A Functional Approach To The Classification Problem

Zmicier Zaleznicenka, #4134575

Delft University of Technology
Faculty of Electronic Engineering, Mathematics and Computer Science
D.V.Zhaleznichenka@student.tudelft.nl

Abstract. This report describes the project completed by the author during IN4355 Functional Programming course at TU Delft. The project goal was to implement two classification algorithms in a functional style using Python programming language.

1 Introduction

The research topic of the project discussed in this report is application of functional programming techniques to the classification algorithms. The classification problem is one of the well-known statistical challenges and is being studied for several decades already. In statistical studies, classification means identifying a subset of categories from a category set to which a certain instance belongs. Classification is usually based on the existent training set with a number of instances already associated with the categories from a category set. Correlation between the instances and categories is defined by analyzing the quantifiable properties (features) of the subject instances[11].

Many different classification algorithms exist with each of them having different properties. These algorithms can themselves be classified to a number of categories, such as linear classifiers, decision trees, neural networks and more. In the scope of this project we will discuss the implementation of a naive Bayes classifier which is a rather simple representative of the linear classifiers and k-nearest neighbors algorithm belonging to the kernel estimation classifiers.

Classification algorithms are heavily used nowadays in many fields. The application domains where the classifiers are successfully applied are pattern recognition, natural languages processing, internet search, computer vision and more. The unstoppable growth of online data sets which is observed in recent years (also known as data deluge) forced the researchers to develop new efficient data mining techniques to successfully process these data sets. For data mining, classification algorithms are also often of utmost importance.

This report discusses the applicability of certain functional programming techniques to the implementation of two classification algorithms using Python programming language. The contribution of the completed project is the investigation on the applicability of Python as general-purpose functional programming

language and the workability of functional programming techniques for data classification purposes. The source code of the project as well as the latest version of this paper are publicly available at Github[1].

The rest of this paper is organized as follows. In Section 2 we discuss the algorithms implemented in the project. Section 3 gives an insight into the existent approaches in implementing data classifiers used in industry and academia. Section 4 is devoted to the description of a project implementing two classifiers in a functional style. Section 5 contains the evaluation of the developed implementation and its comparison with the other existent software packages. In Section 6 the findings and conclusions are placed.

2 Description of the implemented algorithms

To reach the goals set for this project, it was decided to implement two well-known classification algorithms, namely naive Bayes (NB) and k-nearest neighbors (KNN). Both of these algorithms are relatively simple and are often used in different domains for the text classification, pattern recognition and other tasks. However, these classifiers have different properties. While NB algorithm is characterized as having low variance and high bias, KNN algorithm has high variance and low bias. From this it follows that NB should perform better for smaller data sets and KNN should be better for the large ones. Also, it is needed from the data set to have instances distributed in correspondence with any of the known density functions, i.e. normal distribution density function for NB classifier to show good results. NB classifier also requires feature independence from the data set. For KNN classifier these restrictions do not hold. Apart from these differences, there are some others that will not be covered in this document as it does not have a goal to compare the classifiers themselves.

2.1 Naive Bayes classifier

Naive Bayes is considered to be the simplest classification algorithm possible. However, it has good execution speed and adequate error rate for many classification tasks. As its name implies, this classifier is based on the naive Bayes probabilistic model which assumes that the instance properties (features) are independent given class, that is, $P(X|C) = \prod_{i=1}^{n} P(X_i|C)$ [8].

The implementation of a classifier usually consists of training and classifying stages. For NB classifier, at the first stage the instances from a training set are split into the classes. For these instances we know explicitly to which class each instance belongs. At the second stage, we take the features of the instance from a testing set and find the probabilities of it belonging to all of the classes in order. The highest value of these probabilities is defined by the formula $argmax(P(C)) * \prod_{i=1}^{n} P(X_i|C)$ and the instance is classified as belonging to the category having this highest probability value [6][8].

[1] https://github.com/dzzh/IN4355

2.2 K-nearest neighbors classifier

K-nearest neighbors classifier belongs to the set of kernel estimation algorithms and approaches the classification problem by locating the instances that are closest to the instance to be classified in a feature space[2][10].

