Software: Python and Data Analytics

Seminar Data Mining

Adem Yahmadi
Department of Informatics
Technische Universität München
Email: adem.yahmadi@tum.de

***Abstract*— Python has become one of the most widely used programming languages for data mining and data analytics because of its readability, rapid development capabilities, and extensive ecosystem of scientific libraries. This paper discusses how Python supports data mining by examining both the language and the libraries commonly used in analytical workflows. First, it explains Python's main characteristics, including its programming paradigms, syntax, and performance. It then presents key libraries used throughout the data mining process, including NumPy for numerical computing, pandas for data preprocessing, SciPy for scientific and statistical analysis, scikit-learn for machine learning, and Plotly and Dash for visualization and deployment. The paper also shows how these libraries work together in an iterative data mining workflow that includes data preparation, modeling, evaluation, and result communication. Although Python has limitations due to interpreted execution and runtime overhead, many of these are reduced by optimized libraries implemented in lower-level languages. Overall, Python provides a practical balance of productivity, ecosystem support, and computational performance, making it a widely used platform for modern data mining and analytics.**

I. Introduction

An enormous amount of data is being generated by research and modern digital systems [1]. To extract useful knowledge from these data, for example, in artificial intelligence or machine learning, we need to analyze them [2, 3]. Data mining is the process of finding hidden patterns in data to make better predictions or support better decisions [4]. It therefore involves not only algorithms for data analysis but also software for data processing, modeling, and visualization [4–6].

The choice of programming language affects how easily these workflows can be developed, understood, and maintained [6, 7]. While compiled languages generally provide higher execution performance, higher-level languages often enable faster development and are better suited to experimentation [7, 8]. Many data analysts use Python for its clear syntax, fast prototyping capabilities, and support for external libraries [8, 9]. While tools such as MATLAB are widely used for numerical computing and specialized engineering tasks, Python has become one of the dominant programming languages for data mining, machine learning, and data science projects. It is open-source and has a broad ecosystem that supports everything from data analysis to final software deployment [8, 9].

One of Python's principal strengths is its extensive ecosystem of libraries across many application domains [9]. NumPy provides numerical array operations, pandas supports tabular data manipulation, and SciPy offers scientific and statistical methods [10–12]. Machine learning is supported by scikit-learn through a range of algorithms [13]. Visualization and interactive analysis are enabled by multiple libraries supporting both exploration and visual analysis [5]. Together, these libraries support the main stages of data analysis from raw data to interpretable results [9].

However, this flexibility comes with a trade-off. Python, as a dynamically typed and interpreted language, cannot match most compiled languages in execution speed [7, 14]. In practice, this limitation is often minimized by optimized libraries that move computationally intensive tasks to lower-level implementations [8, 15].

This paper describes Python's role in data mining and analytics. First, it introduces fundamental language characteristics, focusing on programming paradigms, syntax, and performance. It then presents the key libraries for numerical computation, data preprocessing, scientific analysis, machine learning, visualization, and deployment. Finally, these components are connected within an iterative workflow, showing how Python supports the transition from raw data to interpretable results. Deep learning frameworks such as PyTorch and TensorFlow, and interactive environments such as Jupyter, are mentioned only briefly where relevant, since they fall outside the classical data mining scope.

II. Python Language Characteristics

Python is widely used in data mining and data analytics [8, 9]. This is partly due to the language itself and due to the large scientific computing ecosystem supporting it [8, 9]. It provides a readable syntax that supports exploratory work [8, 16]. At the same time, Python's execution model can be slower than that of compiled languages such as C++ and Java [7, 14]. This section discusses Python's programming paradigms, syntax, and performance characteristics.

A. Programming Paradigms and Syntax

Python is a multi-paradigm language supporting procedural, object-oriented, and functional programming [16, 17]. In data analytics, these styles are especially relevant because workflows often combine scripting, reusable components, and compact transformations [8, 17].

- **Procedural programming:** Often used for simple scripts, data loading, and preprocessing.

