A Purely Functional Naive Bayes Classifier

Arnau Abella

March 9, 2020

Abstract
This project provides a *simple* implementation for the naive Bayes classifier in the purely functional programming language Haskell. The project provides a library for modeling and trainining a naive Bayes classifier that can be used to predict, with a certain level of confidence, the label of a given arbitrary sample.

1 Introduction

1.1 Probability Model

The naive Bayes is a conditional probability model: given a problem instance to be classifier, represented by a vector $x = (x_1, \ldots, x_n)$ representing some n features (independent variables), it assigns to this instance probabilities

$$p(C_k|x_1, \ldots, x_n)$$

for each of k possible outcomes or *classes* C_k.

The problem with the above formulation is that if the number of features n is large or if a feature can take on a large number of values, then basing such a model on probability tables is **infeasible**. We therefore reformulate the model to make it more tractable. Using **Bayes' theorem**, the conditional probability can be decomposed as

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)}$$

In practice, there is interest only in the numerator, because the denominator does not depend on C and the values of the features x_i, so that denominator is effectively constant.

We can rewrite the above expression, using the **chain rule**, as follows

$$\begin{aligned} p(C_k, x_1, \dots, x_n) &= p(x_1, \dots, x_n) \\ &= p(x_1|x_2, \dots, x_n, C_k)p(x_2, \dots, x_n, C_k) \\ &= \dots \\ &= p(x_1|x_2, \dots, x_n, C_k) \ \dots \ p(x_n|C_k)p(C_k) \end{aligned}$$

Now, we apply the **naive** assumption that all features in x are **mutually independent**, conditional on the category C_k

$$p(x_i|x_{i+1}, \dots, x_n, C_k) = p(x_i|C_k)$$

Hence, the joint model can be expressed as

$$\begin{aligned} p(C_k|x_1, \dots, x_n) &\propto p(C_k, x_1, \dots, x_n) \\ &= p(C_k)p(x_1|C_k)p(x_2|C)p(x_3|C_k)\dots \\ &= p(C_k)\prod_{i=1}^{n} p(x_i|C_k) \end{aligned}$$

where $\propto$ denotes proportionality.

1.2 The classifier

Given the previous naive Bayes probability model, we can construct a classifier combining it with a **decision rule**.

A common decision rule is *MAP* decision rule (picking the hypoteshis that is most probable).

The corresponding classifier, a Bayes classifier, is the function that assigns a class label $y = C_k$ for some k as follows

$$y = \underset{k \in \{1,\dots,K\}}{argmax} \ p(C_k)\prod_{i=1}^{n} p(x_i|C_k)$$

1.3 Parameter estimation and event model

A class's prior $p(C_k)$ can be computed by calculation an estimate for the class probability from the training set.

To estimate the parameters for a feature's distribution, one must assume a distribution from the training set.

The assumptions on distributions of features are called the *event model* of the Naive Bayes classifier.

For discrete features, *Multinomial* or *Bernoulli* distributions are popular. For continuous features, it is typical to assume a *Gaussian* distribution.

2 Implementation in Haskell

Haskell is a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation.

The idea behind this project was to build a minimal executable to run naive Bayes over an input dataset to show that building such a predictor in Haskell was feasible. But, at the end, it finished being a library that allow a final user to parse a dataset, train a model with the given dataset using a validation technique such as cross validation, and predict unlabeled samples using the trained model.

The only drawback was that the Haskell ecosystem for Data Science is still immature [1]. You can verified it by searching at hackage [2] for the keyword "naive bayes". The only mature option for Bayesian probabilistic was *monad-bayes* [3].

2.1 The project

The project was organized as a Haskell **library** rather than an executable so the code can be extended and use by other people for free.

The main modules are the following:

- Parser
    - Parser.Csv
- Distrib
- Validation

- Classifier
  - Classifier.Dummy
  - Classifier.NaiveBayes

Additionally, there is the Lib module that reexports all the previous modules so you don't need to export one by one.

Apart from the library, the project also includes an **executable** to run a couple of test data sets to experiment a bit with the library.

In the following sections, we are going to visit each module and its implementation.

If you are not interested in the implementation, jump to the next section **??**

2.1.1 Parser

The module *Parser* contains a collection of parser for common storage format such as Comma-separated values (CSV).

For now, the only implemented parser is for CSV. This parser uses the library cassava [4] to levarage the work of parsing a csv file.

```
1  parseCsvFile
2      :: Csv.FromRecord a
3      => FilePath -> HasHeader -> ExceptT String IO [a]
4  parseCsvFile fp hasHeader = do
5    liftEither (checkExtension fp)
6    lbs <- liftIO (LBSC.readFile fp)
7    liftEither $ fmap V.toList (Csv.decode hasHeader lbs)
```

2.1.2 Distrib

In the module *Distrib* you will find common distributions such as Uniform, Gaussian, Gamma, Beta, Categorical, Geometric, Poisson, etc.

```
1  data Distrib where
2    Distrib :: ContDistr d => d -> Distrib
```

For now, you will only find the **Normal distribution** for numerical variables and the probability density function of it.

```haskell
data Distrib where
-- | We don't know much about our numeric variable
-- so we will assume normallity to simplify the implementation.
compNormalDistr :: [Double] -> Distrib
compNormalDistr xs =
  Distrib (normalDistr mean' stdDev')
    where
      v = VU.fromList xs
      mean' = Statistics.mean v
      stdDev'  = max 0.1 (Statistics.stdDev v)

-- | Probability Density Function
pdf :: Distrib -> Double -> Probability
pdf (Distrib d) = Probability . density d
```

2.1.3 Validation

The module *Validation* implements model validation techniques such as simple sample validation, cross validation, bootstrapping validation, etc.