The algorithm idea is the following. Assume we have a set of classified instances defined by their feature vectors and located in the feature space. To classify an instance in the same space, we should find K of its nearest neighbors using the distance function and choose the most frequent class of these neighbors using majority voting.

The idea to classify the instances based on the similarity with the instances lying in the close proximity in the feature space is significantly different from the classification based on Bayes probability model discussed earlier. For example, there is no explicit training stage in KNN algorithm. It is only required to store the feature vectors and the respective class labels in the memory before initiating the classification process.

Despite this algorithm seems to be rather straightforward and easy to implement, it leaves much room for improvement and many researchers work on adjusting its properties for better performance on different data sets. The error rate of the algorithm depends significantly from K value and selection of distance function. Many different distance functions are designed and described in scientific literature.

For this project, Euclidean distance was used as distance function. However, one important adjustment was made to the data sets prior to submitting them to the classifier to let the distance function in KNN classifier and probability formulas in NB classifier to work correctly. To address the co-existence of continuous and discrete features in some datasets they had to be normalized.

2.3 Normalization

Feature vectors in some datasets used to test the correctness of the algorithm shared both continuous (numerical) and discrete (categorical) features and there was a need in the distance function that would give adequate results for these mixed setups. To solve this problem, both continuous and discrete values were normalized prior to the classification. For continuous values, z-score normalization was applied. For different discrete values, 1 was taken as the distance value. For same discrete values, 0 was taken.

This normalization showed rather good results as can be seen in the evaluation section. However, these results can be improved further by optimizing the normalization process. The problem of normalization for mixed datasets was diligently studied in [9] and a better solution to it was proposed. The implementation of the mentioned approach for this project was left for future work. It is interesting, however, that this solution was proposed only recently. The authors of the mentioned paper report that not much work has been done beforehand to

[2] N-dimensional metric space for the instances having N features

address the normalization of the mixed feature vectors which is rather important for data mining and other application domains and does not seem to be an overly complex problem.

3 Existent approaches to building the classifiers

Since the classification topic is important for many real-world tasks, there are many software packages that provide their customers a number of implementations for the classification algorithms. As the logic behind many of the classifiers is rather simple, implementations of most of such algorithms exist for virtually every programming language used in production. Classification algorithms are studied in academia and often used as the lab assignments. Their implementations exist both in general mathematical and statistical packages (SPSS, R, Mathematica, SAS, Statistica etc.) and as specialized classification software, proprietary and open-source (BayesiaLabs[3], BAYDA[4], GPU-FS-kNN[2] etc.). Most of these implementations are written using general-purpose iterative programming languages like Java or C/C++. However, there are the implementations of naive Bayes and k-nearest neighbors algorithms in functional programming languages. Thus, Andrew Bienert discusses an implementation of binary NB classifier in F# in [3]. Aleksovski et al. in [1] discuss an implementation of KNN classifier in Haskell.

4 Application structure and implementation

The discussed project was implemented as a command-line application reading data from files and reporting to the output stream. The application structure consists of the main program responsible for I/O and normalization, a class container and classifiers code.

Internally, all the instances are represented simply as lists of their features. This makes it possible to perform operations on them in a functional way.

4.1 Main program

Main program accepts a number of parameters allowing to adjust certain application properties. It is located in `machine_learning.py` file.

The only positional argument accepted by the main program is the name of the dataset. Application parser was written to support the data sets from UCI Machine Learning Repository[4]. To work with any of the data sets from this repository, it has to be downloaded and placed into its own directory within `data_sets` directory in the project root. When the main program is launched, it accepts `.data` file from the named directory as the training set. If the program

[3] http://bayesia.com
[4] http://archive.ics.uci.edu/ml/

also finds a `.test` file in the same directory, it is accepted as the testing set. Otherwise, the testing set is generated as a subset of the training set.

Main program assumes NB being a default classifier. To choose KNN classifier, it has to be called with `-c knn` argument. `-k <value>` argument is used to set K value for this classifier. `-p <percent>` specifies percentage of values to be moved to the testing set from the training set if `.test` file is not found.