- **Object-oriented programming:** Often used to build reusable components, such as model classes, data processing objects, and pipeline structures.
- **Functional programming:** Often used for data transformations with lambda functions, higher-order functions, and list comprehensions.

A typical Python program is a mix of different programming styles [16, 17]. Studies of open-source Python programs show that object-oriented and procedural structures are most common, while functional programming is less common and used for specific purposes [17]. This mixed approach is well-suited to data mining, which combines data loading scripts, reusable preprocessing components, and compact transformation functions [8, 17].

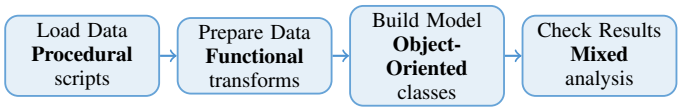


Fig. 1: Illustrative use of programming styles in a Python-based data mining workflow.

Figure 1 illustrates how different programming styles can appear in a Python-based data mining workflow. Real projects usually combine several styles [16, 17], but the figure highlights an important advantage of Python: it supports multiple data mining stages within a single workflow.

Python's syntax is designed for readability [16]. Instead of braces to define code blocks as in C++ or Java, Python uses indentation. This leads to a clean, readable structure across different projects. Because its syntax is visually structured, Python code is often easier to read and modify. This is especially important in data mining, where preprocessing and modeling code is frequently inspected and modified [6].

Python also uses dynamic typing, which means variables do not require explicit type declarations and can refer to different types of objects during execution [16]. This enables fast coding and easy experimentation [8]. However, because many checks occur at runtime, it can lead to performance overhead in computation-heavy tasks [14].

B. Performance

Python's performance depends strongly on how programs are executed. CPython, the standard and most widely used implementation of Python, is written in C and Python [16]. This executes Python programs by compiling source code into bytecode, which is then interpreted by the Python virtual machine (PVM) [16].

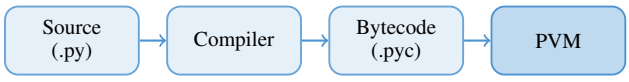


Fig. 2: CPython execution pipeline from source code to bytecode and PVM execution.

Figure 2 illustrates why Python's raw execution speed is usually lower than that of compiled languages [14, 18]. However, for most data mining applications, rapid development, easy testing, and code modification can be more important than maximum execution speed [7, 8].

Another key concept in standard CPython is the Global Interpreter Lock (GIL). In simple terms, the GIL is a mutex that allows only one thread to hold control of the Python interpreter [19]. This lowers the performance of multi-threaded, CPU-heavy tasks on multi-core systems [20]. In data mining, this matters for tasks that would otherwise benefit from thread-level parallelism, such as running cross-validation folds or hyperparameter searches in parallel. For this reason, Python programs often rely on multiprocessing or on external libraries implemented in compiled code, some of which can release the GIL during expensive numerical operations [10].

Python's memory model is another source of overhead. CPython uses reference counting with a cyclic garbage collector for memory management, and Python integers are full objects rather than primitive machine values, which adds memory and runtime overhead [18].

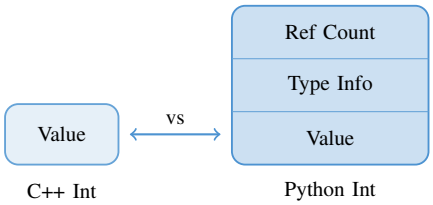


Fig. 3: Illustrative memory comparison between a C++ integer and a Python integer object.

Figure 3 illustrates this overhead. Note that a C++ integer mainly contains the raw value, while a Python integer has additional object-level structure and metadata [18]. This increases memory usage, which can lead to poor cache locality in large datasets [14].

In practice, many performance limitations can be avoided using optimized libraries. For example, NumPy supports vectorized operations implemented in compiled code, avoiding the overhead of standard Python loops [10]. As a result, Python is often used as a high-level interface to faster native implementations [8, 15].

