Machine Learning In Python

Ms. Prajakta Ambalkar, Mr. Dhiraj Rane
1Department of Computer Science, MCA-sem5, RTMNU, Nagpur, Maharashtra.
2Department of Computer Science, MCA, RTMNU, Nagpur, Maharashtra.

Abstract: Scikit-learn is an increasingly popular machine learning library. Written in Python, it is designed to be simple and efficient, accessible to non-experts, and reusable in various contexts. In this paper, we present and discuss the libraries that are numpy, pandas, and scipy also discuss the design choices for the application programming interface (API) of the project. In particular, we describe the simple and elegant interface shared by all learning and processing units in the library and then discuss its advantages in terms of composition and reusability. Scikit-learn is the package focuses on bringing machine learning to non-specialists using a general-purpose high-level language. Emphasis is put on ease of use, performance, documentation, and API consistency. It has minimal dependencies and is distributed under the simplified BSD license, encouraging its use in both academic and commercial settings.

Keywords: API, NumPy, Pandas, SciPy, Algorithms, Models, and Modules.

I. Introduction

Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of Computer Programs that can change when exposed to new data. In this article, we’ll see basics of Machine Learning, and implementation of a simple machine learning algorithm using python. Python community has developed many modules to help programmers implement machine learning. The Python programming language is establishing itself as one of the most popular languages for scientific computing. Thanks to its high-level interactive nature and its maturing ecosystem of scientific libraries, it is an appealing choice for algorithmic development and exploratory data analysis (Dubois, 2007; Milmann and Avaizis, 2011).

Yet, as a general-purpose language, it is increasingly used not only in academic settings but also in industry. Scikit-learn harnesses this rich environment to provide state-of-the-art implementations of many well-known machine learning algorithms, while maintaining an easy-to-use interface tightly integrated with the Python language. This answers the growing need for statistical data analysis by non-specialists in the software and web industries, as well as in fields outside of computer-science, such as biology or physics. Scikit-
Machine Learning In Python

learn differs from other machine learning toolboxes in Python for various reasons: i) it is distributed under the BSD license ii) it incorporates compiled code for efficiency, unlike MDP (Zito et al., 2008) and pybrain (Schaul et al., 2010), iii) it depends only on numpy and scipy to facilitate easy distribution, unlike pymvpa (Hanke et al., 2009) that has optional dependencies such as R and shogun, and iv) it focuses on imperative programming, unlike pybrain which uses a data-flow framework. While the package is mostly written in Python, it incorporates the C++ libraries LibSVM (Chang and Lin, 2001) and LibLinear (Fan et al., 2008) that provide reference implementations of SVMs and generalized linear models with compatible licenses. Binary packages are available on a rich set of platforms including Windows and any POSIX platforms. Furthermore, thanks to its liberal license, it has been widely distributed as part of major free software distributions such as Ubuntu, Debian, Mandriva, NetBSD and Macports and in commercial distributions such as the “Enthought Python Distribution”.

II. Project Vision

Code quality. Rather than providing as many features as possible, the project’s goal has been to provide solid implementations. Code quality is ensured with unit tests—as of release 0.8, test coverage is 81%—and the use of static analysis tools such as pyflakes and pep8. Finally, we strive to use consistent naming for the functions and parameters used throughout a strict adherence to the Python coding guidelines and numpy style documentation. SD licensing. Most of the Python ecosystem is licensed with non-copy left licenses. While such a policy is beneficial for adoption of these tools by commercial projects, it does impose some restrictions: we are unable to use some existing scientific code, such as the GSL. Bare-bone design and API. To lower the barrier of entry, we avoid framework code and keep the number of different objects to a minimum, relying on numpy arrays for data containers. Community-driven development. We base our development on collaborative tools such as git, github and public mailing lists. External contributions are welcome and encouraged. Documentation. Scikit-learn provides a 300 page user guide including narrative documentation, class references, a tutorial, installation instructions, as well as more than 60 examples, some featuring real-world applications. We try to minimize the use of machine-learning jargon, while maintaining precision with regards to the algorithms employed.

