Simple Classifier

Gunjan Dayani[1], Brian Catraguna[2],
Jayesh Gaur[3], Junda Ai[4], Shreyash Gondane[5]
Electrical and Computer Engineering
New York University

April 2023

1 Background

Machine learning is a powerful tool that enables computers to learn patterns from data and make predictions or decisions without being explicitly programmed. Classification is a common machine learning task that involves assigning labels to input data based on a set of predefined categories. There are several classification algorithms that can be used to solve this task, including logistic regression, decision trees, random forests, support vector machines (SVM), k-nearest neighbors (k-NN), and neural networks.

The goal of this project is to implement and compare different classification algorithms using Python programming language. The algorithms will be trained and tested on a given or predefined dataset, and their performance will be evaluated using different accuracy metrics such as accuracy score, precision, recall, F1 score, and confusion matrix. The performance of each algorithm will be compared and analyzed to determine which algorithm performs better for the given dataset.

In order to accomplish this task, we will start by importing the necessary libraries and loading the dataset. We will then preprocess the data by removing any missing or irrelevant data, and transforming the categorical features into numerical values. Next, we will split the data into training and testing sets using a user-defined method, such as splitting by a fixed size or using a random state to ensure reproducibility.

After preprocessing the data and defining the split, we will implement and train each classification algorithm on the training set, and evaluate its performance on the testing set using different accuracy metrics. We will repeat this process for each algorithm and compare their performance.

Finally, we will present our results and analysis, discussing the strengths and weaknesses of each algorithm, and identifying the best algorithm for the given dataset. This project will provide a practical introduction to machine learning classification algorithms and their performance evaluation using Python, while also allowing the user to customize the splitting method for their specific needs.

2 Existing Work

Lazy Predict is a Python library designed to simplify the machine learning process by automating model selection and evaluation tasks. It offers a user-friendly interface that allows individuals, even those without extensive programming knowledge, to quickly prototype and explore different machine learning algorithms. By abstracting away complexities, Lazy Predict reduces the barrier to entry for newcomers and provides a range of classification and regression algorithms to choose from. It generates performance metrics for each model and facilitates comparative analysis, enabling users to make informed decisions about the most suitable algorithms for their tasks. With functionality for hyperparameter tuning and integration with popular machine learning libraries, Lazy Predict has become a popular choice for streamlining and expediting the machine learning workflow.

While Lazy Predict offers convenient automation and simplification of the machine learning process, it does have certain limitations. Firstly, Lazy Predict may not support all the advanced customization options and specific requirements that experienced users might need for their machine learning tasks. Additionally, it may not handle complex feature engineering or preprocessing tasks, which could limit its effectiveness in certain domains. Lastly, as a general-purpose library, Lazy Predict may not be optimized for highly specialized or domain-specific machine learning problems, potentially impacting its performance in those scenarios.

3 Project Goals

The primary objective of this library is to empower individuals new to machine learning, regardless of their proficiency in Python or programming. Our aim is to simplify the complexities of the machine learning process, enabling users to delve into the world of classification algorithms without the need for extensive coding knowledge.

Our library excels in streamlining the entire machine learning pipeline, encompassing crucial stages such as data preparation, data splitting, model training, and model evaluation. By exposing a single user-friendly class, we enhance accessibility and manageability, enabling users to navigate the process effortlessly.

One distinguishing feature of our library is the flexibility it offers in terms of algorithm selection. Users have the option to specify their preferred machine learning algorithm or leverage the capability to run multiple algorithms simultaneously on their dataset. This functionality empowers users to explore various algorithmic approaches, broadening their understanding and aiding in decision-making.

Additionally, our library incorporates an automatic hyperparameter tuning mechanism. This mechanism automates the process of finding the most optimal parameters for each model, saving users valuable time and effort. By leveraging this functionality, users can effortlessly identify the best configurations for their chosen algorithms, enhancing the performance and accuracy of their models.

Upon completion of the machine learning process, our library provides users with a comprehensive comparison among the best-performing models for the given dataset. This valuable insight facilitates easier decision-making and empowers users to identify the most effective algorithm that aligns with their specific requirements. With this information at their disposal, users can make informed choices and confidently select the algorithm that best suits their needs.

