Machine learning lab assignment 2
Naive Bayes classifier

Kevin Serrano, *EMARO+, UNIGE*

***Abstract*—This lab report introduces the concept of Naive Bayes classifier and its performance on a small categorical dataset. The classifier will be implemented from scratch in MATLAB and the main steps on how the algorithm works are listed. Solutions to specific related problems such as zero likelihood probabilities or unbalanced training and testing subsets are addressed. Performance of the classifier is tested and pros and cons are discussed.**

***Keywords*—naive bayes, classifier, machine learning, stratified splitting.**

I. Introduction

The Naive Bayes algorithm is a classification technique based on Bayes' theorem assuming independence among predictors (features). It classifies data in two main steps [1]:

1) Training step: Using the training data, the method estimates the parameters of a probability distribution, assuming predictors are conditionally independent given the class.
2) Prediction step: For any unseen test data, the method computes the posterior probability of that sample belonging to each class. The method then classifies the test data according the largest posterior probability.

Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Bayes' theorem provides a way for calculating posterior probability $P(c|x)$ given other three probabilities [2]:

$$P(c|x) = \frac{P(c)\,P(x|c)}{P(x)} \tag{1}$$

where

- $P(x|c)$ is the likelihood probability
- $P(c)$ is the class prior probability
- $P(x)$ is the predictor prior probability

II. Approach

As previously mentioned, the algorithm has two main steps. We'll start off by describing the training step and move forward until reaching the prediction step.

A. Training step (fitting model)

It is essential to know which type of probability distribution our data corresponds to. Table I shows the dataset we'll be working with throughout this report, which is basically a prediction of whether players are able to play based on different meteorological conditions. First thing we notice is that the observations are categorical, that is, each variable can take only one of a fixed number of possible values (levels). Furthermore, each predictor has its own fixed set of possible values independent from the other predictors. This observations correspond to a **multivariate multinomial distribution**. A classifier for this type of probability distribution must perform the following steps:

1) Record the distinct categories represented in the observations of the entire predictor and transform from categorical to numerical representation (easier to handle in a computer).
2) Separate the observations by response class.
3) For each response class, fit a multinomial model using the category relative frequencies and total number of observations.

To better illustrate the last step, we'll provide an example using the *Outlook* predictor. Table II shows the prior probabilities, which can be computed by counting elements and then dividing them over the total number of observations. Likelihood, in the general case, can be computed by using

TABLE I: Weather categorical dataset

Outlook	Temperature	Humidity	Windy	Play
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	normal	FALSE	yes
rainy	mild	high	TRUE	no
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes

TABLE II: Prior probabilities table for *Outlook*

Outlook	No	Yes	
overcast		4	4/14 = 0.29
rainy	2	3	5/14 = 0.36
sunny	3	2	5/14 = 0.36
Total	5	9	
	5/14 = 0.36	9/14 = 0.64	

the following formula:

$$P(x_j = L|c) = \frac{n_{j|k}(L)}{n_k} \tag{2}$$

where

- L is the level
- c is the class
- x_j is the predictor (feature) j
- $n_{j|k}(L)$ is the number of observations for which predictor j equals L in class k
- n_k is the number of observations in class k

Then, as an example, we can calculate the probability of whether the players will be able to play given sunny weather.

$$P(\text{yes}|\text{sunny}) = \frac{P(\text{yes})\,P(\text{sunny}|\text{yes})}{P(\text{sunny})} = \frac{(9/14)(2/9)}{5/14} = 0.4$$

B. Managing zero probabilities

Due to the small size of the overall dataset, there are some predictor-level-class combinations that won't be present in the training set and will produce zero probabilities if we use equation 2. By having zero probabilities we're assuming that certain events will never occur, which is rather unrealistic. For dealing with zero probabilities we can apply **smoothing**. One of the most used forms of smoothing was invented by Laplace and it is often referred to as **additive smoothing**.

$$P(x_j = L|c) = \frac{1 + n_{j|k}(L)}{n_j + n_k} \tag{3}$$

where

- n_j is the number of distinct levels in the predictor

C. Testing step (prediction)

The probabilities previously computed will now be used to estimated an unknown output by using Bayes' theorem.

$$P(c_i|X) = P(c_i)\prod_{j=1}^{m} P(x_j|c_i) \tag{4}$$

where

- c is the class
- X is an i-th dimensional observation vector

Notice that we've excluded the effect of $P(x_j)$ since it only acts as a scalar factor and final results won't be affected by it.

For each observation in the testing set we'll compute 2 probabilities, one for each class. We'll then classify according to the highest probability.

$$\text{Play} = \begin{cases} \text{yes} & \text{iff } P(\text{yes}|X) > P(\text{no}|X) \\ \text{no} & \text{otherwise} \end{cases} \tag{5}$$

III. Experimental results

A. `naiveBayesKlassifier` class

The Naive Bayes classifier was implemented in MATLAB as a class called `naiveBayesKlassifier` as to mimic existing MATLAB or python libraries. I has two basic methods which correspond to the training and testing steps aforementioned.

