Comparing Object-Oriented Principles in Python and Java for Machine Learning Frameworks

Walker Narog, Shuchi Shah

University of Colorado Boulder, Computer Science Department

430 UCB, 1111 Engineering Dr.

Boulder, Colorado 80309

Abstract

This paper compares Python's and Java's implementation of object-oriented principles in the machine learning context, by their impact on design, performance, and scalability. Focusing on encapsulation, inheritance, polymorphism, and abstraction, we evaluate popular frameworks such as Scikit-learn (sklearn) in Python and Weka and Smile in Java using the Iris dataset. Our findings show Java achieves faster prediction times (0.1ms vs. 1.8ms for Logistic Regression) and higher accuracy (99.47% vs. 96% for SVM), while Python excels in training speed (1.8ms vs. 24.4ms for SVM). This study provides insight into trade-offs for both languages and provides guidance to ML developers based on project requirements.

Code — Github Repository
Datasets — Iris Species Dataset

Introduction

The rise of machine learning has made Python and Java two of the most popular programming languages used in machine learning (ML), both having different unique strengths and design philosophies. Python, being dynamically typed and flexible, has become a favorite for rapid prototyping and research. In contrast, Java, which is statically typed and strong, is being used heavily for large-scale applications. One of the essential characteristics of both languages is their implementation of object-oriented (OO) principles (encapsulation, inheritance, polymorphism, and abstraction) that form the core of ML framework design and function. OO principles matter in ML because they promote modularity and reusability, which are vital for creating complex algorithms. How these principles are implemented in Python and Java must be understood in order to evaluate their impact on framework performance, scalability, and maintainability. This paper seeks to address the research question: *How do Python and Java differ in implementing object-oriented principles, and how do these differences influence the design and performance of ML frameworks?* We use the Iris dataset for comparison due to its simplicity and widespread adoption, ensuring clarity in evaluating OOP implementations across languages. By answering these questions, we aim to provide insight that can help guide developers in the selection of languages for their ML applications.

Implementation and Comparison of Object-Oriented Principles

The implementation of object-oriented principles in Python and Java shows slight differences. **Encapsulation** is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates (GeeksforGeeks). Encapsulation in Python can be seen when a class has its own methods/functions that can be private or public within it. Protected methods have a single underscore at the beginning (i.e., _age), private methods have the double underscore (i.e., __age), and finally, no underscore means public (i.e., age). Private attributes (e.g., *__species, __age*) and controlled access via getter/setter methods, as demonstrated in the *Animal* class below:

```
class Animal:
    def __init__(self, species, age):
        self.__species = species
        self.__age = age

    def get_species(self):
        return self.__species

    def set_species(self, species):
        self.__species = species

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age >= 0:
            self.__age = age

animal = Animal("Dog", 3)
print("Species:", animal.get_species())
print("Age:", animal.get_age())

animal.set_species("Cat")
animal.set_age(5)

print("Updated␣Species:", animal.get_species())
print("Updated␣Age:", animal.get_age())
```

Private methods provide the capability to make the objects' complexity hidden (Nzerue-Kenneth et al, 2023). Java encloses using the private keyword, and the variables and data are fetched through getter and setter methods, as seen in the *EncapsulatedAnimal* class:

```
class EncapsulatedAnimal {
    private String species;
    private int age;

    public EncapsulatedAnimal(String species, int age) {
        this.species = species;
        this.age = age;
    }

    public String getSpecies() {
        return species;
    }

    public void setSpecies(String species) {
        this.species = species;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}


public class Main {
    public static void main(String[] args) {
        EncapsulatedAnimal animal = new EncapsulatedAnimal("Dog", 3);
        System.out.println("Species:␣" + animal.getSpecies());
        System.out.println("Age:␣" + animal.getAge());

        animal.setSpecies("Cat");
        animal.setAge(5);

        System.out.println("Updated␣Species:␣" + animal.getSpecies());
        System.out.println("Updated␣Age:␣" + animal.getAge());
    }
}
```

Java's higher level of controlling access enables the data to be more secure against unwanted modifications. Both languages include public, protected, and private members (Jain 2023). Java also includes another default access modifier, offering better control (Saleh, Zykov, and Legalov 2021). For example, in our Python implementation, the BaseClassifier class encapsulates data loading and preprocessing, while Java's BaseClassifier uses private fields like trainingData with explicit getters, enforcing stricter access control. Hence, while Python offers flexibility in encapsulation, Java's stricter access controls enhance security and maintainability.