4 System Overview

4.1 UML Diagram

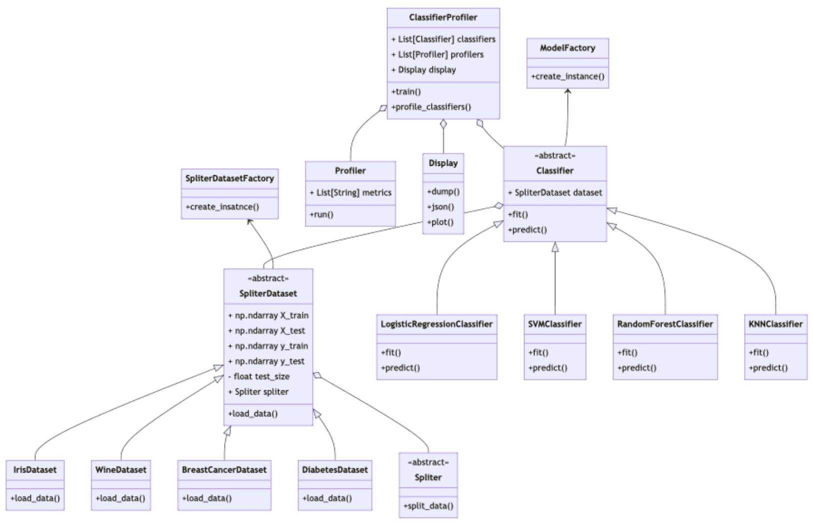


Figure 1: UML class diagram for our classifier library.

Our software design is composed of different modules that can be used as dependencies to each other. The program starts with creating a dataset, which is represented by the abstract class `SpliterDataset`. This class does not depend on anything and can be instantiated by the user. Alternatively, the user can choose to use predetermined datasets which are already available from the `sklearn` library.

The `SpliterDataset` class allows the user to split the dataset using a

`Spliter` object, which is responsible for performing the actual splitting. The resulting train and test sets are stored as properties `X_train`, `X_test`, `y_train`, and `y_test`. Different subclasses of `SpliterDataset` implement different ways of splitting the data. These concrete implementations can be obtained using the `SpliterDatasetFactory` class, which implements the Factory pattern.

The dataset object is then injected into the classifier, which is represented by the abstract class `Classifier`. Different subclasses of `Classifier` implement different machine learning algorithms, such as logistic regression, SVM, random forest, and KNN. In this class, the dataset is trained using the `X_train` data and the `y_train` labels. The classifier can then be used to make predictions.

The `Classifier` class is constructed using the `ModelFactory`, which is responsible for creating instances of the different classifiers. These classifiers can then be used by the `ClassifierProfiler` class, which is responsible for profiling the performance of the different algorithms on the determined dataset. The profiler also needs a list of metrics, such as accuracy, balanced accuracy, and ROC AUC. The results of the profiling can be visualized using the `Display` class.

The `ClassifierProfiler` class is composed of a list of classifiers, a list of profilers, and a display object. The `train()` method is responsible for training all of the classifiers using the dataset, and the `profile_classifiers()` method is responsible for running the profilers on the trained classifiers.

5 User Input

To use the "Simple Classifier" library, users can provide input through a YAML configuration file. The YAML file allows users to customize various aspects of the machine learning process, including the choice of classifiers, dataset name, splitting strategy, test size, profile metrics, and display format.

Users can specify the desired classifiers by listing their names under the "classifier_names" section in the YAML file. The library supports multiple classifiers, and users can include as many as they prefer.

The dataset name should be specified under the "dataset_name" field in the YAML file, indicating the dataset to be used for training and evaluation.

Users can choose the splitting strategy for dividing the dataset into training and testing sets. The available strategies include "percentage" and "random." For the percentage strategy, users can set the test size (e.g., 0.50 for a 50% test size) under the "test_size" field.

Under the "profile_metrics" section, users can specify the metrics they want to evaluate and compare for the classifiers. Metrics such as accuracy, precision, recall, and F1-score can be included.

The display format can be selected using the "display_format" field. The options include "dump" and potentially others, depending on the library's functionality.

