Bayesian classifiers in action with Scala

Pol Alvarez Vecino

June 28, 2017

Abstract

This project explores some basic probability-based classifiers, addresses some of the issues they present, and finally uses cross-validation to compare them against a batch of datasets.

1 Introduction

The project's goal is to implement three basic classifiers, Maximum a Posteriori, Naive Bayes, and Tree-Augmented Naive Bayes, and compare their performance. The implementation is done in Scala [1], a general purpose language with OO, functional programming, and strong static typing which runs in the JVM and is designed to overcome many of Java problems and critics. Scala was chosen because it is oriented towards functional programming which allows easier parallelizations which could be very useful for ML-related tasks.

To test the implementations performance we created an orchestrating main class called *PredictorsTest.scala*. The class runs the three different algorithms for each dataset twice: first with 10x10CV (see Subsection 5.2, and a second time with the whole dataset as train and test (see Section 5). It is also responsible of creating and updating the Confusion Matrix. To keep the code simple, this class defines a trait[1] called **classifier** that all classifiers must implement. It defines a getter function for the class variable values (required to build the confusion matrix) and the predict method which takes a sample as argument and returns a list of predictions. This trait is implemented by the three methods making their testing equal regardless the classifier.

The code is well structured and heavily commented so this document will only mention the more theoretical details of the implementation inviting the reader to check the actual code.

2 Max a Posteriori

Associated files:

[1]traits are scala's equivalent of interfaces

· **MaxAPosteriori.scala,** contains class implementing Maximum a Posteriori classifier

Maximum a Posteriori (MAP) is quite straight forward and can be easily understood checking the actual code. It is worth mentioning that for convenience during tests and comparisons, the conditional probabilities data structure has a default value of 0. This means that when encountering a key not seen during training, instead of crashing, the algorithm attributes 0 probability to the prediction. The variability introduced by this random predictions is very high when using 10x10CV heavy limiting its accuracy, nonetheless the results are added for reference.

3 Naive Bayes

Associated files:

· **NaiveBayes.scala,** contains class implementing Naive Bayes classifier

Naive Bayes classifier implementation structure follows the one provided in the lab. The probability formula is given by:

$$P(x) = P(\omega_k) * \prod_{j=1}^{d} P(X_j = x_j|\omega_k) \tag{1}$$

Naive Bayes predictor (NB) implements the basic classifier and counting methods to create the conditional probabilites but is improved with some more features. The null values are again treated with the 0 default value. However, having a null probability for an attribute turns the whole prediction to 0. To avoid that we use Laplace correction as described in subsection 3.2.

Using Scala functional programming features, the prediction method is parametrized with 3 functions: probability formula, probability filter, and accumulate function. The **probability function** specifies how to compute the probability given the numerator and denominator of the fraction, thanks to this it is possible to use simple probability or Laplace correction (described in next section). The **probability filter** and **accumulate function** are used to implement the simple or log probability, the filter is either $f(x) = x$ or $f(x) = log(x)$, the accumulation just describes if the probabilities shoud be summed of multiplied.

This combinations result in 4 execution modes:

1. Simple probability without Laplace correction
2. Log probability without Laplace correction
3. Simple probability with Laplace correction
4. Log probability with Laplace correction

3.1 Log probability

In order to avoid numerical precision problems when multiplying really small probabilities, the log probability formula 2 is used to predict new values. Logarithms are a strictly increasing function so we can use them to turn the product into a sum avoiding numerical issues while retaining the same prediction ordering. It is worth noting that when applying logs the results are no longer probabilities (can be bigger than 1).

$$P(x) = lnP(\omega_k) * \sum_{j=1}^{d} lnP(X_j = x_j|\omega_k) \tag{2}$$

3.2 Laplace correction

Laplace correction formula 3 prevents null probabilites (i.e. a 0 value in a conditional probability) from turning the final prediction into 0, for simple probabilities, or tend to infinity when using log probabilites. Roughly speaking, the correction applies a constant term to both numerator a denominator to prevent the 0 value. This value is the prior probability of each class (assuming they are all equal) which is defined by $1/V_k$ being V_k the number of classes. This prior probability is weighted with a factor p whose value should be optimized (for example with cross validation). In this paper, when using Laplace correction, the value of p is always 1.

$$P_L(X_j = x_j|\omega_k) = \frac{|\{x \in S_k \wedge X_j = x_j\}| + p}{|\{x \in S_k\}| + p * V_k}, p \in \mathbb{N} \tag{3}$$

