G. Varoquaux
Parietal, INRIA, CEA Institute, France

L. Buitinck
University of Amsterdam

G. Louppe
University of Liège, Belgium

O. Grisel
Parietal, INRIA, CEA Institute, France

F. Pedregosa
Parietal, INRIA, CEA Institute, France

A. Mueller
Amazon Development Centre, Berlin

Editor: Matthai Philipose

SCIKIT-LEARN: Machine Learning Without Learning the Machinery

Machine learning is a pervasive development at the intersection of statistics and computer science. While it can benefit many data-related applications, the technical nature of the research literature and the corresponding algorithms slows down its adoption. Scikit-learn is an open-source software project that aims at making machine learning accessible to all, whether it be in academia or in industry. It benefits from the general-purpose Python language, which is both broadly adopted in the scientific world, and supported by a thriving ecosystem of contributors. Here we give a quick introduction to scikit-learn as well as to machine-learning basics.

A SOFTWARE PROJECT ACROSS COMMUNITIES

Project vision Scikit-learn was born from the observation that most standard machine-learning algorithms were out of reach of the users that could most benefit from them: researchers – biologists, climate scientists, experimental physicists – or developers of web services or domain-specific applications. In the scientific world, implementations of these algorithms mostly consisted of scattered pieces of code on researchers' web pages. Unifying efforts could be found in statistics-specific environments and libraries such as the R language [16], the Weka Java toolkit [7], or the Shogun C++ library [18]. Of these, R is a custom *language,* while Weka is a Java library with a custom GUI, and Shogun is a fairly technical C++ code base. In scikit-learn, we chose Python to reach many users, technical or not, tying machine learning into a general-purpose language good for simple interactive programming [15], sophisticated scripting, as well as full-blown applications.

Scikit-learn aims at bridging the gap between machine-learning research and applications by providing a library, and not an environment, in a general-purpose programming language, relying on domain-agnostic data structures [14]. Emphasis is put on quality and ease of use, which implies focus on installation issues, documentation, and API design [4]. For the project to be usable in real-world settings, computational performance is also a priority.

The Python data ecosystem Machine learning is only a small part of a data-analysis pipeline, and scikit-learn dovetails nicely into the rich Python ecosystem. These include powerful scientific and numeric tools [12, 6], but also extensive support for text-processing, web services, cryptography and visualization. For its numerical needs, scikit-learn leverages NumPy arrays [20] (data structures for efficient numerical computation), SciPy (a collection of classic numerical algorithms), matplotlib [9] (a package for scientific plotting) and Cython

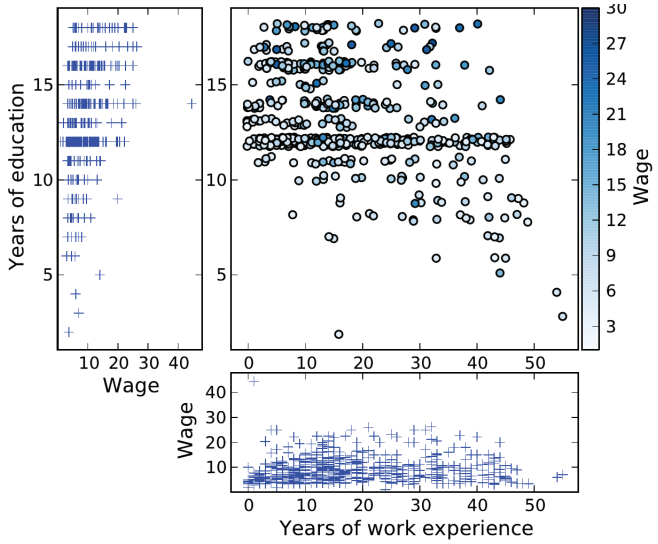

FIGURE 1. Wage as a function of years of work experience and education; data from [2].

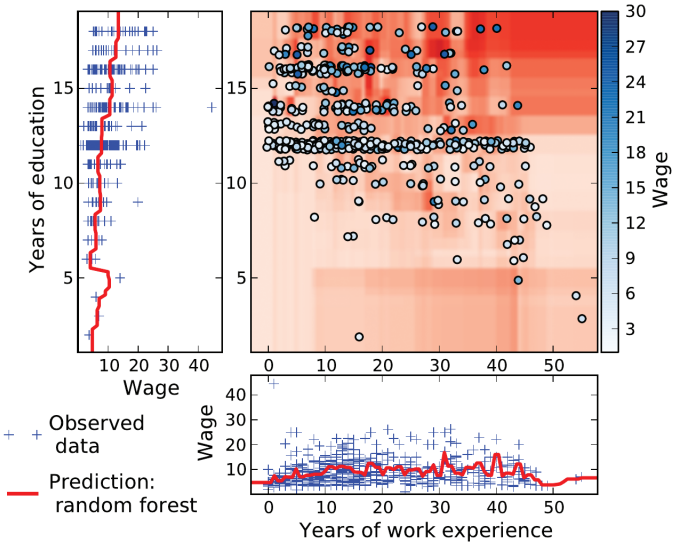

FIGURE 2. Wage prediction from years of work experience and education, using random forests.