Although the language is typically slower than compiled alternatives, its readable syntax, combined with its rich ecosystem of libraries, makes it highly effective for most data mining tasks [9].

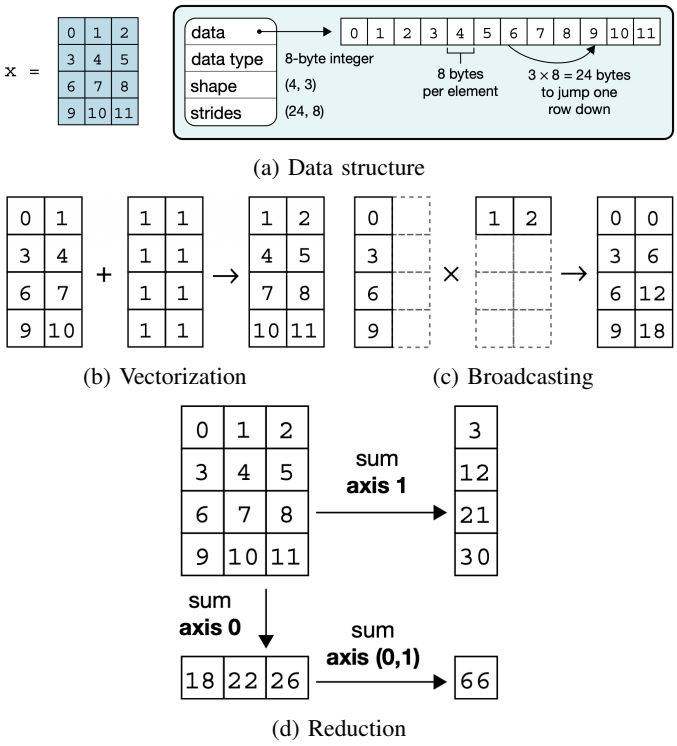
III. The Python Ecosystem for Data Analytics

Python is useful for data analytics not only because of the language itself, but also because of its libraries [9]. There are libraries for each step of a typical data mining process: numerical computation, data preprocessing, statistical analysis, machine learning, data visualization, and deployment. This section discusses NumPy, pandas, SciPy, scikit-learn, Plotly, and Dash as connected parts of this ecosystem.

A. Numerical and Array-Based Computing

NumPy is a fundamental library for data analytics in Python [10]. It makes numerical calculations very efficient. The core data type of NumPy is the `ndarray`, a homogeneous multidimensional array of values of the same data type across one or more dimensions [21]. It is widely used as a common representation for numerical data in scientific computing and machine learning [10, 13].

An `ndarray` stores its data together with metadata, such as the data type, shape, and strides, in a contiguous memory buffer [21]. The data type defines how each value is stored, the shape defines the size of each dimension, and the strides determine how NumPy moves through memory when accessing elements [21]. It also enables slicing operations that create memory views rather than copies, reducing memory usage for large datasets [21].



(a) Data structure

(b) Vectorization

(c) Broadcasting

(d) Reduction

Fig. 4: Core NumPy concepts: memory representation, vectorization, broadcasting, and reduction, adapted from [10].

So, where does NumPy's performance come from? As shown in Figure 4, arrays can be manipulated through vectorized operations, broadcasting, reductions, and compiled low-level execution [21]. Vectorization applies operations to whole arrays rather than relying on slow Python loops. Broadcasting allows arithmetic operations between arrays of different shapes by following specific alignment rules. NumPy compares dimensions from the end, and two dimensions are compatible if they are equal or one is 1. In these cases, the size-1 array is virtually expanded without copying data, usually by adjusting how the data is read rather than actually duplicating it. For example, adding an array of shape $(4, 3)$ and a row vector of shape $(1, 3)$ produces a $(4, 3)$ result in which the row vector is reused across the four rows. By delegating these operations to optimized low-level routines, NumPy significantly improves computational performance [21]. Listing 1 illustrates how vectorization, broadcasting, and reductions are implemented in practice.

```
import numpy as np
X = np.arange(12).reshape(4, 3)
Y = X[:, :2] + 1                # vectorization
Z = X[:, :1] * X[:1, 1:3]       # broadcasting
r1 = X.sum(axis=1)              # row-wise reduction
r0 = X.sum(axis=0)              # column-wise reduction
r = X.sum()                     # full reduction
```