Inheritance is the capability of a class to derive properties and characteristics from another class (GeeksforGeeks). Python supports multiple inheritance, allowing a subclass like *Dog* to override methods from its parent *Animal*:

```
class Animal:
    def speak(self):
        return "Some␣sound"


class Dog(Animal):
    def speak(self):
        return "Bark"


dog = Dog()
print(dog.speak())
```

This promotes code reusability and flexibility. Python also supports single, multilevel, hierarchical, and hybrid inheritance (Nzerue-Kenneth et al, 2023), allowing developers to develop intricate class hierarchies. Java restricts inheritance to single inheritance but compensates with interfaces, which allow classes to implement multiple behaviors without the complexities of multiple inheritance. For example, *InheritedDog* extends *AnimalBase*:

```
class AnimalBase {
    String speak() {
        return "Some␣sound";
    }
}

class InheritedDog extends AnimalBase {
    String speak() {
        return "Bark";
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("\nInheritance");
        AnimalBase myDog = new InheritedDog();
        System.out.println(myDog.speak());
    }
}
```

This design choice in Java allows class hierarchies to remain simple and conflict potential low, thus making code simpler to extend and maintain. Both languages make use of the super() function to call parent class functionality, but Java's single inheritance model is designed with clarity and structure in mind, while Python's multiple inheritance allows for greater flexibility (Saleh, Zykov, and Legalov 2021). Ultimately, Python's multiple inheritance enables rapid prototyping, whereas Java's single inheritance with interfaces prioritizes code stability and maintainability.

Polymorphism allows objects to behave differently based on their specific class type. Polymorphism is achieved differently since they have varying typing systems. Python is dynamically typed, allowing functions and operators to behave differently based on their class type. This makes Python decide the type of function or operation at runtime dynamically, and thus very flexible to employ for quick prototyping and experimenting in ML frameworks. For example, as seen with the *sound()* method in *Cat* and *Dog*:

```
class Cat:
    def sound(self):
        return "Meow"


class Dog:
    def sound(self):
        return "Bark"


def make_animal_sound(animal):
    print(animal.sound())


make_animal_sound(Cat())
make_animal_sound(Dog())
```

Java differs, as it is statically typed, but allows for any subclass to be treated as an instance of its superclass. This makes polymorphism heavily reliant on interfaces and inheritance, allowing subclasses to be treated as instances of their superclass or interface. This makes Java more flexible, allowing a method to modify any subclass of the type it works with. The following Java example demonstrates polymorphism using an AnimalSound interface, where both Cat and PolymorphicDog implement the same method (*sound()*) but provide different behaviors:

```
interface AnimalSound {
    void sound();
}

class Cat implements AnimalSound {
    public void sound() {
        System.out.println("Meow");
    }
}

class PolymorphicDog implements AnimalSound {
    public void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
    System.out.println("\nPolymorphism");
        AnimalSound cat = new Cat();
        AnimalSound dog = new PolymorphicDog();
        cat.sound();
        dog.sound();
    }
}
```

As seen in both code snippets, polymorphism adds flexibility to any language. Allowing developers to write more generic and reusable code while maintaining type safety where applicable.

Abstraction is providing only essential information about the data to the outside world, hiding the background details or implementation. Abstraction is also handled differently in the two languages. Python does not natively support abstract classes but provides the ABC (Abstract Base Classes) module and the @abstractmethod decorator to define abstract methods. As seen in this example:

```
from abc import ABC, abstractmethod


class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass # Blueprint


class Dog(Animal):
    def make_sound(self):
        return "Woof"


d = Dog()
print(d.make_sound())
```

This allows developers to create blueprints for classes, ensuring that derived classes implement specific methods while hiding unnecessary details. Java, on the other hand, uses the *abstract* keyword to declare abstract classes and methods, enforcing a clear structure for inheritance and implementation. This explicit support for abstraction in Java promotes better design patterns and modularity, making it easier to manage complex ML frameworks. As seen here:

```
abstract class Animal {
    abstract void makeSound();
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof");
    }
}

public class Main {
    public static void main(String[] args) {
    System.out.println("\nPolymorphism");
        System.out.println("\nAbstraction");
        Animal abstractDog = new Dog();
        abstractDog.makeSound();
    }
}
```

Both languages achieve abstraction effectively, but Java's explicit syntax enforces design rigor, while Python's ABC module balances flexibility with structure.

