AutoPlumber: A Data Preprocessing Pipeline Optimizer

William Andrian Dharma T - 13523006
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: williamadt123@gmail.com , 13523006@std.stei.itb.ac.id

***Abstract*—A crucial part in machine learning is data preprocessing. Choosing the appropriate pipeline before modelling could significantly improve model accuracy. However, with the number of possibilities available to choose from, an optimized search algorithm would be beneficial to reduce search time. AutoPlumber aims to solve this problem by creating an optimized search algorithm for common data preprocessing techniques to quickly test possibilities with a simple user setup.**

***Keywords*—Machine Learning, Data Science, Optimization**

I. Introduction

Data Science has been a rapidly growing field in the last few decades joined with the increase of data collection by services and companies around the world. A comprehensive analysis of the data possessed can lead to an improved understanding of the problem at hand and reduces guessing in decision-making. From analysis, oftentimes modelling is done as the next step of the operation to do classification or regression tasks as some examples. Before data is passed on to the model, it has to be preprocessed in order to make the data operable by the model, or to improve the predictive quality of the model. Examples of preprocessing include imputation, outlier removal, encoding, and many more. The abundance of options available to choose from makes this task a rather cumbersome one, especially if handling with large amounts of data, where each trial run could take a considerable amount of time. Trying out every possible combination of techniques on every feature using a bruteforce search would then be very time consuming and not efficient. With that problem in mind, this library is built with the goal of an creating an optimzed preprocessing pipeline searcher, which could be very useful in the early stages of modelling to act as a baseline to build from. As most of the existing optimization frameworks deal with model selection and hyperparameter tuning such as *Hyperopt* [1] and *Optuna* [2], we believe that this library could be a novel solution in this field.

The searching algorithm implemented here will be variations of greedy algorithms with different heuristics which would reduce the search space in order to search more efficiently. State nodes would contain the current pipeline chosen and would expand into nodes based on the available options that can be taken in the next step of the preprocessing. A bruteforce option would be available for the user to choose to ensure an exhaustive search of all options.

II. Preliminaries

A. *Tree*

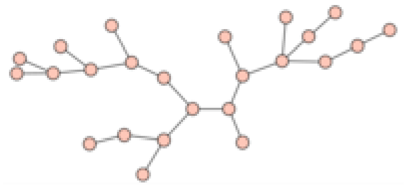

Fig 1. An example of a tree

Source: Some tree.svg. *Encyclopedia of Mathematics.* URL: http://encyclopediaofmath.org/index.php?title=Some_tree.svg&oldid=5255

A tree is a connected graph which does not contain any circuits [3]. Following that definition, several properties can arise such as:

1. For every pair of vertices in a tree, there is only one path
2. A tree with n vertices has n − 1 edges
3. A tree with a distinguished vertex is called a rooted tree, with the distinguished vertex being called the root
4. A vertex with a degree of one is called a pendant vertex
5. A tree with two or more vertices will have at least two pendant vertices

The tree we will be working with will be a rooted tree, with the root being the initial state of the data. For the purposes of this paper, we will refer a vertex as a node, and a pendant vertex as a leaf node instead.

Edges between nodes can be unweighted or weighted. A weighted edge could represent a cost to transition between two nodes.