4 Tree-Augmented Naive Bayes

Associated files:

- **TANaiveBayes.scala,** contains class implementing Tree-Augmented Naive Bayes classifier
- **UnionFindSet.scala,** union-find set data structure for Kruskal algorithm [2]
- **TANTree.scala,** contains class TANTree which holds all information required for prediction

Tree Augmented Naive Bayes [3] (TANaiveBayes or TANB) method tries to soften the independence assumption between variables of Naive Bayes with a tree-like dependency structure. In the structure, each variable represents a node of the tree and its conditional probability depends on the class variable and its parent node. The probability is given by the formula in equation 4.

$$P(x) = P(\omega_k) * P(X_{root}|\omega_k) * \prod_{j=2}^{d} P(X_j = x_j|\omega_k, X_{j.parent}) \tag{4}$$

Worth noting is that the root of the tree just depends on the class variable so it is out of the product (index starts at 2). As in Naive Bayes, the prediction was parametrized to be able to use log-probabilites and Laplace correction.

The tree is built in five steps:

1. Compute conditional mutual information for each different pair of attributes.
2. Build a graph with each attribute as a node and each edge annotated with their mutual information.
3. Build a maximum weighted spanning tree.
4. Turn the undirected tree to directed one by choosing a variable and setting all to flow out of it.
5. Link all nodes to a new node representing class variable.

Detailed information of the process in [3] and [4]. This implementation was more complex than previous classifiers so it was split into three classes.

The main class, TANaiveBayes, is responsible of implementing the classifier trait, and performing steps 1-3. Once the maximum weighted tree is built, it is fed to the TANTree class which directs the tree and stores all the information required to make a prediction. With this setup, TANaiveBayes acts as a constructor and wrapper for the TANTree class because it computes the required info for the tree and its method predict calls TANTree.predict. Figure 1 shows the dependencies tree generated for Votes classifier. Inspecting the actual dependencies is out of scope, but one quick look shows a meaningful relation: politicians voting to spend on el Salvador aid also voted for aid to Nicaragua (more details about the datasets in Subsection 5.1).

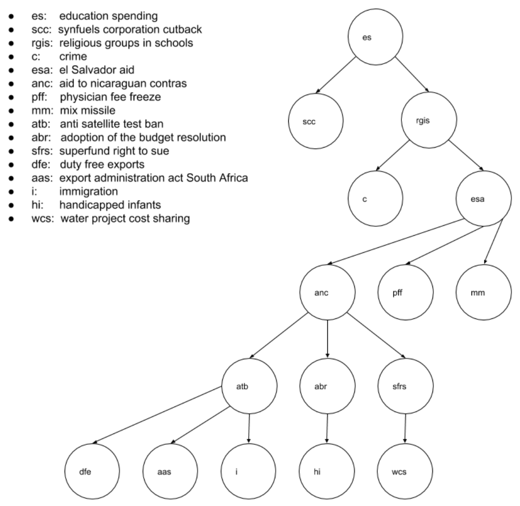
es: education spending
scc: synfuels corporation cutback
rgis: religious groups in schools
c: crime
esa: el Salvador aid
anc: aid to nicaraguan contras
pff: physician fee freeze
mm: mix missile
atb: anti satellite test ban
abr: adoption of the budget resolution
sfrs: superfund right to sue
dfe: duty free exports
aas: export administration act South Africa
i: immigration
hi: handicapped infants
wcs: water project cost sharing
es
scc
rgis
c
esa
anc
pff
mm
atb
abr
sfrs
dfe
aas
i
hi
wcs

Figure 1: Tree structure of TANaiveBayes classifier for dataset *votesTr.txt*. The goal is to predict if a politician is republican or democrat based on his budget spending votes.

5 Experiments

Associated files:

- **PredictorsTest.scala,** file orchestrating all the tests execution.
- **ConfusionMatrix.scala,** class containing the confusion matrix and diverse test metrics.

In order to reproduce the experiments performed Scala 2.11 is required. To compile and execute the tests issue the Listing 1. commands inside the folder *"./classifiers/src"*.

Listing 1: Compilation and execution commands to reproduce reported results

```
scalac −cp . ∗.scala
scala −cp . TestsExecutor
```

Note that even if the main file is PredictorsTest when executing the code we must use the object inside the file name: TestExecutor. As explained previously two kind of tests have been performed for each method. First method with 10x10CV, second with whole dataset as train and test.