Listing 1: Core NumPy operations

Despite these advantages, NumPy is not designed for every data processing task. Because `ndarray` objects are homogeneous, they are less suited to mixed tabular data containing numerical, categorical, and textual attributes [11]. Additionally, NumPy arrays have fixed sizes, meaning that resizing them typically requires creating a new array and copying the existing data [21]. Finally, NumPy is mainly designed for working with data in memory, so very large datasets or distributed tasks often require tools like Dask. For GPU-based tasks, libraries such as CuPy, PyTorch, or TensorFlow are commonly used [22].

B. Data Manipulation and Preprocessing

NumPy handles numerical arrays efficiently, but real-world data mining tasks usually involve structured, heterogeneous datasets [11]. The pandas library provides the `DataFrame` data structure for such data [23]. A `DataFrame` is a two-dimensional table-like structure with labeled rows and columns. This format follows a common dataset organization, where rows represent individual observations and columns represent specific features [23].

```
import pandas as pd

df = pd.DataFrame({
    "age": [21, 25, None, 30],
    "group": ["A", "B", "A", "B"],
    "score": [70, 85, 75, 90]
})

df = df.dropna()                          # clean
df["score"] = df["score"] / 100           # scale
avg = df.groupby("group")["score"].mean()  # aggregate
```

Listing 2: Data preprocessing with pandas

Unlike a NumPy array, a `DataFrame` can store different data types across its columns [23]. This makes pandas suitable for datasets that combine numerical, categorical, textual, or temporal attributes [11]. It also introduces labeled indexing, allowing rows and columns to be accessed by name rather than by numerical position. Furthermore, pandas automatically aligns operations based on these labels, which helps preserve consistency when combining or transforming datasets [23].

Listing 2 shows typical preprocessing of a dataset before analysis or model training. Missing values are removed using `dropna()`, the numerical feature `score` is scaled, and

the dataset is summarized by `group` using `groupby()` [23]. These operations are important because raw datasets often contain missing values and must be transformed into a structured, clean format before statistical or machine learning methods can be applied [11].

However, the flexibility of pandas comes at a performance cost. Storing a `DataFrame` in memory typically requires more memory than a NumPy array, since it must store explicit row and column labels to support alignment [23]. This overhead can increase when columns use the generic `object` dtype, which is often used for strings or mixed values and stores references to Python objects [23]. Many pandas workflows are also limited by in-memory processing and therefore require additional tools such as Dask, Apache Spark, database systems, or more memory-efficient data formats [24].

Overall, pandas connects raw structured data with the numerical formats used in Python-based data mining pipelines. It is especially useful for cleaning, transforming, and aggregating tabular datasets before passing them to libraries such as SciPy or scikit-learn for modeling and analysis [11].

C. Scientific and Statistical Computing

While NumPy provides numerical arrays and pandas handles tabular manipulation, many data mining tasks also require mathematical algorithms. SciPy extends the Python data analytics pipeline by providing routines for statistics, optimization, sparse matrices, and distance-based computations [12]. Because SciPy operates directly on NumPy arrays, cleaned data from pandas can be converted into numerical matrices and passed to these computations [12].

A primary application of SciPy in data mining is statistical analysis. The `scipy.stats` module provides probability distributions, statistical functions, and hypothesis tests [25]. For example, a two-sample t-test can be used to test whether the means of two distinct groups differ significantly. This helps determine whether an observed difference is statistically significant or due to random variation in the data.