B. *Search Tree*

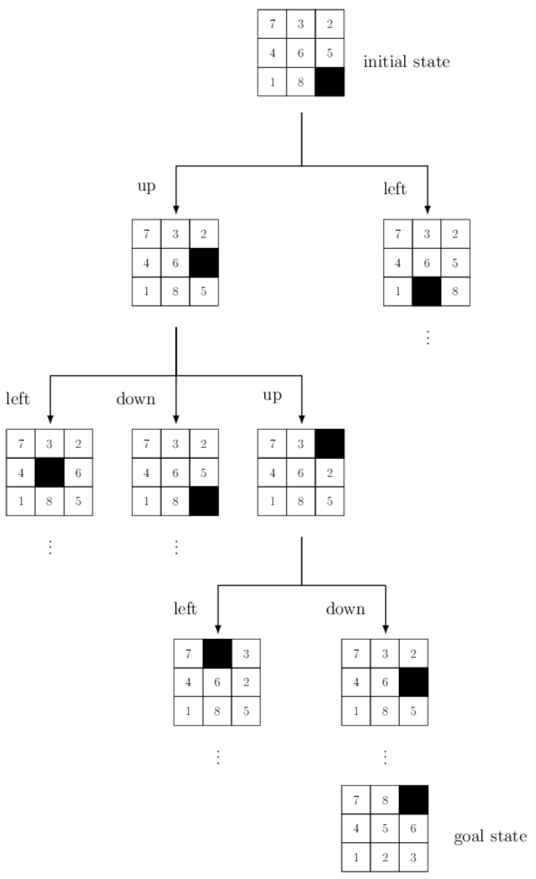

Fig 2. Search tree of a classic 8-puzzle problem
Source: Multi-Agent Route Planning in Grid-Based Storage Systems - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Expansion-tree-of-an-8-puzzle_fig10_322721630

A search tree is an application of a tree for searching problems. Instead of having a singular value, each node has its value representing a state of the problem, with the leaf nodes being the ending states. A state which satisfies the requirements of a goal is called the goal state. A search tree is built by first defining the initial state of the problem. It then adds children nodes by expanding the node, which is the act of listing every valid possible action that can be taken in a the given node.

C. *Greedy Algorithm*

A greedy algorithm is a type of algorithm which is commonly used as a fast and simple way to solve optimization problems. It works by choosing the best immediate solution on every step of the search until it reaches an end state. In other words, a greedy algorithm will choose the local optima at a moment.Selection of the option is based on the cost of transitioning to the state by using a heuristic.

A heuristic is a shortcut strategy to solve a problem quicker than an exhaustive search by trading optimality and completeness for speed. It is usually made using domain knowledge in conjunction with trial and error for a specific problem. By using heuristics, we reduce the search space significantly, but it also means we risk not finding the global optima of a problem.

A greedy algorithm has the following elements:

1. Candidate Set: A set of available choices to choose from in a given step
2. Solution Set: A set of chosen candidates
3. Solution Function: A function to determine if the candidate set can form a solution
4. Selection Function: A function to select a candidate with a heurstic.
5. Feasibility Function: A function to check if the selected candidate can be added to the solution
6. Objective function: A function to evaluate the solution, maximize or minimize.

III. Methodology

AutoPlumber will be built using the python programming language and will also use other libraries such as pandas, numpy, scikit-learn, and others.

A. *Preprocessors*

We provide several classes as wrappers for common preprocessing techniques such as imputation, outlier removal, encoding, scaling. Each feature in the training dataset will have these objects in a chain and together will make up the state of the search tree.

B. *Imputation*

Imputation is the technique of filling out missing values in a dataset. Missing values are commonplace in real world data and thus a method of handling it is necessary. Sometimes rows with missing values are simply dropped from the dataset as a simple solution, however this could reduce the input data significantly if many rows contain missing values. As such, imputation strategies are usually used to handle missing values to preserve the size of the input data.

The imputer class is a class to store the imputation strategy of a single column. It can impute using a strategy such as mean imputation, mode imputation, or median imputation, or we could give it a constant value to impute for example -1, to represent that the data was missing.

```python
import pandas as pd

class Imputer:
    def __init__(self, strategy="mean", constant=None):
    def fit(self, X:pd.Series):
    def transform(self, X:pd.Series) -> pd.Series:
    def fit_transform(self, X:pd.Series) -> pd.Series:
```
Fig. 3. Imputer class methods

C. *Outlier Removal*

Real world data will usually have outliers. They are values with a significant discrepancy than the expected distribution of the vast majority of the data. Careful handling of these values are crucial since an outlier doesn't necesarrily mean an invalid data, and removing them could lead to a loss of information. However, not removing outliers could end up creating noise for the model and reduce its accuracy.