4.2 Class container

The program has two supplementary classes, namely `Clazz` and `Feature` that are placed in `classes.py` file. These classes were written to support data normalization and probability calculations for naive Bayes classifier. These classes are not used in the implementation of KNN classifier.

`Clazz` class is a container for the instances from a training set belonging to one certain class (or category, or outcome). It keeps track of the outcome statistics, namely of the number of instances belonging to it and the number of matches during the classification, and a list of `Feature` objects containing data of the matched instances. Also, it is responsible for providing NB classifier with the probability that the instance being classified belongs to this class.

`Feature` class contains a list of values for a given feature for all the instances. Thus, each `Clazz` object has a list of `Feature` objects with their number equal the number of features in the given data set. Also, `Feature` class has a number of mathematical functions operating on the list of values, like `mean()`, `standard_deviation()` and `variance()`. The other methods in this class analyze the list of features to contain only continuous values and calculate prior probability for NB classifier. The prior probabilities are used by `Clazz` class to calculate class probabilities.

For normalization purposes, main program has an instance of `Clazz` class where all the instances from the training set are stored. NB classifier has as many `Clazz` instances as the number of outcomes in the training set.

The existence of these supplementary classes generally violates the principles of functional programming. It was decided to introduce them due to the need to save the state between the certain stages of program execution. First of all, for normalization there was a need to save all the instances from a training set in a data structure. Secondly, there was a need to gather statistics, i.e. number of hits during the classification. Despite it was possible to implement this functionality using only the sequences and functions on them, that would significantly complicate the structure of the code.

The classifiers described below are also organized as classes though in this situation their structure is mostly dictated by the need to separate their functionality and let the main program to call them interchangeably, not by the need to use object-oriented programming techniques. Classifier classes can easily be rewritten as standard Python modules containing only function calls.

4.3 Naive Bayes classifier

The most challenging part of NB classifier, namely probability calculations, is implemented in the class container. The classifier itself is a very simple class gathering classification statistics and holding a list of `Clazz` objects. After its initialization, the main program analyzes the outcomes that exist in the training set and calls `add_class(name)` method to add a new instance of `Clazz` class to the classifier.

The training process is implemented in `train(instance)` method. The classifier adds the instance to one of its classes based on the known instance outcome.

Classification logic is implemented in `classify(instance)` method. For all of the outcomes the classifier calculates their class probability and decides on the highest probability. Then the result is compared with the real outcome of the instance and if they match, the classifier increments hits number.

4.4 K-nearest neighbors classifier

KNN classifier contains a list of instances and a list of neighbors as well as some statistics-related variables. For this classifier it is needed to keep all the instances from a training set in memory as during the classification process there is a need to measure the distance between the instance from a testing set and all the instances from a training set. In pure functional programming it would be logical not to keep the instances of the training set in the variable but rather read them from a file (say, using a generator function) but it would lead to the drastic performance drop.

Training process for KNN classifier is very simple: we just add an instance from a training set to the list of instances.

Classification process is implemented in `classify(instance)` method. Its logic is the following. We start with an empty list of neighbors and process all the instances from the training set in a loop. At each iteration, we measure the distance between the two instances. If this distance is less than the distance to the farthest neighbor, current instance is added to the list of neighbors. If the list of neighbors is already full, the farthest neighbor is deleted from the list.

After all the instances from the training set are processed, we use majority voting to find the most probable category of the instance being classified and compare it with the real instance outcome. If they match, we increase hits number.

4.5 Functional programming techniques used

While implementing the project, a lot of attention was paid to develop its architecture and write the code in concordance with the principles of functional programming. The application of these principles for Python programs is discussed in [5][7].

Partial functions. In Python, all functions are first-class objects, which allows to use them as arguments for higher-order functions. One of the most used higher-order functions in Python is `functools.partial()` that allows to construct variants of existing functions. In the discussed project, two partial functions were created from a general `read_file()` function used in main program for file input. The implementation of the partial functions allowed to simplify their calls while reading training and testing sets from the different files.

