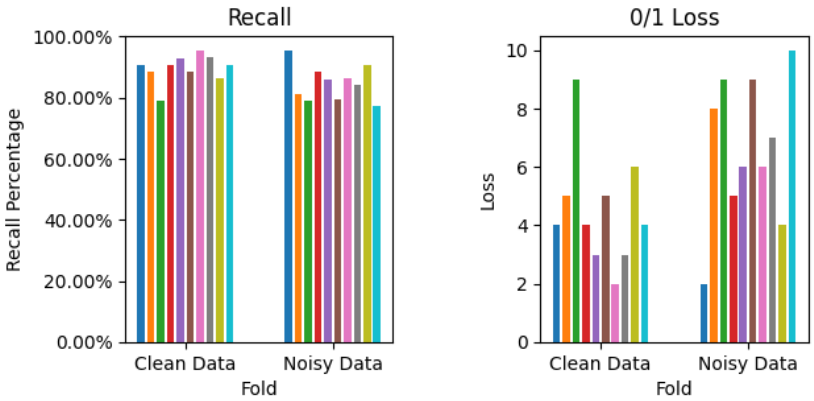

Figure 3: A graph of the recall and 0/1 loss on the *house-votes-84* data set

	Clean Data	Noisy Data
Recall	89.64 ± 4.55%	84.84 ± 5.72%
0/1 Loss (Average of 43.5 examples)	4.5	6.6

Table 3: A table displaying the average values of recall and 0/1 loss calculated using 10-fold cross-validation on the *house-votes-84* data set

3.4 *glass*

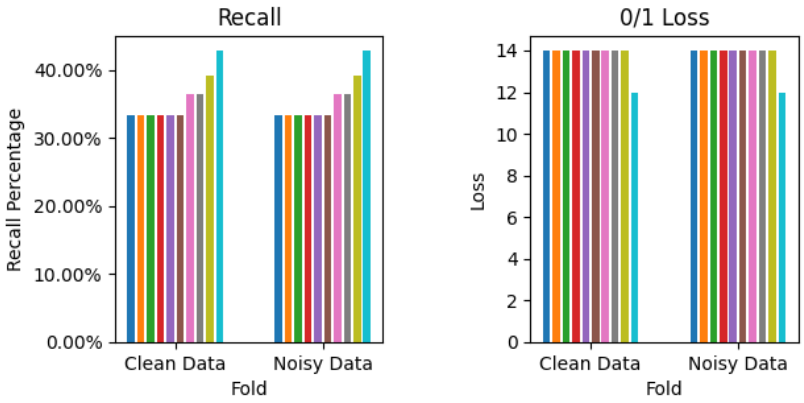

Figure 4: A graph of the recall and 0/1 loss on the *glass* data set

	Clean Data	Noisy Data
Recall	35.47 ± 3.28%	35.47 ± 3.28%
0/1 Loss (Average of 21.4 examples)	13.8	13.8

Table 4: A table displaying the average values of recall and 0/1 loss calculated using 10-fold cross-validation on the *glass* data set

3.5 soybean-small

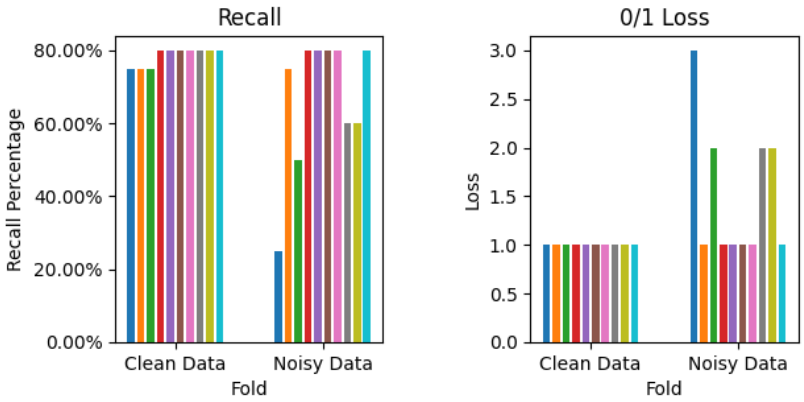

Figure 5: A graph of the recall and 0/1 loss on the *soybean-small* data set

	Clean Data	Noisy Data
Recall	78.50 ± 2.42%	67.00 ± 18.44%
0/1 Loss (Average of 3.5 examples)	1	1.5

Table 5: A table displaying the average values of recall and 0/1 loss calculated using 10-fold cross-validation on the *soybean-small* data set