(a programming language to generate compiled code in a Python-like syntax [1]). A plethora of Python packages can help users with input or preprocessing of data, notably Pandas for columnar data [11], scikit-image for images, and NLTK for text. Finally, the IPython environment [15] is invaluable for interactive work.

Some history. Scikit-learn started circa 2007 as code from David Cournapeau and Matthieu Brucher's PhD work, but dwindled until 2010, when the Parietal team at INRIA adopted the project, hiring a full-time engineer. The team defined basic APIs and the efficient binding of LibSVM [5] they developed gave the project a compelling advantage. In December 2011, the first international sprint was organized with generous funding from Google. Today, the project has grown vastly beyond INRIA into a worldwide open source effort with more than 200 contributors.

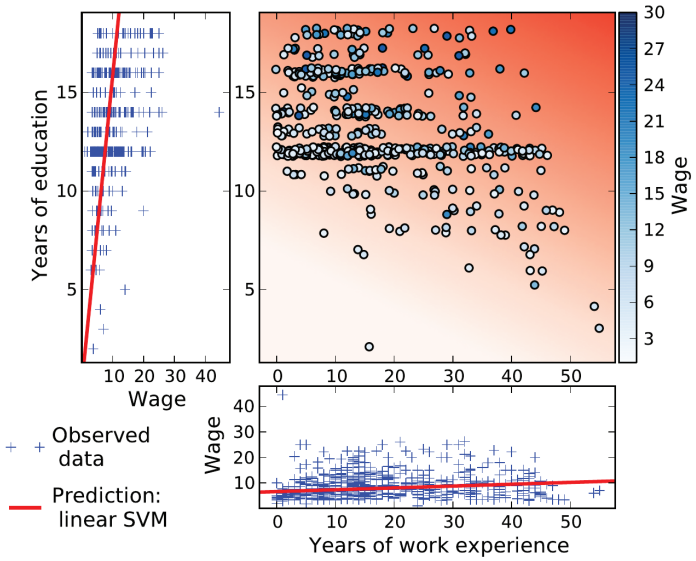

FIGURE 3. Wage prediction from years of work experience and education, using a linear SVM.

A BRIEF INTRODUCTION TO MACHINE LEARNING

Machine learning is about extracting rules from data, most often with the goal of making decisions on new data [8]. As a simple example, we show in Fig. 1 data about wage in the US [2]: 11 *features*, including wage, number of years of education and of work experience, describe 534 individuals, called *samples* in machine-learning jargon.

This example showcases many of the typical difficulties in machine learning. Samples are irregularly distributed, as most individuals completed studies after high school. As a result, there are gaping holes in the 2D plot of education versus work experience, in which statistical claims require extrapolation. The data is very noisy: for a given pair of education and work experience values, wage varies widely. This variability is probably explained by missing factors, such as sector of activity, but adding them to the analysis creates a

more complex picture, in 3D or more, rather than 2D, with even more gaps.

In Fig. 2, we use a popular machine-learning algorithm, random forests, to predic t the wage from the years of education, or the years of work experience, either as separate features, or combined. The patchy square appearance of the prediction is due to the workings of the algorithm, and illustrates the corresponding extrapolation mechanism. The predictions fit the data well, perhaps too well: on the prediction solely from the years of education (left of the figure), it is hard to believe in the bump at five years. This bump is probably a case of *overfitting*: the algorithm is learning its prediction from noise in the data. As a result, the prediction error on new data will be significantly different (usually, higher and therefore worse) than that measured on the training data.

In Fig. 3, we use another popular machine-learning algorithm, linear Support Vector Machines (SVMs). Unlike random forests, it learns decisions with linear functions. It is said that it has a lower *model complexity*, because the number of parameters to learn from the data is much smaller. The risk here is to *underfit*: the algorithm may not fully use the richness of the data, which may not follow a linear law. The art of machine learning consists in choosing the right family of algorithms/models to describe the data well and find the sweet spot between under- and overfit. As the number of features describing the data grows, the number of parameters to learn grows, and thus the risk of overfitting increases. This core difficulty is known as the *curse of dimensionality*.