To execute the "Simple Classifier" library with the specified YAML configuration file, users can use the command-line interface (CLI) by running the following command:

```
python simpleclassifier -y <PATH_TO_YAML_CONFIG_FILE>
```

Users can modify these configurations in the YAML file according to their specific needs and preferences.

```
classifier_names:
  - knn
  - decision_tree
dataset_name: iris
splitting_strategy: percentage
test_size: 0.2
profile_metrics:
  - accuracy
  - precision
display_format: dump
```

In this example, the YAML file specifies the following configurations:
The classifiers to be used are ”knn” and ”decision_tree”. The dataset to be used is ”iris”. The splitting strategy is set to ”percentage”, meaning the dataset will be split into training and testing sets based on the specified test size. The test size is set to 0.2, indicating a 20The metrics to be profiled and compared for the classifiers are ”accuracy” and ”precision”. The display format is set to ”dump”, which will display the results in a specific format.

6 Development Process

Our team follows Agile software development principles, which enable us to efficiently plan, design, develop, test, review, and deploy our code in iterative cycles. This approach allows us to deliver high-quality software in a collaborative and organized manner.

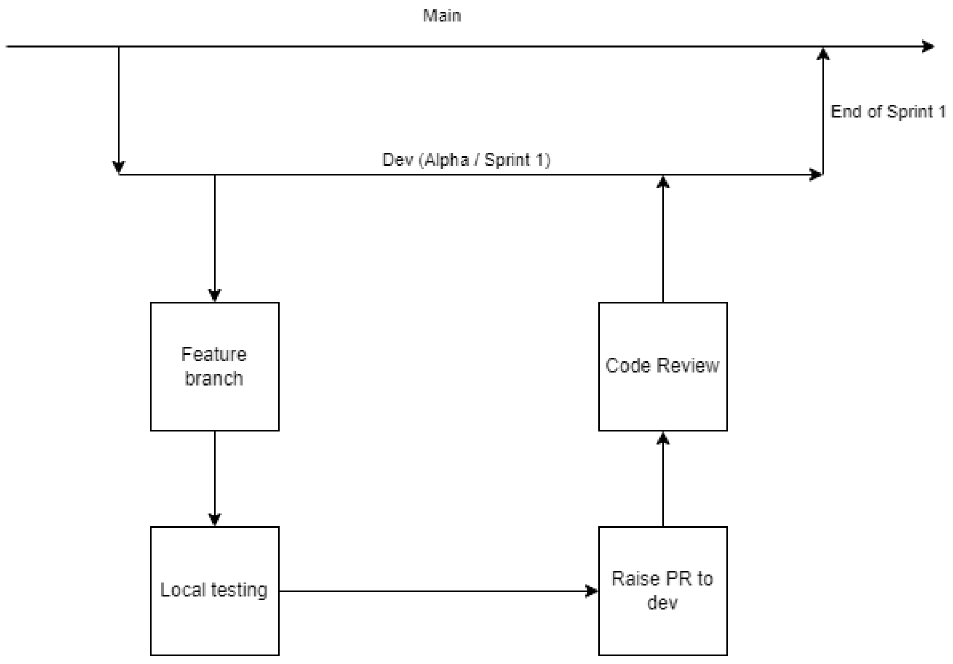
To facilitate our development process, we conduct weekly meetings to plan our sprints and assign tasks to team members. This ensures that everyone knows their responsibilities and can work on specific features or improvements. By dividing the work into manageable tasks, we can maintain focus and productivity throughout the development cycle.

For version control, we utilize Git in conjunction with GitHub for efficient code sharing and collaboration. We adopt a feature branch workflow, which allows us to work independently on our assigned tasks without conflicts. Each team member creates their own branch to implement their feature or fix, keeping the main branch stable.

When a task is complete, we open a pull request on GitHub, allowing other contributors to review the code. This collaborative review process ensures that the implementation aligns with expectations and follows best practices. Any suggested changes or improvements can be discussed and addressed before merging the code into the main branch. Our Git workflow can be summarized as follows:

1. Start with the main branch as the stable base.
2. Create a feature branch for each task or feature.
3. Work on the feature branch independently, making frequent commits.
4. Push the feature branch to the remote repository.
5. Open a pull request on GitHub for code review.
6. Collaborate on the review process and make necessary changes.
7. Once the reviewer approves the code, merge the feature branch into the main branch.