III. Underlying Technologies

Numpy: the base data structure used for data and model parameters. Input data is presented as numpy arrays, thus integrating seamlessly with other scientific Python libraries. Numpy’s view based memory model limits copies, even when binding with compiled code (Van der Walt et al., 2011). It also provides basic arithmetic operations.

Scipy: efficient algorithms for linear algebra, sparse matrix representation, special functions and basic statistical functions. Scipy has bindings for many Fortran-based standard numerical packages such as LAPACK. This is important for ease of installation and portability, as providing libraries around FORTRAN code can prove challenging on various platforms.

Cython: a language for combining C in Python. Cython makes it easy to reach the performance of compiled languages with Python-like syntax and high-level operations. It is also used to bind compiled libraries, eliminating the boilerplate code of Python/C extensions.

A. NumPy:

NumPy derives from an old library called Numeric, which was the first array object built for Python. It was quite successful and was used in a variety of applications before being phased out. NumPy also incorporates features introduced by a library called Numarray, which was written after Numeric but before NumPy. When NumPy was first written, it wasn’t actually called “NumPy”. For about 6 months at the end of 2005, NumPy was called SciPy Core (not to be confused with the full SciPy package which remains a separate package). However, it was decided in January 2006 to go with the historical name of NumPy for the new package.

Inclusion of a Numpy in Python’s standard library: In the opinion of many involved in the NumPy development, an N-dimensional array interface should be part of the Python standard libraries. Hence, a PEP was started to describe what exactly is meant by an array interface, and a webpage was set up with useful information. At the SciPy conference in 2006, Guido and Travis discussed which parts of NumPy should go into Python. They decided that the best course to pursue is to write a series of PEPs to get
1. the data-type object into Python
2. Extend the buffer interface with the array interface.
B. SciPy:

In the 1990s, Python was extended to include an array type for numerical computing called Numeric (This package was eventually replaced by Travis Oliphant who wrote NumPy in 2006 as a blending of Numeric and Numarray which had been started in 2001). As of 2000, there was a growing number of extension modules and increasing interest in creating a complete environment for scientific and technical computing. In 2001, Travis Oliphant, Eric Jones, and Pearu Peterson merged code they had written and called the resulting package SciPy. The newly created package provided a standard collection of common numerical operations on top of the Numeric array data structure. Shortly thereafter, Fernando Pérez released IPython, an enhanced interactive shell widely used in the technical computing community, and John Hunter released the first version of Matplotlib, the 2D plotting library for technical computing. Since then the SciPy environment has continued to grow with more packages and tools for technical computing. Several people used Numeric as a base for their scientific code and developed their own modules. Around 2001, Travis Oliphant, Eric Jones and Pearu Peterson merged their modules in one scientific super package: SciPy was born.

Data structures: The basic data structure used by SciPy is a multidimensional array provided by the NumPy module. NumPy provides some functions for linear algebra, Fourier transforms, and random number generation, but not with the generality of the equivalent functions in SciPy. NumPy can also be used as an efficient multidimensional container of data with arbitrary datatypes. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases. Older versions of SciPy used Numeric as an array type, which is now deprecated in favour of the newer NumPy array code.

C. Pandas:

Pandas is hands down one of the best libraries of python. It supports reading and writing excel spreadsheets, CVS's and a whole lot of manipulation. It is more like a mandatory library you need to know if you’re dealing with datasets from excel files and CSV files. I.e. for Machine learning and data science. Machine Learning with Python. Machine learning is a branch in computer science that studies the design of algorithms that can learn. Typical tasks are concept learning, function learning or “predictive modelling”, clustering and finding predictive patterns.

![Figure 2: Phases of Machine Learning](image)

IV. Core API

"In simple words, an API is a (hypothetical) contract between 2 Softwares saying if the user software provides input in a pre-defined format, the later with extend its functionality and provide the outcome to the user software." All objects within scikit-learn share a uniform common basic API consisting of three complementary interfaces: an estimator interface for building and fitting models, a predictor interface for making predictions and a transformer interface for converting data. In this section, we describe these three interfaces, after reviewing our general principles and data representation choices.