- Properties
  - `predictorNames`: names of each predictor.
  - `targetName`: name of the target vector.
  - `classNames`: the different classes found in the target vector.
  - `numObservations`: total number of observations for the training set.
  - `categoricalLevels`: levels per predictor represented in categorical form.
  - `numericalLevels`: levels per predictor represented in numerical form.
  - `numericalPredictor`: numerical matrix containing the numerical representation of the training categorical set.
  - `numericalTarget`: numerical representation of the target vector.
  - `smoothing`: flag for smoothing (default = YES).
  - `priorProbs`: prior probabilities for each class.
  - `likelihood`: likelihood values per predictor per level (cell array of dimension k_classes × n_predictors).
- Methods
  - `fit(training_set)`: Receives a table object as an input and from this it sets up all the properties of the class
  - `predict(test_set)`: Receives a table object as input and uses the probabilities contained in in `priorProbs` and `likelihood` to compute the prediction vector. If a test target vector is also included the corresponding accuracy of the prediction is also computed.

When using the whole dataset as training set, we obtain the following:

```
naiveBayesKlassifier with properties:

predictorNames: {'Outlook' 'Temperature' 'Humidity'
'Windy'}
targetName: {'Play'}
classNames: [no    yes]
numObservations: 14
categoricalLevels: {{3x1 cell} {3x1 cell} {2x1 cell}
{2x1 cell}}
numericalLevels: {[3x1 double]  [3x1 double]
[2x1 double]  [2x1 double]}
numericalPredictor: [14x4 double]
numericalTarget: [14x1 double]
smoothing: 'Yes'
priorProbs: [0.3571 0.6429]
likelihood: {2x4 cell}
```

Going deeper in some of the priorities we find the mapping of categorical to numerical that yields table III.

TABLE III: Weather numerical dataset

Outlook	Temperature	Humidity	Windy	Play
1	2	1	1	2
1	1	2	2	2
1	3	1	2	2
1	2	2	1	2
2	3	1	1	2
2	1	2	1	2
2	1	2	2	1
2	3	2	1	2
2	3	1	2	1
3	2	1	1	1
3	2	1	2	1
3	3	1	1	1
3	1	2	1	2
3	3	2	2	2

TABLE IV: Likelihood cell array

	Outlook	Temperature	Humidity	Windy
no	$\begin{bmatrix}0.1250\\0.3750\\0.5000\end{bmatrix}$	$\begin{bmatrix}0.2500\\0.3750\\0.3750\end{bmatrix}$	$\begin{bmatrix}0.7143\\0.2857\end{bmatrix}$	$\begin{bmatrix}0.4286\\0.5714\end{bmatrix}$
yes	$\begin{bmatrix}0.4167\\0.3333\\0.2500\end{bmatrix}$	$\begin{bmatrix}0.3333\\0.2500\\0.4167\end{bmatrix}$	$\begin{bmatrix}0.3636\\0.6364\end{bmatrix}$	$\begin{bmatrix}0.6364\\0.3636\end{bmatrix}$

- `categoricalLevels` =

$$\left\{\begin{bmatrix}\text{overcast}\\\text{rainy}\\\text{sunny}\end{bmatrix}, \begin{bmatrix}\text{cool}\\\text{hot}\\\text{mild}\end{bmatrix}, \begin{bmatrix}\text{high}\\\text{normal}\end{bmatrix}, \begin{bmatrix}\text{FALSE}\\\text{TRUE}\end{bmatrix}\right\}$$

- `numericalLevels` =

$$\left\{\begin{bmatrix}1\\2\\3\end{bmatrix}, \begin{bmatrix}1\\2\\3\end{bmatrix}, \begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1\\2\end{bmatrix}\right\}$$

The `likelihood` cell array contains all the likelihood values arranged in a 3D structure where the first dimension corresponds to the k-th class, the second dimension corresponds to the n-th predictor and the third dimension corresponds to the level.