The main branch reflects the stable and tested version of the code. By following this Git workflow and utilizing the collaborative features of GitHub, we ensure that our code is thoroughly reviewed, maintainable, and continuously improved.



Tests are efficiently written using the pytest framework, utilizing fixtures and the mark.parametrize decorator to set up and create input test cases. Continuous integration is achieved through GitHub Actions, which triggers test execution upon any commits made to the main branch and pull requests directed towards the main branch.

Documentation is meticulously built using Sphinx, enhanced with the visually appealing and user-friendly Furo third-party theme. The documentation process involves automatic generation from docstrings within the source code, allowing easy reference and navigation. Cross-referencing is also implemented, enabling seamless navigation from code references to their corresponding explanations. Moreover, comprehensive user guides are included to facilitate users in understanding and configuring various aspects of the tool. To provide convenient online access, the documentation is deployed as a static website using Google Cloud Storage.

7 Lessons Learned

Throughout the development of this project, several valuable lessons were learned, highlighting the importance of various aspects of the software development life cycle. These lessons have significantly enhanced our understanding and proficiency as software developers.

One key lesson learned was the significance of thorough system design before diving into the coding phase. By investing sufficient time in the design phase, we were able to avoid potential issues and rework. The complexity of the problem at hand necessitated frequent design changes, and had we neglected this due diligence, the existing code would have required frequent rewriting. This experience reinforced the importance of proper planning and design in ensuring a robust and adaptable codebase.

For team members with limited exposure to environmental science problems, this project served as a valuable learning experience in understanding the mathematical complexity inherent in such domains. The challenge of representing an environmental system accurately involves considering a multitude of factors, similar to the curse of dimensionality. Recognizing and addressing this complexity improved our ability to develop solutions that reflect the real-world intricacies more effectively.

Furthermore, the project exposed us to a diverse range of software tools and best practices. We gained proficiency in generating documentation using Sphinx, implementing Continuous Integrations, and adhering to Python's best practices. The hands-on experience with these tools and practices significantly enhanced our overall software development skills and allowed us to adopt industry-standard approaches.

8 Future Work

Expansion of Performance Metrics: In addition to the existing metrics such as accuracy and precision, incorporating additional evaluation metrics such as the confusion matrix, recall, and F1-score can provide a more comprehensive assessment of the classifiers' performance. This would enable users to gain deeper insights into the model's behavior and its strengths and weaknesses.

User-Friendly Visualization: Enhancing the project with visually appealing outputs can greatly enhance the user experience. Visualizations such as interactive plots, ROC curves, and feature importance charts can provide intuitive representations of the classifier's performance and aid in result interpretation. Adding visualization capabilities would make the library more user-friendly and facilitate better understanding of the classification models.

Also, Building a user interface (UI) for the "Simple Classifier" library can further enhance its accessibility and ease of use. A graphical interface could allow users to interact with the library's functionalities without requiring extensive knowledge of the command-line interface. The UI could provide a seamless experience for configuring classifiers, selecting datasets, and visualizing results, making the library more accessible to a broader audience.

Showcasing the "Simple Classifier" library on a dedicated website can increase its visibility and provide a platform for users to explore its features and capabilities. The website could serve as a central hub for documentation, tutorials, and sample code, allowing users to access resources and stay updated on the latest developments. Additionally, the website can serve as a community platform for user discussions, feedback, and contributions.

As machine learning evolves, integrating advanced techniques such as ensemble learning, deep learning models, or transfer learning into the "Simple Classifier" library can expand its capabilities and provide more diverse options for users. Incorporating these techniques would enable users to tackle complex classification problems and explore cutting-edge approaches within the user-friendly environment of the library.

9 References

1. Atlassian. "Comparing Workflows - Feature Branch Workflow." Available at: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
2. Sphinx Documentation. "Sphinx Tutorial." Available at: https://www.sphinx-doc.org/en/master/tutorial/index.html
3. scikit-learn Documentation. Available at: https://scikit-learn.org/stable/