A. General principles

As much as possible, our design choices have been guided so as to avoid the proliferation of framework code. We try to adopt simple conventions and to limit to a minimum the number of methods an object must implement. The APIs is designed to adhere to the following broad principles:
Consistency. All objects (basic or composite) share a consistent interface composed of a limited set of methods. This interface is documented in a consistent manner for all objects. Inspection, Constructor parameters and parameter values determined by learning algorithms are stored and exposed as public attributes. Non-proliferation of classes. Learning algorithms are the only objects to be represented using custom classes. Datasets are represented as NumPy arrays or SciPy sparse matrices. Hyper-parameter names and values are represented as standard Python strings or numbers whenever possible. This keeps scikit-learn easy to use and easy to combine with other libraries. Composition. Many machine learning tasks are expressible as sequences or combinations of transformations to data. Some learning algorithms are also naturally viewed as meta-algorithms parameterized on other algorithms. Whenever feasible, such algorithms are implemented and composed from existing building blocks. Sensible defaults. Whenever an operation requires a user-defined parameter, an appropriate default value is defined by the library. The default values should cause the operation to be performed in a sensible way (giving a baselinesolution for the task at hand).

B. Data representation:

In most machine learning tasks, data is modelled as a set of variables. For example, in a supervised learning task, the goal is to find a mapping from input variables $X_1, \ldots, X_p$, called features, to some output variables $Y$. A sample is then defined as a pair of values $([x_1, \ldots, x_p]^T, y)$ of these variables. A widely used representation of a dataset, a collection of such samples, is a pair of matrices with numerical values: one for the input values and one for the output values. Each row of these matrices corresponds to one sample of the dataset and each column to one variable of the problem. In scikit-learn, we chose a representation of data that is as close as possible to the matrix representation: datasets are encoded as NumPy multidimensional arrays for dense data and as SciPy sparse matrices for sparse data. While these may seem rather unsophisticated data representations when compared to more object-oriented constructs, such as the ones used by Weka (Hall et al., 2009), they bring the prime advantage of allowing us to rely on efficient NumPy and SciPy vectorised operations while keeping the code short and readable. This design choice has also been motivated by the fact that, given their pervasiveness in many other scientific Python packages, many scientific users of Python are already familiar with NumPy dense arrays and SciPy sparse matrices. From a practical point of view, these formats also provide a collection of data loading and conversion tools which make them very easy to use in many contexts. Moreover, for tasks where the inputs are text files or semi-structured objects, we provide vectorizer objects that efficiently convert such data to the NumPy or SciPy formats. For efficiency reasons, the public interface is oriented towards processing batches of samples rather than single samples per API call. While classification and regression algorithms can indeed make predictions for single samples, scikit-learn objects are not optimized for this use case. (The few online learning algorithms implemented are intended to take mini-batches.) Batch processing makes optimal use of NumPy and SciPy by preventing the overhead inherent to Python function calls or due to per-element dynamic type checking. Although this might seem to be an artefact of the Python language, and therefore an implementation detail that leaks into the API, we argue that APIs should be designed so as to tie a library to a suboptimal implementation strategy. As such, batch processing enables fast implementations in lower-level languages (where memory hierarchy effects and the possibility of internal parallelization come into play).

```python
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(penalty="l1")
clf.fit(X_train, y_train)
```