Several methods can be used to identify and handle outliers. Z-score based outlier removal uses a certain Z-score threshold to classify values as outliers. For example, a threshold of 3 means that values over 3 standard deviations away from the mean will be considered as outliers. However, a caveat of this technique is that the data needs to be normally distributed for it to work well.

Another method is the Interquartile Range (IQR) based outlier removal. An outlier is defined as values below the lower bound and above the upper bound which are defined as:

$$Lower\ Bound = Q1 - Threshold * IQR$$

$$Upper\ Bound = Q3 + Threshold * IQR$$

This method is more robust to skewness in the data, which is when the distribution has a long tail.

Both methods have different end results after detecting the outliers. We could either drop the row entirely or just cap the outlier values to the threshold they break.

```python
import pandas as pd
class ZScoreOutlierRemover:
        def __init__(self, threshold=3,capped=False):
        def fit(self, X:pd.Series):
        def transform(self, X:pd.Series)-> pd.Series:
        def fit_transform(self, X) -> pd.Series:

class IQROutlierRemover:
        def __init__(self, threshold=1.5,capped=False):
        def fit(self, X:pd.Series):
        def transform(self, X:pd.Series) -> pd.Series:
        def fit_transform(self, X:pd.Series) -> pd.Series:
```

Fig. 4. Outlier removal classes

D. Encoding

Categorical data is not directly operable by a model. The data needs to be encoded into a numerical representation. Common encoding techniques such as one-hot encoding, label encoding, and target encoding, are commonly used. Although general guidelines such as using one-hot encoding for low unique valued data and label encoding for high uniqued valued data exist, unfortunately in practice sometimes the results do not align with them. So, again trial and error is used to determine the best action that can be taken. Here we wrap sklearn's encoders to add additional behaviours such as a maximum category limit for the one-hot encoder.

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder as SkLabelEncoder
from sklearn.preprocessing import OneHotEncoder as SkOneHotEncoder

class LabelEncoder:
    def __init__(self):
    def fit(self, X: pd.Series):
    def transform(self, X: pd.Series) -> pd.Series:
    def fit_transform(self, X: pd.Series) -> pd.Series:

class OneHotEncoder:
    def __init__(self, drop_first=False, max_categories=10):
    def fit(self, X: pd.Series):
    def transform(self, X: pd.Series) -> pd.DataFrame:
    def fit_transform(self, X: pd.Series) -> pd.DataFrame:

class TargetEncoder:
    """Target encoder for categorical variables (mean encoding)."""
    def __init__(self, smoothing=1.0):
    def fit(self, X: pd.Series, y: pd.Series):
    def transform(self, X: pd.Series) -> pd.Series:
    def fit_transform(self, X: pd.Series, y: pd.Series) -> pd.Series:
```

Fig. 5. Encoder classes

E. Scaling

Numerical data can be further transformed to improve the quality of a model. They are used to alter the distribution of the data and are crucial for distance based models for example Support Vector Machine (SVM) or k-Nearest Neighbors (KNN), to ensure every numerical column is on the same scale. Log transformations are also useful scaling techniques that can be used to reduce skewness of data. Our pipeline searcher works with sklearn's scalers via an adapter and custom scalers can also work by matching the fit, transform, and fit_transform functions.

```python
class SeriesAdapter:
    def __init__(self, scaler):
    def fit(self, X):
    def transform(self, X):
    def fit_transform(self, X):

```

Fig. 6. Scaler series adapter

F. Pipeline

A ColumnPipeline is a class which contains the processing done to a single column. They are then stored together in a DataFramePipeline class to represent the total preprocessing done to the dataset.

G. AutoPlumber

The main class of the library is the AutoPlumber class. It contains the methods to setup and run the data preprocessing optimization. The constructor asks for the model, scoring, cv, maximum iterations, and early stopping rounds. From there, we define the preprocessing options we want to search from divided into imputation, outlier removal, encoding, and scaling.

We commence the search by calling the fit() function on our dataset. A high-level overview of the search is as follows:

1. Start with an empty pipeline as the initial state
2. In each iteration, find if an improvement can be achieved for each column
3. Choose the best improvement found in the iteration
4. Repeat until no improvement can be achieved or if we have reached the maximum number of iterations

This algorithm is a greedy algorithm that chooses only the best transformation on a column in every iteration. This means the pipeline is built incrementally every iteration and a previously transformed column can get another replacement transformation if the largest improvement is found on it during the current iteration. A more detailed explanation will be discussed in the following section.

H. Search Algorithm

1) Initialization