Machine Learning Framework

ML frameworks in Python and Java typically entail an organized process of five fundamental steps: data collection, data processing, model design, execution, and deployment (Tripathi 2021). The process starts with the collection of raw data from different sources and then pre-processing data, filtering data, transforming data, and encoding data to prepare it for modeling. Following the formatting of data, the process entails choosing and implementing an efficient model that is then trained, tested, and optimized at runtime. Finally, the model is deployed for real-world applications, where its performance is evaluated in practical scenarios. This approach ensures that ML frameworks are scalable, maintainable, and capable of handling complex tasks efficiently.

Case Studies

In order to compare Python and Java using their respective ML frameworks, we measured training time, prediction time, and the accuracy of the three models in each language. To measure training and prediction time, we used the time library in Python and System.time from Java. The accuracy was measured using the Sklearn library's metrics.accuracy_score() feature for Python and the Weka library's classifiers.evaluation feature.

In order to compare the languages' utilization of object-oriented principles like maintainability, adhering to DRY/KISS (Don't Repeat Yourself/Keep it Simple, Stupid), coupling, and the preference for composition over inheritance as our metrics. These principles are important for assessing long-term code quality, extensibility, and developer productivity, especially in real-world ML that evolves.

Here, we compare the performance and implementation of machine learning models applied to the iris species dataset, a popular benchmark dataset in the ML community. We used the Support Vector Machines (SVM), Logistic Regression, and Decision Tree models in both Python and Java. Support Vector Machine (SVM) creates the best boundary between different classes and functions even with complex data. Logistic Regression predicts probabilities (e.g., whether something belongs in a specific class) using an S-shaped curve and is therefore best for simple yes/no decisions. Decision Trees separate data into branches based on significant features, and so they are easy to follow, but can end up too specialized to the training data if not controlled. Both methods have their strengths depending on the problem. Our goal was to select models that are implemented across both libraries with comparable defaults and configurations to ensure an equal comparison. These models were chosen due to their simplicity and interpretability, along with their widespread use in classification problems, which makes them ideal to compare the object-oriented nature and the performance of Python and Java ML frameworks. These models' shared use across languages allows us to isolate how OOP implementations affect performance.

Each model takes multiple hyperparameters. We carefully aligned these parameters across both implementations when possible, and when defaults differed, we documented and justified the differences to maintain consistency in behavior.

The SVM takes four hyperparameters: kernel, regularization (C), tolerance, and gamma. The kernel in an SVM determines the type of decision boundary used to separate data points. In Python, the kernel is set to 'radial basis function' (RBF), which is a non-linear kernel that maps data into a higher-dimensional space, allowing the model to create complex decision boundaries. In contrast, Java's default kernel is 'linear', which assumes a straight-line decision boundary. Regularization is a technique used to prevent overfitting by controlling the complexity of the model. In SVM, the C parameter decides the trade-off between the margin maximization (distance between classes) and the minimization of classification errors. Regularization is set to 1.0 in both Python and Java. The tolerance is set as 1e(-3) in both languages. Tolerance defines the threshold below which the algorithm stops if the changes in model parameters become sufficiently small, indicating convergence. Lastly, the gamma is auto-scaled in Python, and it is not needed in Java because of the linear kernel. Gamma controls the impact of each point on the model, such that higher values create a more complex, sensitive decision boundary and lower values create a smoother, less complex model. While Python and Java share SVM hyperparameters like regularization (C) and tolerance, their kernel choices differ, with Python using a non-linear RBF and Java defaulting to a linear kernel, resulting in inherently distinct decision boundaries and Python's approach being more adaptable for complex datasets. These kernel and regularization variations illustrate how Python's flexibility enables sophisticated model tuning, whereas Java's strict defaults ensure consistency in production environments.