4 Algorithm Behavior and Conclusions

The behavior of our algorithm confirmed both of our hypotheses to be true. The algorithm performed worse in a statistically significant way on both the *glass* dataset and the *soybean-small* dataset, and it performed significantly worse on the noisy versions of the datasets. While we expected the algorithm to perform poorly on the *glass* dataset, we didn't expect it to perform as badly as it did, with a recall of just $35.47 \pm 3.28\%$ and 13.8 0/1-loss out of 21.4 examples (refer to Table 4). It seems as though we underestimated how much the dimensionality increase and loss of specificity due to the discretization of continuous features would affect the algorithm's performance. When considering the algorithm's performance on *soybean-small*, it is hard to know how much of a role the possible conditional dependence between some of the features played because of how high the dimensionality is. Despite this, these factors definitely seemed to play a role based on the algorithm's performance on the dataset. As for the other three datasets, our algorithm performed very well. It performed especially well on the *breast-cancer-wisconsin* dataset, with a staggering recall of $97.14 \pm 1.78\%$, and a low 0/1-loss of 2.0 out of 69.9 examples on the clean data, and $93.28 \pm 3.08\%$ recall and 4.7 0/1-loss on the noisy data (refer to Table 1). Statistical

significance was proven using Student's t-test on the recall and 0/1 loss between the clean and noisy versions of each data set. We set the significance level to be 0.05. On the *breast-cancer-wisonsin* dataset, the p-value calculated for both the recall and 0/1 loss was 0.003. The *iris* data set's recall and 0/1 loss p-values were both 0.016. The *house-votes-84* data set's recall had a p-value of 0.053, and the 0/1 loss had a p-value of 0.051. Although these p-values are higher than our significance level, we believe this still shows statistical significance primarily due to the proximity of the two values to the significance level. Our p-value for both recall and 0/1 loss for the *glass* data set was 1.0. We believe this is an anomaly caused by our data pre-processing techniques and does not adequately represent the data and should not be taken into consideration. Binning and one-hot encoding allowed the data set to be very robust to noise, however, this robustness could simply be due to these techniques, not the actual data. The *soybean-small* data set had p-values of 0.066 and 0.038 for its recall and 0/1 loss respectively. While only the 0/1 loss p-value is below our significance level, we can disregard the results of this test as the variance is not equal. This is due to the small size of the data set. When comparing the two data sets we predicted would do statistically significantly worse than the other, we performed Student's t-test on the average recall and 0/1 loss across all three "good" data sets (*breast-cancer-wisonsin*, *iris*, and *house-votes-84*), and compared them to *glass* and *soybean-small* data sets individually. The p-value for the recall of the "good" values and the *glass* data set was $3.44e^{-21}$ and the p-value for the 0/1 loss was $1.29e^{-16}$. The recall of the "good" values and the *soybean-small* data set gave us a p-value of 0.007, and the 0/1 loss had a p-value of 0.07. While the p-value of the 0/1 loss is not below our significance value, this can again be explained by the lack of equal variance between the two data sets. Due to our high success rate of creating lower p-values than our set significance level, our hypotheses are shown to be true.

5 Summary

Our model implemented a Naive Bayes algorithm, and we tested it on five cleaned datasets. Our results confirmed both our hypotheses. First, we showed that a model using Naive Bayes will perform significantly worse on the *glass* and *soybean-small* datasets, as both of the models trained and tested on these data sets had a much lower recall than the other three. Second, we showed that our Naive Bayes model performs worse on noisy data due to its independence assumption and simplicity. This can be seen as both the recall and 0/1 loss of the noisy version of the data set is worse than its unaltered version. Generally, our model performed well. Across the clean and noisy versions of all 5 data sets, our average recall was $76.70 \pm 5.68\%$, and our average 0/1 loss was 5.14 on 30.66 examples. These values show that our Naive Bayes model performs well as a classification algorithm across many kinds of data sets, and can accurately predict a class using both numeric and categorical data.

Appendix A. Work Split

As per the Design Document, the work of this project was split by the following

- Data pre-processing: Alex
- K-fold cross-validation: Alex
- Learning and testing algorithms: Kellen
- Video: Both (Data pre-processing script by Alex, Model script by Kellen, Recorded by Kellen)
- Paper:

a. Abstract, Problem Statement, Experimental Approach and Program Design: Kellen

b. Results, Summary: Alex

c. Algorithm Behavior and Conclusion: Both