LEARNING WITH SCIKIT-LEARN

Setting up models. Learning algorithms in scikit-learn are embodied in *estimators*, objects instantiated with parameters that control learning. *Training data* is passed to the fit method, which accepts an $(n \times p)$ data matrix **X**, represented as a NumPy array or SciPy sparse matrix, where n is the number of samples, and p the number of features. When there is a quantity or class label to predict, a second argument **y** (usually a 1D $n \times 1$ array) contains the desired outcomes for the rows in **X**. For a regression task such as wage prediction, these are real-valued, while in classification they are integers or strings that serve as labels for the class to which each sample belongs. Because data seldom arrives in this format in the real world, scikit-learn comes with flexible feature extraction code to make data suitable for consumption by estimators.

Supervised models: learning to predict. The goal of supervised models is the prediction of some value of interest. The corresponding estimators have a predict method, that takes a data matrix **X** and returns a predicted **y**.

The performance of a model measures its ability to correctly predict new data. Because of overfitting, the training data used for learning should in general not be used to also evaluate model performance, as it may result in overly optimistic estimates of the true prediction error of the model. Instead, the correct way to obtain an unbiased estimate is to leave out *test* data, untouched during training and used for model evaluation only. Alternatively, a *cross-validation* scheme may also be used, where the data is repeatedly split into *train* and *test* subsets.

Scikit-learn provides integrated support for cross-validation, as specific iterators defining the train and test subsets. Several functions and objects accept these iterators as arguments to perform cross-validation internally. For instance the cross_val_score function measures the prediction score of an estimator. Cross-validation can also be used to tune the meta-parameters of an estimator, such as the sparsity of a sparse model. For this, specific *meta-estimators*, such as the GridSearchCV, take another estimator at construction time, and use cross-validation internally to set its parameters. When used together with cross_val_score, they perform *nested* cross-validation, *i.e.*, the meta-parameters are set independently of the test data.

Unsupervised models: learning to transform. Unsupervised learning covers all learning applications in which there is no clearly identified variable to predict. For instance, in a *clustering* application, the task is to group together observations that are similar. *Dimension reduction* tries to find simplified representations that capture well the properties of the data. *Novelty detection* finds in a new dataset observations that differ from the data in the train set.

As the uses of unsupervised learning are very diverse, it is not possible to have an API as uniform as for supervised learning. While some unsupervised estimators can be used to predict a characteristic of new data, such as in novelty detection, many are useful to transform data, as in dimension reduction. Scikit-learn estimators can have a transform method, used for this purpose. A Pipeline estimator can then chain estimators to form a new estimator that applies transformations before calling the fit or predict method of the last object.

IN PRACTICE: PUTTING SCIKIT-LEARN TO WORK

A simple text-mining example. Not every dataset comes as a set of numbers. For instance in natural language processing (NLP), observations are strings, which needs to be transformed into the data matrix expected by estimators. Often, features used to build decisions indicate the presence of certain words, or their frequencies. In spam filtering, it can be very relevant to extract from a document that it contains the term "casino." In other NLP problems, such as finding proper names, events such as "next token is a Roman numeral" may be relevant, as in "J.P. Kennedy III." Reusing the transform API, scikit-learn provides *feature extraction* objects, which turn such unstructured data into a data matrix. In these situations, the **X** matrix is a matrix of counting statistics. Since each word is typically present in only a few documents, the data matrix will be mostly zeros. SciPy offers a *sparse matrix* object that stores the zeros in such matrices implicitly, and many scikit-learn estimators can use this object to improve scalability.

In Fig. 4 we give an example of fully functioning code to do sentiment polarity classification for movie reviews: given a review, it outputs whether that review is positive about the movie it concerns. The code demonstrates the use of pipelines, and also shows how machine learning ties into other Python modules: downloading, unpacking and learning from a dataset are all done in one programming language. This script first fetches a hand-labeled movie review dataset [13] and unpacks it to disk using standard Python libraries. It then loads these using scikit-learn data loading

functions, makes a pipeline of a tf–idf feature extractor [17] (that turns strings into a sparse matrix based on word frequencies) and logistic regression, and finally trains the model. Meta-parameters for the feature extractor and the classifier are set manually. We can test our classifier on two new movie reviews.[1]

```
>>> clf.predict(["Worst movie I ever saw",
... "Godzilla, eat your heart out!"])
array([0, 1])
```

Here, we asked the estimator to predict whether the reviews were positive. The first review is predicted as being negative (0), while the second is positive (1). We have anecdotal evidence that the classifier works. To get a measure for the classifier's accuracy we retrain it in a cross-validation scheme. This scheme yields the fraction of correct answers on three different folds:

```
>>> from sklearn import cross_validation
>>> cross_validation.cross_val_score(clf,
    data.data, data.target)
array([0.9026946, 0.8723724, 0.8633634])
```

We see that the expected accuracy is about 88%. Experimenting with different classification models is easy: e.g., replacing LogisticRegression with svm.LinearSVC gives a linear support vector machine.