Beyond statistics, SciPy also provides optimization tools [12]. Many analytical problems can be formulated as the minimization of an objective function. In data mining, this often occurs when model parameters are estimated by minimizing a loss function that measures the difference between predicted and observed values. A common example is the least-squares objective:

$$\min_{\theta_0,\theta_1} \sum_{i=1}^{n} \left(y_i - (\theta_0 + \theta_1 x_i)\right)^2, \tag{1}$$

where x_i is an input value, y_i is the observed target value, and θ_0 and θ_1 are parameters to be estimated.

```python
from scipy import stats, optimize
import numpy as np

a = np.array([1.2, 1.5, 1.7])       # group A
b = np.array([2.0, 2.1, 2.4])       # group B
t, p = stats.ttest_ind(a, b)        # t-test

x = np.array([1, 2, 4, 5, 6])       # feature
y = np.array([1, 2, 4, 5, 7])       # target

def loss(theta):                    # squared error
    t0, t1 = theta
    y_pred = t0 + t1 * x
    return ((y - y_pred) ** 2).sum()

res = optimize.minimize(loss, x0=[0, 1])
theta = res.x                       # parameters
```

Listing 3: Statistical testing and optimization with SciPy.

Listing 3 demonstrates statistical testing and optimization using SciPy. The t-test compares groups `a` and `b`, returning the test statistic t and corresponding p-value p [25]. The optimization part fits the linear model $y = \theta_0 + \theta_1 x$ by minimizing the sum of squared errors [25]. In this implementation, θ_0 represents the intercept and θ_1 the slope, illustrating SciPy's role as a numerical layer for statistical analysis and parameter estimation.

SciPy also supports other key data mining tasks. The `scipy.sparse` module provides efficient structures for handling large matrices containing mostly zero values, while `scipy.spatial` offers distance computations for clustering, nearest-neighbor search, and similarity analysis [25].

Although the library has many strengths, SciPy also has several limitations. First, SciPy has no native GPU support, so all computations run on the CPU unless GPU-focused libraries such as CuPy, PyTorch, or TensorFlow are used [22]. Unlike modern deep learning frameworks, SciPy is not designed for automatic differentiation [26]. Finally, SciPy is also not intended for distributed computing or larger-than-memory tasks. For this case, libraries such as Dask, Apache Spark, or database systems are more suitable [24].

Overall, SciPy connects cleaned numerical data with statistical tests, optimization routines, sparse representations, and distance metrics. It is often used alongside pandas, NumPy, and scikit-learn.

D. Machine Learning Frameworks

Building on top of NumPy, pandas, and SciPy, machine learning frameworks provide tools for model training, evaluation, and validation. In the Python ecosystem, scikit-learn is a standard library for classical machine learning and data mining tasks [13]. It provides implementations of supervised learning algorithms, such as classification and regression, as well as unsupervised approaches like clustering and dimensionality reduction [13].

A core design principle of scikit-learn is its unified *estimator* API [27]. Different estimators use the same fundamental methods: `fit` extracts patterns from training data, `transform` maps data into a suitable representation, and `predict` generates outputs from a trained model [27]. This consistent

interface makes it much easier to combine preprocessing steps and learning algorithms.

The following implementation extends the previous tabular dataset by using the `age` feature to predict the numerical target `score`. Listing 4 demonstrates how feature scaling and regression can be applied through the same consistent estimator-style interface.

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

X = df[["age"]]                    # feature
y = df["score"]                    # target

scaler = StandardScaler()
scaler.fit(X)                      # learn mean/std
X_scaled = scaler.transform(X)     # scale feature