The Logistic Regression model takes in four hyperparameters as well: regularization, max iterations, tolerance, and solver. These hyperparameters control model convergence, optimization strategy, and generalization strength, all of which influence both performance and accuracy. The regularization is set to L2 in both Python and Java. L2 regularization, also known as Ridge regularization, incorporates a penalty term proportional to the square of the weights into the model's cost function. This encourages the model to evenly distribute weights across all features, preventing overreliance on any single feature and thereby reducing overfitting. ("How Does L1 and L2 Regularization Prevent Overfitting," 2024). In both languages, the max number of iterations is 1000 in this case, but can be set to anything. Iterations refer to the number of steps the algorithm will take in adjusting model parameters during optimization. Tolerance is set to 1e(-4) in Python and does not need to be set in Java, as Weka uses an internal optimizer. The solver is set to 'lbfgs' in Python and doesn't need to be set in Java because of the internal optimizer. The variations in how each of these hyperparameters is controlled reflect the philosophies of the supporting designs. Python's scikit-learn favors explicit, user-level control, while Weka depends on abstraction and internally controlled optimization strategies.

Finally, the Decision Tree model takes in four hyperparameters: the split criterion, max depth, pruning, and minimum samples split. These parameters control how the tree is built and how much it generalizes versus overfits

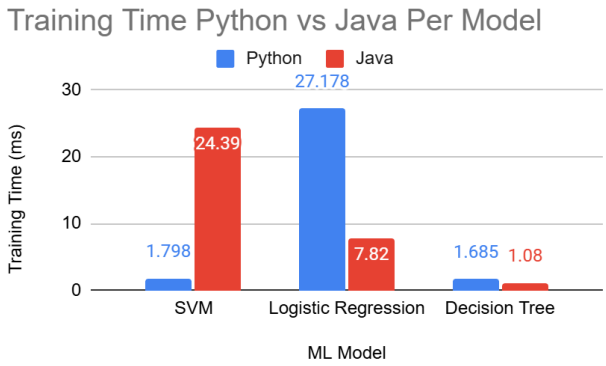
the data. In Python, the split criterion was set to 'gini', which measures impurity. In Java, a confidence factor of 0.25 was used, which indirectly controls splits by determining pruning thresholds. The split criterion is used to decide how to split the data at each node. Max depth had no limit and was unlimited in both languages. Max depth is the maximum number of levels or layers the tree can have. Both languages had pruning disabled. Pruning is the process of removing branches from the tree after it is built to prevent overfitting by simplifying the model and reducing its complexity. The minimum samples to split was set to two in both Python and Java. This is the minimum number of samples required to create a new split at a node. Whereas both languages reach the same tree structure, Python's clear split criteria, as opposed to Java's confidence factors, reflect their contrasting philosophies relating to interpretability versus automation.

Experimental Setup and Methodology

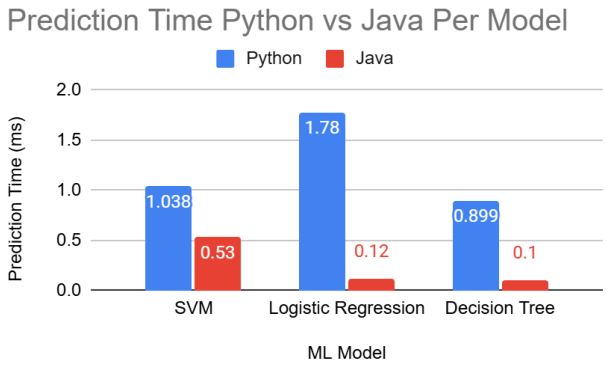
To ensure a fair and consistent comparison between the Python and Java implementations, the following steps were followed in both versions:

1. **Reading CSV data:** The iris dataset was loaded from a CSV file using language-specific tools, pandas in Python, and CSVLoader in Java (Weka).
2. **Splitting into train/test sets:** The dataset was randomly split into training and testing using a 70/30 ratio in both languages.
3. **Training classifiers:** Three classifiers- Support Vector Machines (SVM), Logistic Regression, and Decision Tree- were trained using scikit-learn in Python and Weka/Smile in Java.
4. **Running each model multiple times:** Each model was executed multiple times (100 times) with different randomized splits to ensure consistent results and minimize bias due to sampling.
5. **Measuring accuracy and timing:** Training and prediction times were recorded using time in Python and System.nanoTime in Java. Accuracy was measured using metrics.accuracy_score (Python) and Evaluation (Java).
6. **Averaging the results:** The performance metrics (accuracy, training time, prediction time) were averaged across all runs to provide a more reliable and robust comparison.