Listing 1.1: Application of partial functions.

```
from functools import partial

read_training = partial(read_file, file=file_prefix+'.data',
    is_training=True)
read_testing = partial(read_file, file=file_prefix+'.test',
    is_training=False)

training_set = read_training()
if os.path.exists(file_prefix + '.test'):
    testing_set = read_testing()
```

Another higher-order function that is often used in Python is `start_new_thread()` used in multi-threaded programs. However, it has not been used in this project due to the lack of time. However, parallelizing the classification function can improve the program while running it at multi-core machines.

List slices and comprehensions. Python provides rich functionality for operating with iterators and list comprehensions allow to perform different operations on iterable variables. Listing 1.2 shows the core logic of the program which is implemented in just four lines of code using the list comprehensions.

Listing 1.2: Core logic of the application.

```
#Precompute data for normalization
statistics = Clazz(len(training_set[0])-1,'training_set')
[statistics.add_match(instance) for instance in training_set]

#Train and classify
[classifier.train(instance) for instance in normalize(
    training_set)]
[classifier.classify(normalize_instance(instance)) for instance
    in testing_set]
```

List slices are also an important in Python as they provide easy access to the list partitions. In this project, the slices were used to get access to the instance outcome. As the outcome was stored as the last feature in the list, it was accessed as `features[-1]`. Another usage of the slices was while splitting the training set into training and testing sets.

Listing 1.3: Splitting training set.

```
if not os.path.exists(file_prefix + '.test'):
    offset = get_split_offset(args.percentage, training_set)
    testing_set = training_set[offset:]
    training_set = training_set[:offset]
```

Lambda expresiions. In this project, lambda expressions were used as predicates for `filter()` and `reduce()` functions. Two examples containing lambda expressions are discussed in the consequent paragraph.

`map()`, `filter()`, `reduce()` functions. These built-in functions were often used by Python 1.x programmers to perform operations on iterable data. However, after introduction of list comprehensions in Python 2.0, the first two of these functions became somewhat obsolete as the comprehensions duplicate their functionality and have more concise syntax. Thus, `map()` function was not used in this project, `filter()` was used only once in prior probability computation for discrete features (see Listing 1.4) instead of a list comprehension with `where` clause.

Listing 1.4: Usage of filter function.

```
if not self.is_continuous():
   #straightforward computation for string values
   return len(filter(lambda x: x == instance, self.instances)) \
        / float(num_matches) + SAMPLE_CORRECTION
```

`reduce()` function was used more often to perform data aggregation over iterables. E.g., the following expression was used to compute mean for a list of feature values: `reduce(lambda x,y: x+y, self.instances)/ len(self.instances)`.

It is worth to notice that `reduce()` function was not always working as expected and had sometimes to be substituted by the loops. For instance, this function was initially used to determine whether a list contains only float values: `reduce(lambda x,y: type(x)is float and type(y)is float,self.instances)`. For some reason, this function was always returning `False`. It was substituted by the following loop that works as expected.

Listing 1.5: A loop determining whether the list contains only floats.

```
for instance in self.instances:
    if not type(instance) is float:
        return False
    return True
```

Another impression from a reduce function was that its syntax is overly complex and is often hard to understand after the code is written. The usual loops are generally more understandable.

5 Evaluation

The application performance was tested on three data sets from UCI repository. Their characteristics can be found in a Table 1.

Name	**# instances**	**Features type**
iris	150	continuous
car	1728	discrete
adult	32561+16281	mixed

Table 1: Data sets characteristics.

The testing results for these data sets can be found in Table 2. For *iris* and *car* data sets the testing set was derived from the training set (percentage of data left for training is shown in column 3), for *adult* data set there was a testing file available with 16281 instances.