model = LinearRegression()
model.fit(X_scaled, y)             # train model
y_pred = model.predict(X_scaled)   # predict
```

Listing 4: Preprocessing and modeling with scikit-learn.

The `StandardScaler` computes the mean and standard deviation of the input features and transforms them to have zero mean and unit variance. Next, `LinearRegression` fits the relationship between these scaled features and the continuous target variable. In more complex workflows, these operations are typically encapsulated within a `Pipeline`, ensuring that preprocessing and model training are executed together in a fixed sequence [27].

scikit-learn has many features, but it is primarily a library for classical machine learning. It does not provide deep learning frameworks, nor does it support automatic differentiation, as in PyTorch and TensorFlow [22, 26]. It also operates primarily on in-memory data structures; large or distributed tasks may require additional tools such as Dask-based extensions or Spark, while GPU support in scikit-learn itself remains limited. Most scikit-learn algorithms require numerical input arrays; therefore, categorical and textual features must be encoded or vectorized before use [27].

Overall, scikit-learn provides a unified layer for applying machine learning algorithms once data has been cleaned, transformed, and numerically represented. Its consistent API and pipeline support make it especially valuable for creating reproducible data mining workflows.

E. Visualization and Deployment

Data mining results should not only be computed, but also presented in a form that users can understand [5]. Python supports this final stage through libraries for interactive visualization and lightweight web deployment.

The Plotly library generates interactive visualizations directly from NumPy arrays or pandas `DataFrame` objects [28]. Unlike static plots, Plotly figures are rendered in the browser and support interactions such as zooming, hovering, and selection. This interactivity is useful for exploring statistical relationships, hidden trends, outliers, and specific patterns within a dataset [5].