```
current_pipeline = DataFramePipeline()  # Starts empty
current_score = -np.inf
columns_to_optimize = list(X.columns)
no_improvement_count = 0 # Tracks stagnation
```

Fig. 7. Search initial state

2) Iterative Optimization

For every iteration, we conduct a greedy search on every column to find the best transformation we can do currently. We then evaluate the newly modified pipeline and compare it with the last iteration's score and we update our pipeline if it performs better.

```
for iteration in range(self.max_iterations):
    if self.verbose:
        self.logger.info(f”\\n--- Iteration {iteration + 1} ---")

    improved = False
    best_iteration_score = current_score
    best_iteration_pipeline = None

    # Try optimizing each column
    for column_name in columns_to_optimize:
        if self.verbose:
            self.logger.info(f”Testing column: {column_name}”)

        # Find best pipeline for this column
        best_column_pipeline, column_score = self._greedy_search_column(
            column_name, X, y, current_pipeline
        )

        if best_column_pipeline is not None:
            # Create new pipeline with this column optimization
            test_pipeline = current_pipeline.copy()
            test_pipeline.add_column_pipeline(column_name, best_column_pipeline)

            try:
                test_pipeline.fit(X, y)
                test_score = self._evaluate_pipeline(test_pipeline, X, y)

                if test_score > best_iteration_score:
                    best_iteration_score = test_score
                    best_iteration_pipeline = test_pipeline
                    improved = True

                    if self.verbose:
                        self.logger.info(f”  Improvement found! Score: {test_score:.4f}”)

            except Exception as e:
                if self.verbose:
                    self.logger.warning(f”  Pipeline failed: {str€}”)

    # Update current best if improved
    if improved and best_iteration_pipeline is not None:
        current_pipeline = best_iteration_pipeline
        current_score = best_iteration_score
        no_improvement_count = 0

        if self.verbose:
            self.logger.info(f”New best score: {current_score:.4f}”)
    else:
        no_improvement_count += 1
        if self.verbose:
            self.logger.info(“No improvement in this iteration”)

    # Record iteration
    self.search_history_.append({
        ‘iteration’: iteration + 1,
        ‘score’: current_score,
        ‘improved’: improved
    })

    # Early stopping
    if no_improvement_count >= self.early_stopping_rounds:
        if self.verbose:
            self.logger.info(f”Early stopping after {no_improvement_count} iterations without improvement”)
        break

# Store best results
self.best_pipeline_ = current_pipeline
self.best_score_ = current_score
self.is_fitted = True
```

Fig. 8. Iterative optimization code

For every column, every possible transformation available for choosing is tried with the current DataFramePipeline to evaluate the score.

```python
def _greedy_search_column(
    self,
    column_name: str,
    X: pd.DataFrame,
    y: pd.Series,
    current_pipeline: DataFramePipeline
) -> Tuple[ColumnPipeline, float]:
    """
    Perform greedy search for the best preprocessing pipeline for a single column.
    """
    column = X[column_name]
    column_type = self._detect_column_type(column)

    if self.verbose:
        self.logger.info(f"Optimizing column '{column_name}' (type: {column_type})")

    # Get all possible transformer combinations for this column
    transformer_combinations = self._get_applicable_transformers(column, column_type)

    best_score = -np.inf
    best_column_pipeline = None

    # Try each transformer combination
    for i, transformers in enumerate(transformer_combinations):
        try:
            # Create column pipeline
            column_pipeline = ColumnPipeline(column_name, transformers)