Data set	**Classifier**	**% training**	**Execution time**	**Error rate**
iris	NB	90	1 sec	0-6.7%
iris	NB	80	1 sec	0-10.0%
iris	KNN-1	90	1 sec	0-6.7%
iris	KNN-1	80	1 sec	0-10.0%
iris	KNN-3	90	1 sec	0-6.7%
iris	KNN-3	80	1 sec	0-10.0%
car	NB	90	1 sec	12.8-15.6%
car	NB	80	1 sec	10.1-17.1%
car	KNN-1	90	3 sec	16.2-29.5%
car	KNN-1	80	4 sec	19.1-23.4%
car	KNN-3	90	3 sec	6.9-13.3%
car	KNN-3	80	4 sec	10.7-14.7%
adult	NB	-	93 min	20.7%
adult	KNN-1	-	149 min	19.6%
adult	KNN-3	-	160 min	16.9%

Table 2: Testing results.

For *iris* and *car* data sets we have various error rates as the testing set is randomly generated from the training set. The difference in results is due to this randomness. For these data sets, all the calculations were done for five times and the maximum and minimum error rates observed are reported in the paper. For *adult* data set (which is a well-known Census Bureau data set), the testing set was always same as it was taken from the accompanying testing file.

The testing results show that the algorithms were implemented correctly. KNN-1 classifier (KNN with K=1) has the worst results and KNN-3 (KNN with

K=3) has the best results as its results do not depend on the data distribution pattern and feature independence as requested for correct NB operations. NB results lie in between. The error rate for both classifiers can be decreased if applying the normalization procedure as described in [9]. The reference implementation[5] of NB classifier gives 16.1% error rate, KNN-1 gives 21.4%, KNN-3 gives 20.4%. The results with 80% of instances in the training set are worse than with 90% of instances as the classifiers work better with more training data available.

6 Conclusion

Python programming language provides good support of functional programming techniques and it is possible to implement rather large programs in this particular style with Python. In comparison with purely functional languages, i.e. Haskell, Python has an advantage that it allows to write the programs in different styles, e.g. mix object-oriented, procedural and functional programming approaches in a single program. If using this language feature with a good understanding of the benefits of each programming style, it may help the developers to write clean and understandable code, and do it fast. A disadvantage of doing functional programming in Python is that some of its language instruments to support functional approach have difficult syntax and the expressions written with them are not easy to understand afterwards, if comparing with Python iterative programming constructs.

References

1. D. Aleksovski, M. Erwig, and S. Dzeroski. A functional programming approach to distance-based machine learning. In *Conference on Data Mining and Data Warehouses (SiKDD 2008)*. Jozef Stefan Institute, 2008.
2. A.S. Arefin, C. Riveros, R. Berretta, P. Moscato. GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs. In *PLoS ONE 7(8)*: e44000. doi:10.1371/journal.pone.0044000, 2012.
3. A. Bienert. An implementation of a Binary Bayes Classifier in F#. http://andrewbienert.blogspot.nl/2011/05/implementation-of-binary-bayes.html.
4. P. Kontkanen, P. Myllymaki, T. Silander and H. Tirri. BAYDA: Software for Bayesian Classification and Feature Selection. In *Proc. of the 4th Int'l Conf. on Knowledge Discovery and Data Mining*, 1998.
5. A. M. Kuchling. Python 2.7.3 documentation - Functional Programming HOWTO. http://docs.python.org/2/howto/functional.html
6. C.D. Manning, P. Raghavan and H. Schutze. Introduction to information retrieval. Cambridge University Press, 2008.
7. D.Mertz. Charming Python: Functional programming in Python. http://www.ibm.com/developerworks/linux/library/l-prog/index.html

[5] The reference performance of different classifiers for Census Bureau data set is available with the data at UCI website.

8. I. Rish. An empirical study of the naive Bayes classifier. In *Proceedings of IJCAI-01 workshop on Empirical Methods in AI*, pages 41–46, Sicily, Italy, 2001.
9. M.M. Suarez-Alvarez, D. Pham, M.Y. Prostov and Y.I. Prostov. Statistical approach to normalization of feature vectors and clustering of mixed datasets. In *Proc. R. Soc. A*, 2011. doi: 10.1098/rspa.2011.0704.
10. Wikipedia - K-nearest neighbor algorithm. http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm.
11. Wikipedia - Statistical classification. http://en.wikipedia.org/wiki/Statistical_classification.