```
import plotly.express as px

fig = px.scatter(df, x="age", y="score",
                 color="group")
fig.show()
```

Listing 5: Creating an interactive scatter plot with Plotly Express.

Listing 5 visualizes the relationship between `age` and `score`, using distinct color encodings to represent categorical groups. Such plots help inspect the data before or after model training, making patterns easier to interpret [5].

The Dash framework complements Plotly by providing a Python framework for constructing interactive dashboards and data applications [29]. A typical Dash application can combine visualizations, user controls, analytical tables, and model outputs into a unified web interface. This allows analytical results to be explored without directly interacting with Python code.

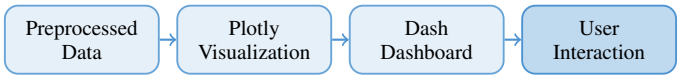


Fig. 5: Visualization and deployment stage in a Python-based data mining workflow.

Figure 5 shows how preprocessed data or model outputs are transformed into interactive visualizations and dashboards. This step helps make data mining results easier to inspect, communicate, and use in practical environments [5].

Beyond dashboards, Python also supports lightweight deployment. The FastAPI library can make a trained model available through a web API, while Dash can present model outputs in an interactive interface [30]. This helps move experimental results toward simple application prototypes.

All of these tools have practical limitations. Plotly figures generated with Python can become slow for very large datasets [28]. Similarly, deploying Dash applications may require additional infrastructure, such as a web server [29]. Deploying FastAPI applications also has limitations, including input validation, memory usage, and deployment setup [30].

Overall, visualization and deployment tools extend the Python data mining ecosystem by turning numerical results into interactive plots, dashboards, and lightweight applications.

IV. Python in Data Mining Workflows

The previous section discussed individual data mining libraries. In practice, data mining is an iterative workflow rather than a simple sequence of isolated library calls [31]. Data must be prepared, models trained, and results evaluated, which means that earlier choices are often revised in response to later outcomes. Python offers a unified environment for all stages of development, making repeated experimentation easier.

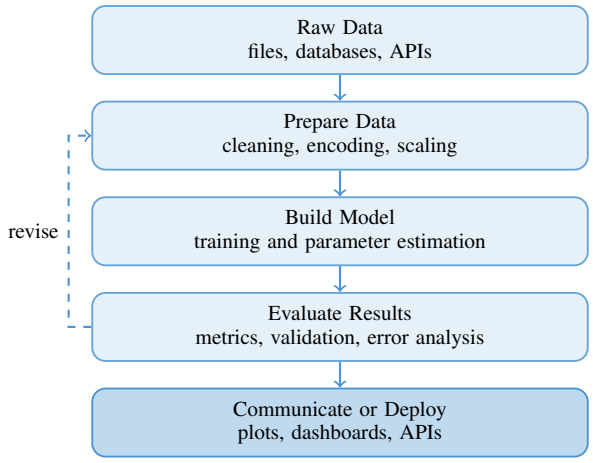


Fig. 6: Iterative structure of a Python-based data mining workflow.

Figure 6 shows a simplified version of the cycles involved in data mining [31]. In reality, data mining rarely ends after a single pass through the data. Evaluation may reveal that missing values were handled poorly, that the selected features are ineffective, or that an alternative model should be considered. In such cases, the workflow returns to earlier stages and is refined iteratively.

The ability to continually refine a workflow is one of Python's greatest strengths, since the same dataset can pass through several pipeline stages within the same programming environment. A dataset can be processed with pandas, represented numerically with NumPy, analyzed statistically with SciPy, modeled with scikit-learn, and visualized with Plotly or Dash. Even though these libraries serve different purposes, they remain highly compatible because they build on shared data structures and follow similar workflows. Interactive environments such as Jupyter notebooks, widely used for exploratory analysis, extend the same workflow by providing an interface for stepwise experimentation [32].

This integration is important because early decisions strongly affect later results [33]. Poor preprocessing can reduce model quality; unsuitable features may fail to reveal hidden patterns; and weak evaluation strategies can lead to misleading conclusions. Python simplifies this process by allowing developers to inspect intermediate results, modify the workflow, and repeat the analysis within the same environment.

Reproducibility is critical in practical data mining. During experimentation, many workflow configurations are evaluated, with different preprocessing steps, feature sets, model parameters, and evaluation metrics [34]. If these choices are not tracked, it becomes difficult to explain why one model performs better than another [33]. Python workflows can be organized into reusable scripts, notebooks, or pipelines, enabling analyses to be repeated and later verified. For larger projects, workflow orchestration and experimentation-tracking tools help record pipeline structure, model parameters, and results [34].

Deployment adds extra requirements. A model in production must handle new input data, provide predictions via an accessible interface, and be continuously monitored [33]. Model performance may degrade when incoming data differs from the training data, a problem known as data drift or concept drift, depending on the situation [35]. To address this, modern Python-based workflows often include monitoring, retraining, and version control steps [36]. Deployment may also involve packaging services, automating pipelines, and integrating models into production systems, all of which require infrastructure beyond that provided by standard analytics libraries.

The workflow, therefore, connects experimentation, evaluation, and deployment. Once a model has been evaluated, its results can be communicated through plots, dashboards, or lightweight applications. Python's key role in data mining is integration: its ecosystem supports the entire iterative process, from raw data and analysis to interpretable and practical results.

V. Conclusion

This paper discussed Python as a software environment for data mining and data analytics. It showed that Python's role in this field is based not only on the language itself, but also on its ecosystem of libraries. Python's readable syntax, flexible programming paradigms, and dynamic style support fast experimentation, while its raw performance is limited by interpreted execution, memory overhead, and runtime type handling.