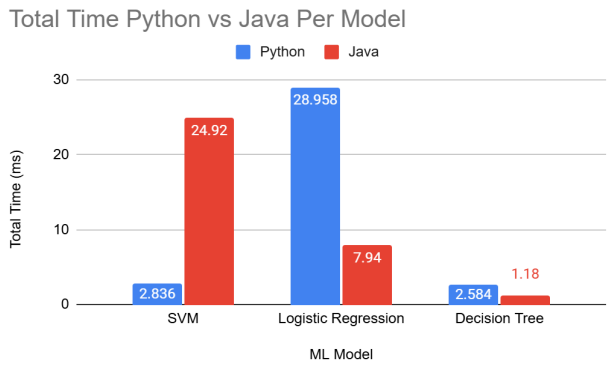
Results and Analysis



Training time is employed to quantify how long it took each model to acquire patterns on the Iris data, recorded from initialization to the end of the fitting process. Our study resulted in some interesting numbers relating to training time. For example, Python took minimal training time when using the SVM model, but took significantly longer when using the Logistic Regression model. Java was the opposite, as it was slow to train using SVM and shorter for Logistic Regression. The training time for the Decision Tree model was quick for both languages.



Prediction time quantifies how long each trained model takes to classify a new Iris sample, averaged across 100 test instances. The data recorded for the prediction time each model took to predict weighed heavily in Java's favor. Python was slower than Java in each model and was especially slow using the Logistic regression model. Although everything was fairly quick, as they all took less than 2 milliseconds.



Total execution time combines training and prediction durations, reflecting end-to-end system efficiency for each language. The overall time each language took is heavily reliant on the time it took to train each model and thus looks similar to that graph. Java took much longer using the SVM model and was fairly quick using Logistic Regression, and Python was vice versa. Both languages had a quick overall time for the Decision Tree model, though.

Analysis of Object-Oriented Principles

Encapsulation is shown in both implications by bundling data and methods within classes. The Python *BaseClassifier* class encapsulates data loading (*split_data(), prepare_features()*) and evaluation logic (*evaluates()*), and internal attributes like *training_memory* and *prediction_time* are managed within the class. There is tighter encapsulation in Java with private variables (e.g., *trainingTime*, *testData*) being accessed by getters (e.g., *getTrainingTime()*). Subclasses such as *LogisticRegressionClassifier* encapsulate Weka model instances (e.g., Logistic) even further, making only necessary methods such as *train()* and *predict()* available. This maintains data integrity and modular design.

Both implementations use inheritance by taking advantage of a base class (*BaseClassifier*) to combine common operations such as data splitting and evaluation. Python child classes (e.g., *SVMClassifier*, *LogisticRegressionClassifier*) inherit data processing from *BaseClassifier* and extend *train()*/*predict()* for model-specific functionality (e.g., *self.model.fit()*). Similarly, Java subclasses like DecisionTreeClassifier and *SVMClassifier extend BaseClassifier* with reuse but override its data loading suitably (e.g., *decisionTree.buildClassifier()*). This method reduces redundancy and makes adding new classifiers easier.

Polymorphism is used in both languages. In Python, *run_classification()* dynamically calls *train()* and *predict()* on any *BaseClassifier* subclass (e.g., *SVMClassifier*, *LogisticRegressionClassifier*) without knowing the implemented model. *SVMClassifier* and *LogisticRegressionClassifier* have different implementations for *train()* (*svm.SVC()* vs. *LogisticRegression*), but are both called the same way. In Java, Main.java's *testClassifier()* method employs polymorphism through the use of a *BaseClassifier* reference, where any subclass (for example, *SVMClassifier*, *DecisionTreeClassifier*) is subject to testing through the same *evaluate()* method. This simplifies the task of benchmarking and future extension.