            # Create test pipeline
            test_pipeline = current_pipeline.copy()
            test_pipeline.add_column_pipeline(column_name, column_pipeline)

            # Fit and evaluate
            test_pipeline.fit(X, y)
            score = self._evaluate_pipeline(test_pipeline, X, y)

            if score > best_score:
                best_score = score
                best_column_pipeline = column_pipeline

            if self.verbose and i % 5 == 0:
                self.logger.debug(f"  Tested {i+1}/{len(transformer_combinations)} combinations")

        except Exception as e:
            if self.verbose:
                self.logger.warning(f"  Combination {i} failed: {str(e)}")
            continue

    if self.verbose:
        self.logger.info(f"  Best score for '{column_name}': {best_score:.4f}")
    return best_column_pipeline, best_score
```
Fig. 9. Column greedy search

3) *Example*

We have an example dataset we would like to optimize with the columns: [“num_1”, num_2”, “cat_1”, “cat_2”], column names starting with num being numerical columns and cat being categorical columns. We try out every transformer combination based on our options for every column and we list the best ones from each of them before choosing to transform the best column:

1. Iteration 1:
    a. Num_1: [impute_mean + standard_scale], score = 0.65
    b. Num 2: [impute median + robust scale], score = 0.72, best score
    c. Cat 1: [impute mode + label encode], score = 0.58
    d. Cat 2: [one hot encode], score = 0.61
2. Iteration 2 current:{num_2:[impute median + robust scale]}:
    a. Num 1: [impute mean + log scale], score = 0.78, best score
    b. Num 2: [impute mean + min max scale], score = 0.70
    c. Cat 1: [target encode], score = 0.76
    d. Cat 2: [one hot encode], score = 0.75
3. Iteration 3 current:{num_1:[impute mean + log scale], num_2: [impute median + robust scale]}, continue until done

IV. Benchmarks

To demonstrate our library, we will test on the titanic dataset, https://www.kaggle.com/c/titanic/data, with a random forest model and a logistic regression model. A basic preprocessed dataset with median imputed values for the numerical columns and mode imputed values for the categorical columns which are then encoded using a label encoder, is chosen as the control benchmark which will be compared against the AutoPlumber optimized search. We will be using accuracy as the scoring metric and a cross validation count of 5 splits.