What about big data? *Big data* is a major trend in data analytics: applying machine learning to large datasets can yield useful new insight. While scikit-learn is most efficient in the "medium data" range, using shared memory multiprocessors rather than clusters, it has facilities for dealing with datasets that don't fit in any single machine.

Thinking in terms of the $(n \times p)$ data matrix, we can distinguish between the case of large numbers of samples n, and large numbers of features p. For a large-p case, scikit-learn's options include random projections, which reduce data dimensionality while preserving most of the geometric structure. For the large-n case, some estimators expose a partial_fit method that does *online learning*. Instead of loading all of the data in a single huge matrix, the user can feed the estimator chunks of the data, and the algorithm finds a good approximation to its objective after a few passes over the dataset.

With large collections of text, one faces both challenges: the dimensionality p tends to increase with sample size n, as the number of different words used is greater in large collections [10, 88–89]. Beyond the mere size of the data matrix created by a standard vectorization approach, outlined earlier, another challenge is that the vectorization must be done in two passes over the documents, as the size of the output vectors is not known up front. *Feature hashing* can be used to tackle both of these challenges and produce a streaming vectorizer, the HashingVectorizer in scikit-learn. Words are run through a (stateless) hash function, rather than a lookup table, to map words to indices. Collisions may occur, but the hashing algorithm [21] is specially designed to mitigate their effect. The dimensionality of the output vectors can be set by the user; the bigger, the less likely collisions will occur. A full example doing online learning on the 20 Newsgroups corpus[2] is given in Fig. 5 (a downloader for this corpus comes with scikit-learn).

```
import urllib, tarfile # From the Python standard library
from sklearn import linear_model, pipeline, datasets, feature_extraction

# Download, unpack and load the dataset.
urllib.urlretrieve('http://bit.ly/1juuXIr', 'tmp.tgz')
tarfile.open('tmp.tgz').extractall(path='.')
data = datasets.load_files('txt_sentoken')

clf = pipeline.Pipeline([('step1', feature_extraction.text.TfidfVectorizer(min_df=2,
                                                sublinear_tf=True, ngram_range=(1, 2))),
                         ('step2', linear_model.LogisticRegression(C=5000))])
clf.fit(data.data, data.target)
```

FIGURE 4. Simple text-processing code to download movie reviews and learn positive versus negative ratings.

NURTURING AN OPEN SOURCE PROJECT

The greatest asset of the scikit-learn project is its breadth of contributors, coming from different backgrounds, working in different institutions, with a variety of applications: more than 200 contributors in total, 15 active core contributors in the past year. Indeed, not only is there strength in numbers, but the convergence of points of view make for better code and better design. From a developer's perspective, the project is managed with the goal of enabling newcomers to contribute, while keeping quality high.

Our development workflow relies on intensive code review, facilitated by the online version control environment GitHub (http://github.com). The code review process is a great asset to raising the quality of the code, from all points of view: algorithmic, numerical robustness, homogeneity of APIs, ease of use and documentation. Another central tool for maintaining quality in the project is the use of unit testing (some 88% of the lines of code are covered). All would-be code contributions are automatically tested using Travis (http://travis-ci.org).

The open-source community-driven model has produced a useful, high-quality artifact in scikit-learn. However, we find that this model does not fit in the standard professional incentive structure, whether it be in academia or in the industry. Indeed, getting due credit for grassroots free software that "just works" is difficult. None of the grant proposals submitted to fund development by core contributors has so far been accepted, and there is no straightforward revenue model. Even more challenging is rewarding properly the long tail of small contributors.

Defining scope and priorities is a key challenge. As scikit-learn's purpose is to provide tried-and-true, mostly turn-key, predictive tools, we have to walk the fine line between supporting powerful

[1] The code can be tried interactively by simply copy-pasting it into an interactive Python interpreter or IPython [15].
[2] http://qwone.com/~jason/20Newsgroups/

techniques and utilizing our limited project cycles on proven technologies. Scikit-learn has become a reference machine learning toolkit. As such, it should offer a wide variety of learning algorithms. However, each new feature comes with a maintenance cost. Uncontrolled growth would lead the project to pick up weight until it came to a crawl.