Abstraction simplifies complexity by exposing only high-level operations. The Python *BaseClassifier* abstracts away data preprocessing (e.g., *prepare_features()*), allowing child classes to focus solely on model training. Users interact with streamlined methods like *train()* without needing to understand CSV parsing or memory tracking. In Java, *BaseClassifier* hides Weka-specific details (e.g., Instances, CSVLoader) behind abstract methods like *getClassifier()*, requiring subclasses to define their model while shielding users from low-level intricacies. This reduces cognitive overhead. For instance, one can deploy *LogisticRegressionClassifier* without grappling with Weka's Logistic class internals.

Analysis of Software Design Principles

The findings provide a mixed comparison of Python and Java on several key software design principles, including maintainability, adherence to DRY/KISS principles, coupling, and the application of composition over inheritance. On maintainability, Python's modular design, as evident in the run_classification function, promotes readability and conciseness, which makes the code more understandable and easier to modify for programmers. However, Python's dynamic typing can cause errors at runtime, thus compromising maintainability since bugs won't be detected until runtime. In our code, Python's run_classification function is concise (20 lines) but lacks type hints, while Java's Main.java requires verbose type declarations (e.g., Instances dataset) but catches type mismatches at compile time. Conversely, Java's modularity in Main.java is benefited by strong typing and compile-time checks that help detect errors early and enhance maintainability. Nevertheless, Java's verbosity decreases readability since it makes the code more difficult to scan and understand immediately. Therefore, Python's brevity enables up-front development speed, while Java's compile-time protections offer better guarantees for long-term system support.

Adhering to DRY/KISS principles, Python demonstrates its strength by the employment of reusable functions like evaluate in the BaseClassifier, which means clean and tidy code appropriate for these principles. Python's non-enforcement typing sometimes means code can be duplicated in big or ill-disciplined codebases. Java, on the other hand, enforces DRY/KISS principles more rigidly

through reusable functions like evaluate and cross-validate the BaseClassifier. While this makes it more maintainable and reusable, Java's verbosity at times causes code duplication since programmers may find it a hassle to refactor or abstract the duplicated code due to the inherent boilerplate in the language.

Both languages achieve low coupling between the BaseClassifier and its subclasses, but each language does it differently. Python's dynamic typing reduces coupling because it allows for greater flexibility in component interaction, though the lack of compile-time checking can inadvertently introduce coupling if type incompatibilities or interface revisions are not detected until runtime. Java's strong type-checking capabilities also reduce coupling by forcing dependencies to be explicit and obvious. However, Java's rigid class hierarchy can sometimes contribute to coupling, as parent class modifications can require extensive modifications in subclasses.

Lastly, both languages enable composition over inheritance highly, although with varying trade-offs. The usage of Python's composition through classes like SVMClassifier, LogisticRegressionClassifier, and DecisionTreeClassifierStyle follows a plain syntax that uses the dynamic type system of the language to facilitate reusable and adaptable components. However, this flexibility can be abused by developers since the developers can unconsciously rely on inheritance even when composition should be used instead. In Java, composition is also used effectively in similar classes with the benefit of strong type-checking and compile-time error checking, which ensures correct implementation. However, verbosity in Java makes the implementation of composition more difficult, with more explicit and thorough code than in Python.

Conclusion

Our work reveals distinct trade-offs between Java and Python when using object-oriented principles for machine learning. Python's minimal syntax and dynamic typing promote prototyping velocity and code conciseness, while Java's static typing and encapsulation enforce robustness at the cost of verbosity. They did reflect in performance measures: Python performs best in training time (1.8ms vs. 24.4ms for SVM), while Java performs faster in prediction times (0.1ms vs. 1.8ms for Logistic Regression) and slightly in accuracy (99.47% vs. 96% for SVM).

Python/sklearn is preferable for research-centered workflows where iteration speed and ecosystem tools (e.g., matplotlib, pandas) outweigh deployment concerns. Its dynamic nature and concise syntax help accelerate experimentation, though runtime type safety remains a trade-off. Java/Weka stands out in production environments demanding strict type safety, memory control, or JVM compatibility. Our results demonstrate its superior prediction latency and accuracy, critical for high-throughput applications. The choice between the two languages ultimately depends on the specific needs of the project and the trade-offs that developers are willing to accept.