In this snippet, a LogisticRegression estimator is first initialized by setting the penalty hyper-parameter to "l1" for $\ell$ regularization. Other hyper-parameters (such as C, the strength of the regularization) are not explicitly given and thus set to the default values. Upon calling fit, a model is learned from the training arrays X_train and y_train, and stored within the object for later use. Since all estimators share the same interface, using a different learning algorithm is as simple as replacing the constructor (the class name); to build a random forest on the same data, one would simply replace Logistic Regression (penalty="l1") in the snippet above by RandomForestClassifier (). In scikit-learn, classical learning algorithms are not the only objects to be implemented as estimators. For example, pre-processing routines (e.g., scaling offeatures) or feature extraction techniques (e.g., vectorization of text documents) also implement the estimator interface. Even stateless processing steps, that donot require the fit method to perform useful work, implement the estimator interface. As we will illustrate in the next sections, this design pattern is indeed of prime importance for consistency, composition and model selection reasons.
V. Predictors

The predictor interface extends the notion of an estimator by adding a predict method that takes an array X test and produces predictions for X test, based on the learned parameters of the estimator (we call the input to predict “X test” in order to emphasize that predict generalizes to new data). In the case of unsupervised learning estimators, this method typically returns the predicted label or values computed by the model. Continuing with the previous example, predicted labels for X test can be obtained using the following snippet:

```python
ypred = clf.predict(Xtest)
```

VI. Estimators

The estimator interface is at the core of the library. It defines instantiation mechanisms of objects and exposes a fit method for learning a model from training data. All supervised and unsupervised learning algorithms (e.g., for classification, regression or clustering) are offered as objects implementing this interface. Machine learning tasks like feature extraction, feature selection ordimensionality reduction are also provided as estimators. Estimator initialization and actual learning are strictly separated, in a way that is similar to partial function application: an estimator is initialized from a set of named constant hyper-parameter values (e.g., the C constant in SVMs) and can be considered as a function that maps these values to actual learning algorithms. The constructor of an estimator does not see any actual data, nor does it perform any actual learning. All it does is attach the given parametersto the object. For the sake of convenient model inspection, hyper-parameters are set as public attributes, which is especially important in model selection settings. For ease of use, default hyper-parameter values are also provided for all built-in estimators. These default values are set to be relevant in many common situations in order to make estimators as effective as possible out-of-box for non-experts. Actual learning is performed by the fit method. This method is called with training data (e.g., supplied as two arrays X train and y train in supervised learning estimators). Its task is to run a learning algorithm and to determinemodel-specific parameters from the training data and set these as attributes on the estimator object. As a convention, the parameters learned by an estimator are exposed as public attributes with names suffixed with a trailing underscore (e.g., coef for the learned coefficients of a linear model), again to facilitate model inspection. In the partial application view, fit is a function from datato a model of that data. It always returns the estimator object it was called on, which now serves as a model of its input and can be used to perform predictions or transformations of input data. From the start, the choice to let a single object serve dual purpose as estimator and model has mostly been driven by usability and technical considerations. From the user point of view, having two coupled instances (i.e., an estimator object, used as a factory, and a model object, produced by the estimator) indeed reduces the ease of use and is also more likely to unnecessarily confuse newcomers. From the developer point of view, decoupling estimators from models also creates parallel class hierarchies and increases the overall maintenance complexity of the project. For these practical reasons, we believe that decoupling estimators from models is not worth the effort. A good reason for decoupling, however, would be that it makes it possible to ship a model in a new environment without having to deal with potentially complex software dependencies. Such a feature could however still be implemented in scikit-learn by making estimators able to export a fitted model, using the information from its public attributes, to an agnostic model description such as PMML (Guazzelli et al., 2009). To illustrate the initialize-fit sequence, let us consider a supervised learning task using logistic regression. Given the API defined above, solving this problem is as simple as the following example.

VII. Model Selection

As introduced in Section 2, hyper-parameters set in the constructor of an estimator determine the behaviour of the learning algorithm and hence the performance of the resulting model on unseen data. The problem of model selection, therefore, to find, within some hyper-parameter space, the best combination of hyper-parameters, with respect to some user-specified criterion. For example, a decision tree with too small a value for the maximal tree depth parameter will tend to underfit, while too large a value will make it overfit. In scikit-learn, model selection is supported in two distinct meta-estimators, GridSearchCV and RandomizedSearchCV. They take as input an estimator (basically, a composite), whose hyper-parameters must be optimized, and a set of hyper-parameters settings to search through. This set is represented as a mapping of parameter names to a set of discrete choices in the case of grid search, which exhaustively enumerates the “grid” (Cartesian product) of complete parameter combinations. Randomized search is a smarter algorithm that avoids the combinatorial explosion in grid search by sampling a fixed number of times from its parameter distributions (see Bergstra and Bengio, 2012).
VIII. Implementation