5.1 Datasets

List of the datasets used and their classification objective:

- **cmcTr.txt,** contraceptive method used.
- **lensesTr.txt,** kind of lenses to wear (or none).
- **WeatherNominalTr.txt,** play tenis depending on the weather.
- **mushroomTr.txt,** predict if a Mushroom is edible or toxic.
- **votesTr.txt,** democrat or republican politican.
- **pimaTr.txt,** predict if an individual has diabetes [5]
- **titanicTr.txt,** surival to the Titanic accident.
- **germanTr.txt,** german language level.

All the datasets are formated following libsvm format with the exception that class value is the last one (instead of the first) and is named (classLabel:classValue). Lenses and WeatherNominal are small test datasets so they are not used when computing validation error with 10x10CV because too few samples per fold would be available and results would be highly biased (specially with MAP classifier which chooses randomly when the posterior probability for an attribute is null). Despite that, we have also executed all the datasets without validation (whole dataset for train and test) for reference and comparison.

5.2 Cross-validation

The validation method used in the experiments is 10-fold and 10 iterations cross-validation (10x10CV). For each method and dataset, data is shuffled and 10 cross-validation is performed. This is repeated 10 times, reshuffling data every time. The final model accuracy is the average of all partial ones.

As stated *Lenses* and *WeatherNominal* datasets were not used during 10CV because they were too small.

5.3 Contingency matrix for muticlass problems

Contingency matrix implementation was first translated from the python implementation seen in the labs. Afterwards, we added the accuracy, precision, specificity, and sensitivity metrics to it. However, the inclusion of the *Lenses* and *Contraceptive Method* datasets, which are multi-class problems, required to revise the contingency matrix implementation and reported measures.

The matrix extension to an arbitrary number of classes was trivial, reporting in a list all the possible combinations of predicted-truth labels. For the metrics, the sensitivity and precision become a per-class metric (measuring the true-positives against all the other classes). Mean sensitivity and precision of all the classes is also reported. Specificity metric is no longer used because it is computed the same way as the sensitivity (having 2+ classes means there are no longer negative labels, they are just the positive labels of one of the rest targets).

6 Results

Figure 2 shows the accuracy results of each classifier and dataset when using all the data as training and validation. Maximum a Posteriori has, as expected, perfect accuracy in all consistent datasets (same attributes imply same class) because it simply memorizes all combinations. The decrease in accuracy observed in NB is caused by the independence assumption between the variables. This same reason is the one causing TANaiveBayes to have improved accuracy over NB because it can represent and take into account some of the dependencies between the variables.

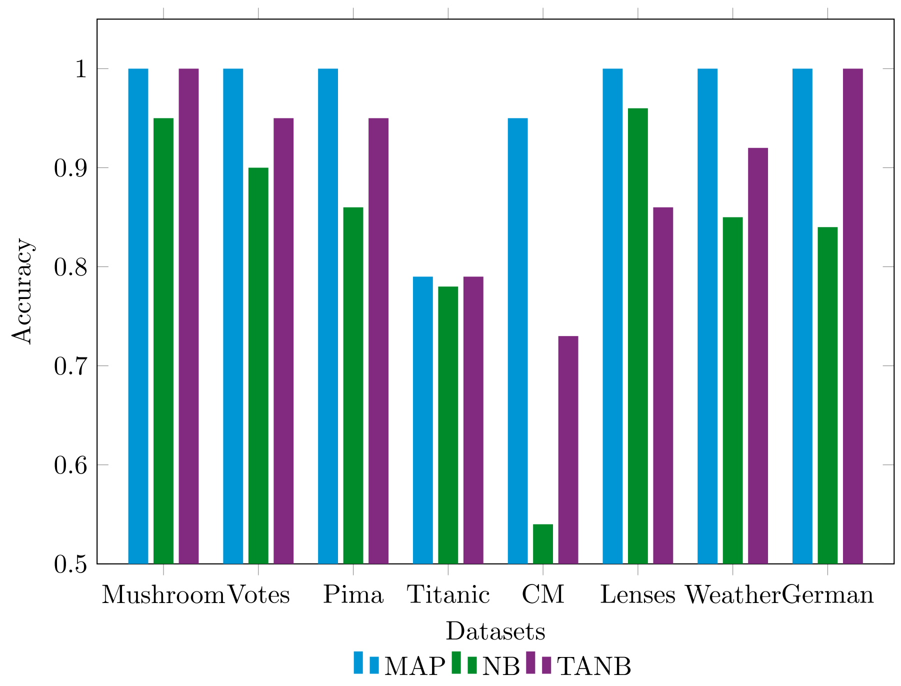

Figure 2: Accuracy for each classifier and dataset using the whole dataset for training and validation. MAP accuracy is 1.0 for all datasets which do not contain contradictory samples (same attributes different class). TANaiveBayes outperforms NB in almost all datasets. The performance increase is most probably related to the actual independence of the variables (NB assumption) and the tree ability to represent the dependencies.