Many of these performance limits are reduced through optimized libraries. NumPy moves numerical operations into efficient, low-level implementations; pandas supports tabular data manipulation; SciPy adds scientific and statistical routines; and scikit-learn offers an environment for classical machine learning. Plotly and Dash extend this workflow by making results easier to explore and communicate.

The paper also showed that these libraries should not be seen as isolated tools. In practical data mining, they are often combined in an iterative workflow: raw data are cleaned, transformed, represented numerically, analyzed, modeled, evaluated, and revised when necessary. Early decisions, such as missing-value handling or feature selection, can strongly influence the final result.

However, Python is not ideal for every situation. Very large datasets or GPU-heavy workloads may require additional tools or specialized frameworks. For this reason, Python is often combined with optimized libraries or GPU-based frameworks when higher performance or scalability is needed.

Overall, Python's main strength in data mining is integration: it connects preprocessing, numerical computation, statistical analysis, machine learning, evaluation, visualization, and simple deployment into a single workflow. This makes Python effective for data mining projects.

References

[1] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[2] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aaa8415

[3] T. J. Sejnowski, "Data science is an ecosystem," *Harvard Data Science Review*, vol. 6, no. 1, 2024.

[4] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *AI Magazine*, vol. 17, no. 3, pp. 37–37, 1996.

[5] D. A. Keim, "Information visualization and visual data mining," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 1–8, 2002.

[6] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley *et al.*, "Best practices for scientific computing," *PLoS Biology*, vol. 12, no. 1, p. e1001745, 2014.

[7] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, pp. 23–29, 11 2000.

[8] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[9] I. Stančin and A. Jović, "An overview and comparison of free python libraries for data mining and big data analysis," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 977–982.

[10] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[11] W. McKinney, "Data structures for statistical computing in python," *SciPy 2010*, 2010. [Online]. Available: https://doi.org/10.25080/Majora-92bf1922-00a

[12] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[14] Q. Zhang, L. Xu, X. Zhang, and B. Xu, "Quantifying the interpretation overhead of python," *Science of Computer Programming*, vol. 215, p. 102759, 2022.

[15] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2010.

[16] Python Software Foundation, "The python language and c-api reference manual (v3.14)," 2026, pSF Reference documentation Suite. [Online]. Available: https://docs.python.org/3/

[17] R. Dyer and J. Chauhan, "An exploratory study on the predominant programming paradigms in python code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 684–695.

[18] G. Barany, "Python interpreter performance deconstructed," in *Proceedings of the Workshop on Dynamic Languages and Applications*, 2014, pp. 1–9.

[19] D. Beazley, "Understanding the python gil," in *PyCon Python Conference*, 2010, pp. 1–62.

[20] S. Gross, "PEP 703 – Making the Global Interpreter Lock Optional in CPython," https://peps.python.org/pep-0703/, 2023, accessed 2026-05-23.

[21] NumPy Developers, "Numpy v2.2 reference and internals manual," 2026, numPy Community Documentation. [Online]. Available: https://numpy.org/doc/

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[23] The pandas development team, "pandas v2.2 api reference and user guide," 2026, pyData Framework Documentation. [Online]. Available: https://pandas.pydata.org/docs/

[24] E. Shaikh, I. Mohiuddin, Y. Alufaisan, and I. Nahvi, "Apache spark: A big data processing engine," in *2019 2nd IEEE Middle East and North Africa Communications Conference (MENACOMM)*. IEEE, 2019, pp. 1–6.

[25] SciPy Developers, "Scipy v1.15 reference guide," 2026, sciPy Community Documentation Suite. [Online]. Available: https://docs.scipy.org/doc/scipy/

[26] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. [Online]. Available: https://jmlr.org/papers/v18/17-468.html

[27] Scikit-learn Developers, "scikit-learn v1.6 user guide and object apis," 2026, scikit-learn Development Documentation. [Online]. Available: https://scikit-learn.org/stable/

[28] Plotly Technologies Inc., "Plotly open source graphing library and express interface reference," 2026. [Online]. Available: https://plotly.com/python/

[29] ——, "Dash developer user guide: Callbacks and deployment frameworks," 2026. [Online]. Available: https://dash.plotly.com/

[30] S. Ramírez, "Fastapi framework documentation suite," 2026. [Online]. Available: https://fastapi.tiangolo.com/

[31] R. Wirth and J. Hipp, "Crisp-dm: Towards a standard process model for data mining," in *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining*, vol. 1. Manchester, 2000, pp. 29–39.

[32] A. Rule, A. Birmingham, C. Zuniga, I. Altintas, S.-C. Huang, R. Knight, N. Moshiri, M. H. Nguyen, S. B. Rosenthal, F. Pérez *et al.*, "Ten simple rules for writing and sharing computational analyses in jupyter notebooks," p. e1007007, 2019.

[33] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[34] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the Machine Learning Lifecycle with MLflow," *IEEE Data Engineering Bulletin*, vol. 41, no. 4, p. 7, 2018.

[35] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–37, 2014.

[36] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine learning operations (mlops): Overview, definition, and architecture," *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023.