Our implementation guidelines emphasize writing efficient but readable code. In particular, we focus on making the codebase easily maintainable and understandable in order to favor external contributions. Whenever practicable, algorithms implemented in scikit-learn are written in Python, using NumPy vector operations for numerical work. This allows for the code to remain concise, readable and efficient. For critical algorithms that cannot be easily and efficiently expressed as NumPy operations, we rely on Cython (Behnel et al., 2011) to achieve competitive performance and scalability. Cython is a compiled programming language that extends Python with static typing. It produces efficient C extension modules that are directly importable from the Python run-time system. Examples of algorithms written in Cython include stochastic gradient descent for linear models, some graph-based clustering algorithms and decision trees. To facilitate the installation and thus adoption of scikit-learn, the set of external dependencies is kept to a bare minimum: only Python, NumPy and SciPy are required for a functioning installation. Binary distributions of these are available for the major platforms. Visualization functionality depends on Matplotlib (Hunter, 2007) and/or Graphviz (Gansner and North, 2000), but neither is required to perform machine learning or prediction. When feasible, external libraries are integrated into the codebase. In particular, scikit-learn includes modified versions of LIBSVM and LIBLINEAR (Chang and Lin, 2011; Fan et al., 2008), both written in C++ and wrapped using Cython modules.

IX. Conclusion

The foremost target of ML researchers is to design more efficient (in terms of both time and space) and practical general purpose learning methods that can perform better over a widespread domain. In the context of ML, the efficiency with which a method utilizes data resources that is also an important performance paradigm along with time and space complexity. Higher accuracy of prediction and humanly interpretable prediction rules are also of high importance. Being completely data-driven and having the ability to examine a large amount of data in smaller intervals of time, ML algorithms has an edge over manual or direct programming. Also they are often more accurate and not prone to human bias. We have discussed the scikit-learn API and the way it maps machine learning concepts and tasks onto objects and operations in the Python programming language. We have shown how a consistent API across the package makes scikit-learn very usable in practice: experimenting with different learning algorithms is as simple as substituting a new class definition. Through composition interfaces such as Pipelines, Feature Unions, and Metaestimators, these simple building blocks lead to an API which is powerful, and can accomplish a wide variety of learning tasks within a small amount of easy-to-read code. Through duck-typing, the consistent API leads to a library that is easily extensible, and allows user defined estimators to be incorporated into the scikit-learn workflow without any explicit object inheritance. While part of the scikit-learn API is necessarily Python-specific, core concepts may be applicable to machine learning applications and toolkits written in other (dynamic) programming languages. The power, and extensibility of the scikit-learn API is evidenced by the large and growing user base, its use to solve real problems across a wide array of fields, as well as the appearance of third-party packages that follow the scikit-learn conventions. ML provide software the flexibility and adaptability when necessary. In spite of some application (e.g., to write matrix multiplication programs) where ML may fail to be beneficial, with increase of data resources and increasing demand in personalised customisable software, ML will thrive in near future. Besides software development, ML will probably but help reform the general outlook of Computer Science. By changing the defining question from “how to program a computer” to “how to empowered to program itself,” ML priorities the development of devices that are self-monitoring, self-diagnosing and self-repairing, and the utilises of the data flow available within the program rather than just processing it. Likewise, it will help reform Statistical rules, by providing more computational stance. Obviously, both Statistics and Computer Science will also embellish ML as they develop and contribute more advanced theories to modify the way of learning.

Reference

[4] This is an archival dump of old wiki content --- see scipy.org for current material available: https://scipy.github.io/old-wiki/pages/History_of_SciPy