Figure 3 shows the mean accuracy of the ten iterations with 10CV.

The big decrease in MAP accuracy, sensitivity, and precision is due to the increased null probabilites when leaving out observations that cause biased predictions (predicts always first class).

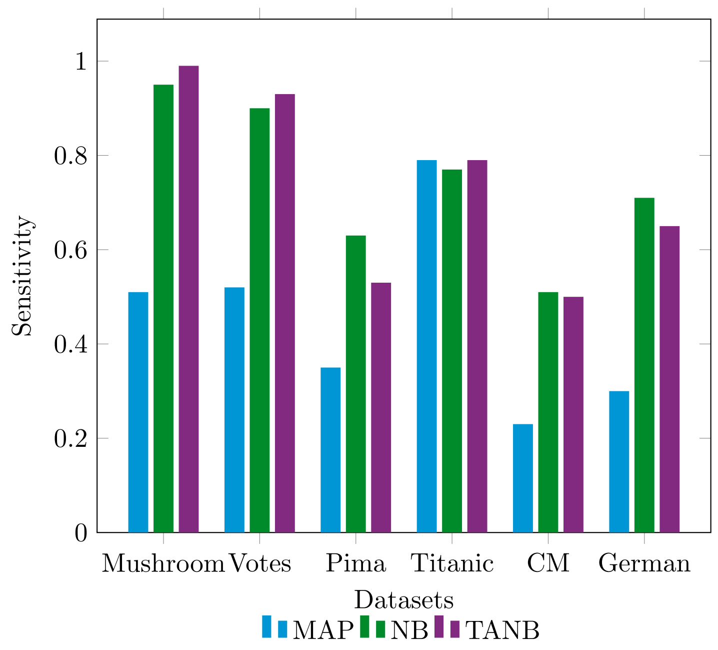

Figure 3: Accuracy for each classifier and dataset using 10x10CV

NB and TANB accuracies are now much closer and we do not see a consistent improvement when using the later. This could be because when using 10x10CV the TANB has more difficulties to learn the correct dependencies among variables. Another option is that, for some folds or iterations, the shuffling produces a model with an awful accuracy making the mean decrease notably. Finally, it could also indicate that for some datasets the variables are actually independent (or close) and trying to figure out dependencies between them is useless and just add noise.

Figure 4, and 5 show the mean sensitivity and precision of the first fold of the first iteration of 10CV for reference (actual values may vary a lot but it provides some insight). It is interesting to see that for German dataset TANaiveBayes classifier is far less sensible and precise than NB but its accuracy remains close enough. This can be caused by imbalance in the class of the samples because for a class the sensitivity may be really bad and for the other acceptable. This will lead to a bad mean. Despite that, if the class with better sensitivity occurs much more frequently the overall accuracy will be good enough. The same argument can be applied to precision, leading to the obseved acceptable accuracy with an awful precision and sensitivity. Again is worth noting that precision and sensitivities come from a single iteration and the reasoning could be biased.