The current implementation only includes cross validation.

```haskell
-- | X-Validation testing algorithm
--
-- Notice that every subset of <training, test> is different
-- and generates a new classifier, so the score is not for a
-- single classifier, but for many.
crossValidation
    :: forall c. Classifier c
    => [(Sample, Class)] -- ^ All Samples
    -> Proxy c
    -> Score
crossValidation allSamples _ =
  let groups = leaveOneOutN (groupBy' 10 allSamples)
  in foldMap test groups
    where
      test (trainingSet, testSet) =
        let classifier = (train trainingSet) :: c
         in foldMap (predictAndGetScore classifier) testSet

```

```
19        predictAndGetScore classifier (sample, clazz) =
20          let predictions = predict sample classifier
21              cmp = (\(_, p1) (_, p2) ->
22                      (_probability p1) `compare` (_probability p2))
23              (pClazz, _) = maximumBy cmp predictions
24           in
25              if pClazz == clazz then hit' else miss'
```

2.1.4 Classifier

In the module *Classifier* the user will find the basic interface of all classifiers

```
1  class Classifier c where
2    -- | Given a training set returns a trained classifier
3    train :: [(Sample, Class)] -> c
4    -- | Given a random sample, predict a class
5    predict :: Sample -> c -> [(Class, Probability)]
```

The following classifier are currently implemented

- A random classifier on *Classifier.Dummy*
- A random classifier on *Classifier.NaiveBayes*

The Naive Bayes classifier has the following structure

```
1  data NaiveBayes = NaiveBayes
2    { labels  :: [Class] -- ^ All labels
3    , priors  :: Map Class Probability
4    , distrib :: Map Class (Map Index Distrib)
5    }
6
7  instance Classifier NaiveBayes where
8    train :: [(Sample, Class)] -> NaiveBayes
9    train = trainNaiveBayes
10   {-# INLINE train #-}
11
12   predict :: Sample -> NaiveBayes -> [(Class, Probability)]
13   predict = predictNaiveBayes
14   {-# INLINE predict #-}
```

The source code can be found on github [5].

2.2 Project testing

2.2.1 Compiling the project

Before running the project, we need to compile it.

I made the effort to simplify compilation and dependency management using nix [10]. Nix is a purely functional package manager that, among other things, allow us to compile Haskell projects in a pure deterministic way.

In order to compile the project you need to follow the following steps:

1. Install nix in your local machine

   ```
   $ curl https://nixos.org/nix/install | sh
   ```

2. Compile the project using nix

   ```
   $ nix-build
   ```

3. Run the executable on the input datasets

   ```
   $ ./result/bin/naive-bayes-classifier
   ```

The compilation steps are only for UNIX systems.

2.2.2 Running the classifier

To test the implementation I added two data sets randomly selected from Kaggel [9]:

- Wine quality [7]
- Wine origin [8]

The results are the following:

```
---------Dataset: wine-quality
#Classes: 6
#Total of samples: 1599
Score {hit = 878, miss = 721, total = 1599}
Accuracy: 54.97 %

---------Dataset: wine-origin
#Classes: 3
```

```
9  #Total of samples: 178
10 Score {hit = 170, miss = 8, total = 178}
11 Accuracy: 95.50 %
```

The accuracy for the first dataset is low, 54.97%.

The accuracy for the second dataset is very high, 95.50%.

I did not spent time analyzing why Naive Bayes performs poor on the first dataset but, in contrast, performs extremly well on the second one as it was not the objective of this delivery but it would something nice to do as a future project.

3 Conclusion

This library is a proof of concept that Naive Bayes classifier can be easily coded in Haskell in about 700 lines of code with the benefits of static correctness, friendly and robust refactoring, high level abstractions, high performance, ... Haskell is a suitable language to build such a probabilistic models.

4 Future work

Regarding to this project, there is a lot of margin for improvement. The current implementation of the library is missing a well defined structure which leads to non composable code that is ready hard to extend. Also, the library is missing a lot of features such as other parsers, distributions, validation and classifiers.

Regarding to the Haskell ecosystem, there is a lot of work to be on the Data Science domain to be at the same level as other languages such as R or Python. This project is a very little step towards a more mature haskell ecosystem. Early adopter companies such as tweag.io are working hard to bring more libraries to the ecosystem and make Haskell a mature enough place to start working on Data Science in it.

References

[1] Gabriel Gonzalez. 2020. State of the Haskell ecosystem. https://github.com/Gabriel439/post-rfc/blob/master/sotu.md

[2] hackage: The Haskell Package Repository. https://docs.haskellstack.org/en/stable/README/

[3] monad-bayes: A library for probabilistic programming. https://hackage.haskell.org/package/monad-bayes

[4] cassava: A CSV parsing and encoding library. https://hackage.haskell.org/package/cassava

[5] naive-bayes-classifier https://github.com/monadplus/naive-bayes-classifier

[6] Kaggel datasets https://www.kaggle.com/datasets

[7] Wine quality https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009#winequality-red.csv

[8] Wine origin https://archive.ics.uci.edu/ml/datasets/Wine

[9] Kaggel datasets https://www.kaggle.com/datasets

[10] Nix: a purely functional package manager https://nixos.org/nix/