B. Splitting dataset into training and test sets

Splitting the available data into training and test subsets is a common practice in machine learning. The process of splitting is preferably done randomly, that is, selecting certain random observations for the training split and leaving the rest for testing. Suitable training:testing ratios range from 70:30 to 80:20

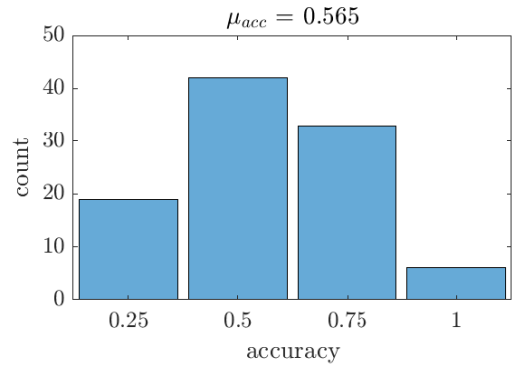
A MATLAB function was developed for this purpose.

```
[train, test] = train_test_split(data, train_size)
```

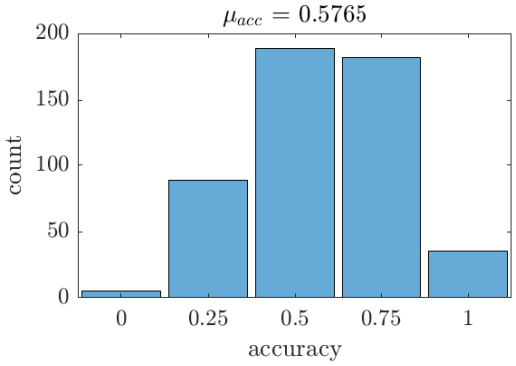
This function returns two vectors containing indices for the training and testing subsets accordingly. The parameter `train_size` must be a number between 0.0 and 1.0 and represent the proportion of the dataset to include in the training split.

C. Testing classifier using random splitting

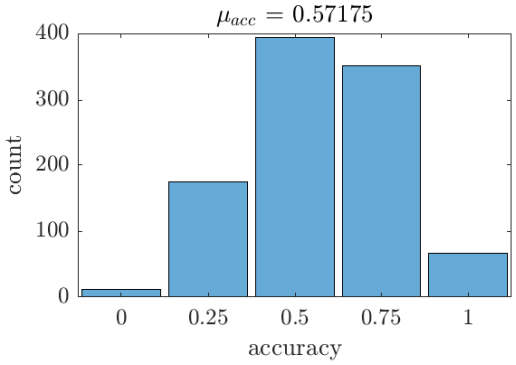
We can now proceed to run several tests by randomly splitting the dataset multiple times and run an accuracy analysis.



(a) 100 iterations



(b) 500 iterations



(c) 1000 iterations

Fig. 1: Performance analysis with random splitting

The train size is 70% of the overall data, which translates

into 10 observations for the training split and 4 for the testing split. We're interested in the average of accuracy but also in visualizing the results of the several tests conducted. A histogram plot is a powerful visualization tool for this.

Code 1: Simple script for testing classifier

```
rng default      % for reproducibility
iters = 100;     % number of iterations
nBK = naiveBayesKlassifier();

for i = 1:iters
  [train_idx, test_idx] = ...
    train_test_split(weather, 0.70);

  train_set = weather(train_idx, :);
  nBK.fit(train_set);

  test_set = weather(test_idx, :);
  [y_pred, accuracy, ~] = nBK.predict(test_set);
  % analyze results
end
```

Results shown in figure 1 show that the mean accuracy (μ_{acc}) lies around 57%. Moreover, from the histogram we observe that we have all possible error combinations, that is, we have cases where we get 0% accuracy going all the way up to 100%, specially when using a large number of iterations.

D. Stratified splitting

The results obtained so far are quite disappointing. Since there are no parameters to tune in a Naive Bayes classifier, we proceed to improve the way we split data. When using simple random permutation we're prone to highly unbalanced training and testing splits.

Stratification is the process of dividing members of a population into homogeneous groups before sampling. In machine learning, stratified sampling is used to preserve the percentage of samples for each class [3]. Given the properties of the weather dataset, if we want a 70:30 ratio we'll have:

- Training split:
  - 4 random observations of the *no* class
  - 6 random observations of the *yes* class
- Test split:
  - 1 random observations of the *no* class
  - 3 random observations of the *yes* class

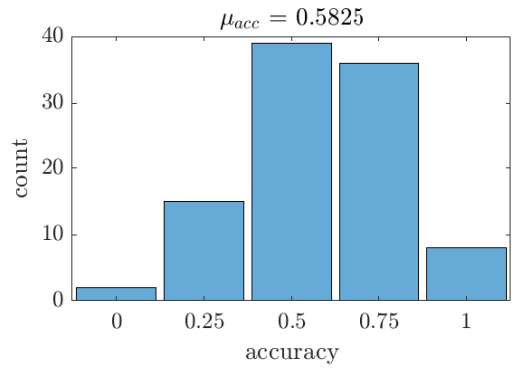
A similar function to `train_test_split` was created but now the output indices will respect the rule for preserving the percentage of samples for each class.

```
[train, test] = stratified_split(data, train_size)
```

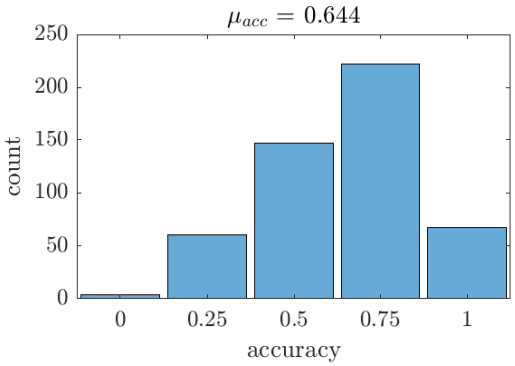
E. Testing classifier using stratified splitting

Same procedure as before was followed with the exception of the function used to split the data. Better results were achieved although we still have cases of 0% and 100% accuracy as seen in figure 2.