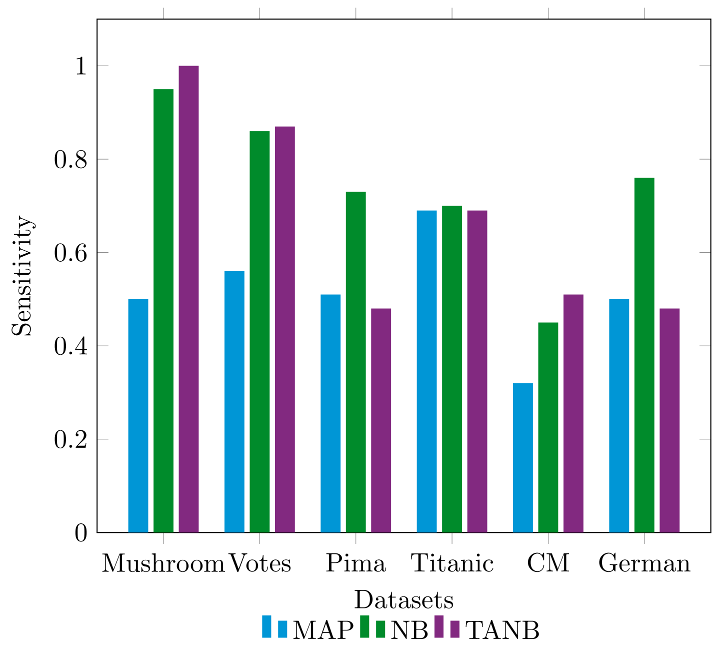

Figure 4: Sensitivity for each classifier and dataset using 10x10CV

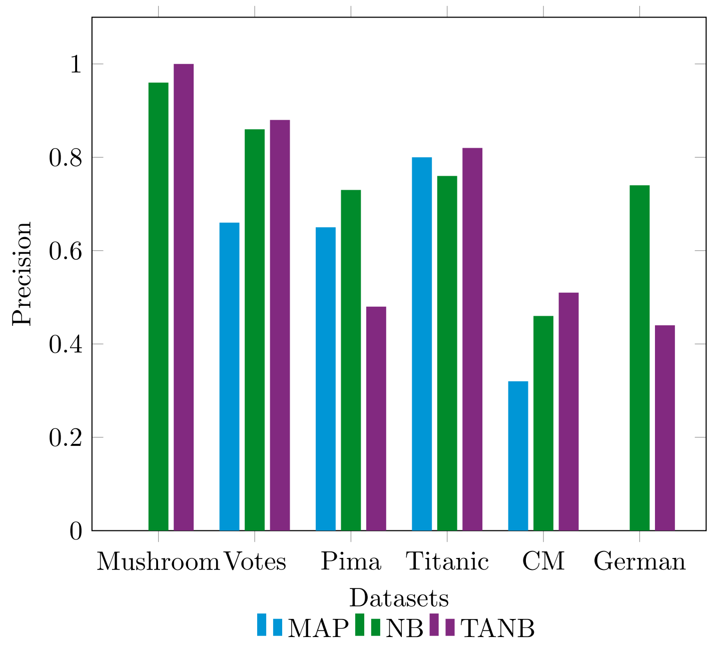

Figure 5: Precision for each classifier and dataset using 10x10CV

7 Conclusions and Future Work

The resulting implementation in Scala is quite clean (for being first time programming with it) and understandable despite being a bit bloated with long parameter types. The static typing system helped to avoid some problems that arise easily in python and also makes the code more legible when dealing with complex data structures such as the TANTree. The probability computation and conditional mutual information loops, which are the most computation-intensive parts, are separated methods which makes them easy to parallelize.

With respect to TANaiveBayes, the method had good results although not significative enough (could be the reason why they are not one of the most applied methods). As future work we could try to manually construct the dependencies tree (or at least tweak it) with the help of an expert of the data and see if that yields better results.

Finally, we have extended the confusion matrix, accuracy, and sensitivity for muticlass problems and decided how to average and report them when dealing with 10x10CV. Despite that, we have seen that the gathered results, even if useful, did not provide enough information to clear picture of the performance. It would be interesting to complement these numerical results with ROC curves and more detailed metrics (were averaging does not hide or bias the regular values).

References

[1] M. Odersky, "The Scala Language Specification," 2011.

[2] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, vol. 7, p. 48, feb 1956.

[3] W. Uther, D. Mladenić, M. Ciaramita, B. Berendt, A. Kołcz, M. Grobelnik, D. Mladenić, M. Witbrock, J. Risch, S. Bohn, S. Poteet, A. Kao, L. Quach, J. Wu, E. Keogh, R. Miikkulainen, P. Flener, U. Schmid, F. Zheng, G. I. Webb, and S. Nijssen, "Tree Augmented Naive Bayes," in *Encyclopedia of Machine Learning*, pp. 990–991, Boston, MA: Springer US, 2011.

[4] H. Padmanaban, "Comparative Analysis of Naive Bayes and Tree Augmented Naive Bayes Models," 2014.

[5] J. W. Smith, J. Everhart, W. Dicksont, W. Knowler, and R. Johannes, "Using the ADAP Learning Algorithm to Forecast the Onset of Diabetes Mellitus,"