A. *Data Distribution*

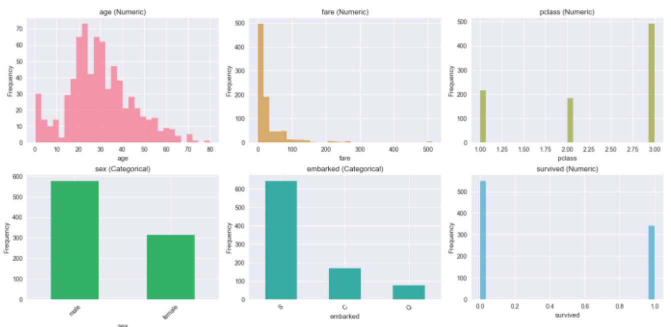

Fig. 10. Data distribution of the titanic dataset

B. *Missing Values*

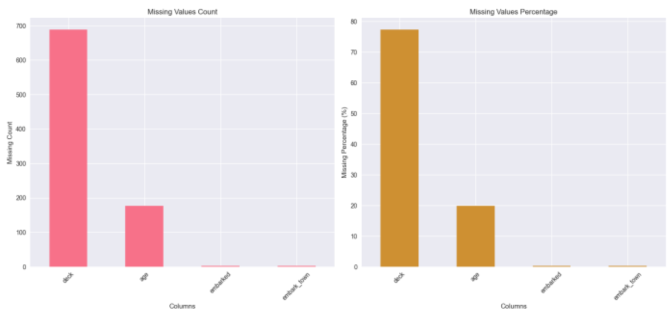

Fig. 11. Missing values of the dataset

C. *Results*

	Method	CV Score	Test Accuracy	CV diff	Test diff
Random Forest	Basic	0.7936	0.7709	+3.25%	+6.96%
	AutoPlumber	**0.8203**	**0.8286**		
Logistic Regression	Basic	0.7978	**0.8045**	+3.17%	-0.69%
	AutoPlumber	**0.8231**	0.7989		

Table 1. Accuracy result comparison

Based on the benchmarks shown on Table 1. , we can observe that AutoPlumber successfully increased our cross validation scores for both models approximately by 3 percent. Test accuracy increased by 6 percent for our random forest model, a 0.69 percent decrease happened to our logistic regression model, although the cross validation score should be the main metric of testing a model's performance. Looking with our cross validation scores, the logistic regression is chosen as the best model of this benchmark, and further analysis on it will be done. Visualizations of the improvement done in each iteration can be displayed to gain a deeper understanding of the optimization process.

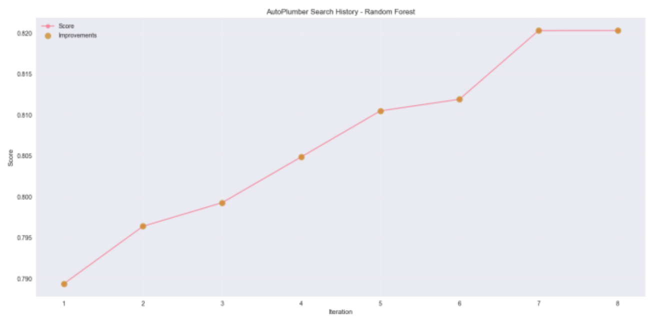

Fig. 12. Random forest iteration improvements

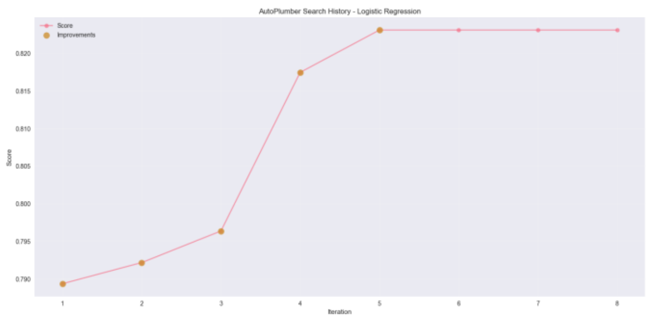

Fig. 13. Logistic regression iteration improvements

We can see from Fig. 12 and Fig. 13 that the model performance will improve on every iteration until it is not possible.

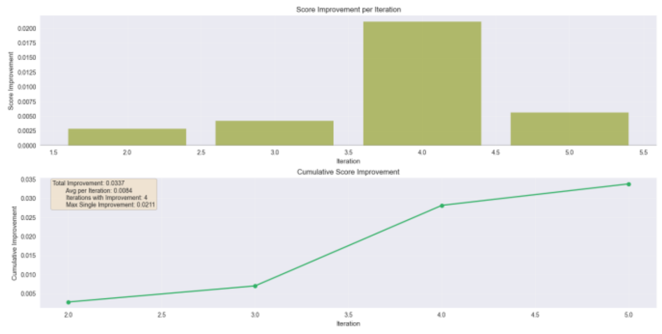

Fig 14. Cumulative improvement graph

ACKNOWLEDGMENT

We would like to thank my lecturer Dr. Nur Ulfa Maulidevi, for teaching us clearly and comprehensively.

REFERENCES

[1] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a Python library for model selection and hyperparameter optimization. Computational Science & Discovery, 8(1):14008, 2015.

[2] Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019, July). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining* (pp. 2623-2631).

[3] IDeo, Narsingh (1974), *Graph Theory with Applications to Engineering and Computer Science* (PDF), Englewood, New Jersey: Prentice-Hall, ISBN 0-13-363473-6,.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juli 2025

William Andrian Dharma T - 13523006

[illegible]