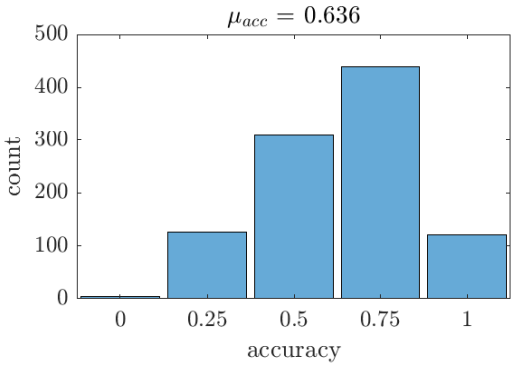
When using only 100 iterations the difference is not noticeable. Only when we increase the number of iterations we start appreciating a change in the histogram bar plot shape.



(a) 100 iterations



(b) 500 iterations



(c) 1000 iterations

Fig. 2: Performance analysis with stratified splitting

IV. Analysis

Two main experiments were conducted in order to test the implemented Naive Bayes classifier. Both of them gave poor results in terms of accuracy. This is due to the small size of the dataset. Only 14 examples of a dataset with 4 categorical predictors isn't enough to properly train the classifier. Taking into account that number of levels for each predictor, there are 72 possible level-predictor-class combinations of which we only have 14, leaving room for uncertainty.

TABLE V: Misclassified observations (red = highly misclassified, yellow = moderately misclassified, light yellow = rarely misclassified)

index	Outlook	Temperature	Humidity	Windy	Play
1	overcast	hot	high	FALSE	yes
	overcast	cool	normal	TRUE	yes
2	overcast	mild	high	TRUE	yes
	overcast	hot	normal	FALSE	yes
3	rainy	mild	high	FALSE	yes
4	rainy	cool	normal	FALSE	yes
5	rainy	cool	normal	TRUE	no
6	rainy	mild	normal	FALSE	yes
7	rainy	mild	high	TRUE	no
8	sunny	hot	high	FALSE	no
	sunny	hot	high	TRUE	no
9	sunny	mild	high	FALSE	no
10	sunny	cool	normal	FALSE	yes
11	sunny	mild	normal	TRUE	yes

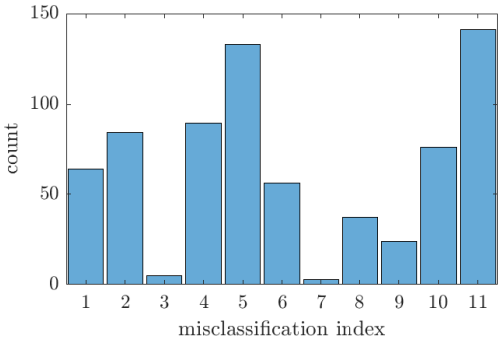


Fig. 3: Misclassification histogram

Another common practice is to look for outliers or observations frequently misclassified. An analysis on these shows that, when running with 500 iterations, 11 out of the total 14 observations get misclassified at least once. Table V and figure 3 complement each other to show which observations get the most misclassified. It is hard to explain exactly why they get misclassified since we're using stratified random splitting; however, misclassification may be caused by a high correlation of features in some training splits, leading to over inflating importance.

V. Conclusions

Naive Bayes classification is a powerful algorithm in the sense that it has almost no parameters to tune and has no variance to minimize; once it is implemented, there is little modification that can be made to the actual algorithm to boost its performance. Instead, we focused on improving the training split selection and gained a 12% increment in accuracy.

Machine learning, as the name suggests, is the process of learning from data. Therefore, data plays a key role in the performance of classification algorithms. In this report, the dataset contained around 20% of all the possible level-predictor-class combinations and we managed to get 65% average accuracy. Having access to more data could potentially improve the performance.

References

[1] T. M. Inc., "Naive bayes classification," October 2017. [Online]. Available: https://fr.mathworks.com/help/stats/naive-bayes-classification.html

[2] E. W. Weisstein, "Bayes' theorem." October 2017. [Online]. Available: http://mathworld.wolfram.com/BayesTheorem.html

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.