On the one hand, scikit-learn has been developed with high standards of quality. We strive to identify and integrate all major methods useful for prediction tasks. We favour time-tested algorithms and put great care in the choice of default parameters. We put effort into making the documentation didactic and pragmatic. Our ambition is that the scikit code base can provide an executable alternative to textbooks, with all the figures generated by code examples.

On the other hand, machine learning is a very active field of research, with promising trends such as deep learning, harnessing GPUs or distributed computing for extra computational power, or extending to new platforms, such as Android or iOS-based embedded devices. However, deep neural networks are an emerging tool with a huge potential, but are still in flux and hard to use without significant expertise. Similarly, GPGPU technology is promising, but not ready for non-expert use. The HPC community and industry have largely achieved source-level compatibility with OpenCL [19], but performance portability and cross-platform driver support remain challenging. Finally, support for mobile platforms such as Android is determined by those platforms' support for Python and the standard C/C++ build chain. Scikit-learn is designed to run on any platform that supports the scientific Python stack.

SUMMARY

The vision of the scikit-learn project is to bring advances in machine learning to users that that may lack the full expertise. As a consequence, we focus on seasoned approaches, rather than implementing ongoing research. We believe that code is an intrinsic part of the scholarly output of scientific research in computational science [3], and scikit-learn strives to be akin to a reference textbook in this research software landscape that also scales to being useable at in production settings. ■

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import HashingVectorizer

vectorizer = HashingVectorizer(decode_error='ignore', n_features=2 ** 16)

dataset = fetch_20newsgroups(shuffle=True, random_state=1, remove=('headers', 'footers'))
X_test = vectorizer.fit_transform(dataset.data[:1000])
y_test = dataset.target[:1000]

from sklearn.linear_model import SGDClassifier
classifier = SGDClassifier() # SVM classifier trained online with stochastic gradient descent

# Artificially increase the size of the training set ten-fold
train_data = 10 * dataset.data[1000:]
train_target = 10 * list(dataset.target[1000:])

for i in range(0, len(train_data), 1000): # Iterate over "mini-batches" of 1000 samples each
    y_train = train_target[i:i + 1000]
    X_train = vectorizer.fit_transform(train_data[i:i + 1000])
    # Update the classifier with documents in the current mini-batch
    classifier.partial_fit(X_train, y_train, classes=range(len(dataset.target_names)))
    print(classifier.score(X_test, y_test))
```

FIGURE 5. On-line learning: the main loop iterates over batches of 1000 documents, vectorizing them on the fly and updating the classier. Running the example shows the error on the test set progressively decreasing as data is accumulated.

REFERENCES

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: the best of both worlds. *CiSE*, 13:31, 2011.

[2] E. R. Berndt. *The practice of econometrics: classic and contemporary.* Addison-Wesley, 1991.

[3] J. B. Buckheit and D. L. Donoho. *Wavelab and reproducible research.* Springer, 1995.

[4] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mu¨ller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. Vanderplas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In ECML PKDD *Workshop on Languages for Machine Learning*, 2013.

[5] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines. *ACM Trans. on Intelligent Systems and Technology*, 2:27, 2011.

[6] V. Haenel, E. Gouillart, and G. Varoquaux. Python scientific lecture notes, 2013.

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11:10, 2009.

[8] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Springer, 2009.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *CiSE*, page 90, 2007.

[10] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* CUP, 2009.

[11] W. McKinney. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.* O'Reilly, 2012.

[12] T. E. Oliphant. Python for scientific computing. *CiSE*, 9:10, 2007.

[13] B. Pang and L. Lee. A sentimental education: Sentiment analysis using subjectivity. In *Proc. ACL*, 2004.

[14] 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. *JMLR*, 12:2825, 2011.

[15] F. Perez and B. E. Granger. IPython: a system for interactive scientific computing. *CiSE*, 9(3):21, 2007.

[16] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, 2013. ISBN 3-900051-07-0.

[17] J. D. Rennie, L. Shih, J. Teevan, and David R. Karger. Tackling the poor assumptions of naive Bayes text classifiers. In *ICML*, page 616. Washington DC, 2003.

[18] S. Sonnenburg, G. Ra¨tsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. de Bona, A. Binder, C. Gehl, and V. Franc. The SHOGUN Machine Learning Toolbox. *Journal of Machine Learning Research*, 11:1799, 2010.

[19] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *CiSE*, 12(3), 2010.

[20] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *CiSE*, 13:22, 2011.

[21] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proc.* ICML, 2009.