In the future, this could be extended by comparing deep learning frameworks (e.g., TensorFlow vs. Deeplearning4j) or larger datasets to assess scalability. Additionally, the frameworks in both languages could be stress tested on larger and more complex datasets.

References

Albatera, P. J. T.; Soñega, J. Y.; Daluyon, K. D. O.; and Lincopinis, D. 2024. *Analyzing Python and Java for Artificial Intelligence Development: A Comparative Study. International Journal of Research Publication and Reviews* 4(10): 1151–1167. https://www.researchgate.net/publication/381157614_Analyzing_Python_and_Java_for_Artificial_Intelligence_Development_A_Comparative_Study.

Ghinea, G.; Semwal, V. B.; and Khandare, A., eds. 2025. *Intelligent Computing and Networking.* Lecture Notes in Networks and Systems, volume 1172. Springer, Singapore. https://doi.org/10.1007/978-981-97-8631-2.

"Introduction of Object Oriented Programming" *GeeksforGeeks*, 09 Feb 2023, https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/

Jain, A. 2023. *Comparative Analysis of Java and Python in Machine Learning: Investigate and Compare the Suitability and Performance of Java and Python for Machine Learning Tasks. International Journal of Research Publication and Reviews* 4(10): 1151–1167. https://doi.org/10.55248/gengpi.4.1223.123305

Kumar, P.; and Sharma, A. 2023. *Understanding of Machine Learning with Deep Learning: Architectures, Workflow, Applications, and Future Directions. Computers* 12(5): 91. https://doi.org/10.3390/computers12050091

Lodhi, Y. R., & Shukla, S. V. S. (2024). Applying object-oriented principles to machine learning frameworks for improved scalability. *International Journal of Science, Innovation, and Engineering, 1*(3), 10-17. https://www.ijsci.com/index.php/home/article/view/21.

Mishra, Aryan. "How Does L1 and L2 Regularization Prevent Overfitting?" *GeeksforGeeks*, 14 May 2024, https://www.geeksforgeeks.org/how-does-l1-and-l2-regularization-prevent-overfitting/.

Nzerue-Kenneth, P. E.; Onu, F. U.; Denis, A. U.; and Igwe, J. S. 2023. *Detailed Study of the Object-Oriented Programming (OOP) Features in Python. British Journal of Computer Networking and Information Technology* 6(1): 83–93. https://www.researchgate.net/publication/376049332_Detailed_Study_of_the_Object-Oriented_Programming_OOP_Features_in_Python.

Quraishi, R.; Salian, A.; Khasgiwala, D.; and Khandare, A. 2025. *Comparing Advanced Programming Languages Based on Machine Learning (ML)*. In Ghinea, G.; Semwal, V. B.; and Khandare, A., eds., *Intelligent Computing and Networking*, volume 1172. Springer, Singapore. https://doi.org/10.1007/978-981-97-8631-2_6.

Rane, N.; Mallick, S. K.; Kaya, Ö.; and Rane, J. 2024. *Machine Learning and Deep Learning Architectures and Trends: A Review.* In *Applied Machine Learning and Deep Learning: Architectures and Techniques*, 1–38. Deep Science Publishing. https://www.researchgate.net/publication/385095492_Machine_learning_and_deep_learning_architectures_and_trends_A_review.

Saleh, H., Zykov, S., & Legalov, A. (2021). Eolang: Toward a new Java-based object-oriented programming language. I. Czarnowski, R. J. Howlett, & L. C. Jain (Eds.), *Intelligent Decision Technologies* (Vol. 238, pp. 373–380). Springer. https://doi.org/10.1007/978-981-16-2765-1_30

Tripathi, A., Singh, A. K., Singh, K. K., Choudhary, P., & Vashist, P. C. (2021). Machine learning architecture and framework. In K. K. Singh, M. Elhoseny, A. Singh, & A. A. Elngar (Eds.), Machine Learning and the Internet of Medical Things in Healthcare (pp. 1-22). Academic Press. https://doi.org/10.1016/B978-0-12-821229-5.00005-7