Uniform Cost Search as a Strategy for Hyperparameter Optimization

Wilson Yusda - 13522019
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13522019@std.stei.itb.ac.id

***Abstract*—Hyperparameter tuning is crucial for enhancing the performance of machine learning models, yet it remains a challenging task due to the vast search space. This research explores the use of Uniform Cost Search (UCS) for hyperparameter tuning, systematically prioritizing parameter configurations based on their potential to improve model performance. UCS is applied to a dataset to find the optimal hyperparameters and the results are compared with those obtained using RandomizedSearchCV. Through a series of structured experiments, this study evaluates the effectiveness of UCS in navigating the hyperparameter space efficiently. The findings provide insights into the advantages and potential of UCS as a strategic tool for hyperparameter optimization in machine learning.**

Keywords—Uniform Cost Search; Hyperparameter Tuning; Machine Learning Optimization; Model Performance

I. INTRODUCTION

Hyperparameter tuning is the process of selecting the best set of hyperparameters for a machine learning model to optimize its performance. Unlike model parameters, which are learned during the training process, hyperparameters are set prior to training and can significantly influence the behavior and accuracy of the model. These include settings such as learning rate, number of trees in a random forest, or the regularization strength in a regression model. Effective hyperparameter tuning involves systematically searching through a predefined space of hyperparameter values to identify the combination that yields the best performance on a validation dataset.

For regression models, common hyperparameters include the regularization strength, learning rate (for gradient descent-based methods), and the number of features to consider. For decision trees and ensemble methods like random forests and gradient boosting, key hyperparameters include the maximum depth of the tree, the number of trees in the ensemble, the minimum number of samples required to split a node, the learning rate (for boosting algorithms), and the subsample ratio of the training instances. Additionally, parameters like the maximum number of features considered for splitting and the regularization term to prevent overfitting are crucial for these models. These hyperparameters must be carefully tuned to balance model complexity and performance, ensuring the model generalizes well to new data.

The process of finding these parameters involves systematic exploration of the hyperparameter space. Traditional methods include grid search and random search, which systematically or randomly sample the parameter space. However, Uniform Cost Search (UCS) is also a powerful method for hyperparameter tuning. UCS uses a cost-based approach to prioritize parameter configurations, allowing it to efficiently explore the most promising areas of the hyperparameter space. By assigning costs inversely related to model performance, UCS focuses on configurations that are likely to yield higher accuracy. This method not only streamlines the tuning process but also helps in avoiding overfitting by systematically evaluating and selecting the best parameter combinations.

In this paper, the writer present a meticulously cleaned and model-engineered dataset, which serves as the foundation for our hyperparameter tuning experiments using the XGBoost model. We employ Uniform Cost Search (UCS) to identify the optimal hyperparameters, leveraging its systematic cost-based approach to prioritize the most promising configurations. To assess the effectiveness of UCS, we compare its results with those obtained using RandomizedSearchCV. This comparison highlights the strengths and potential advantages of UCS in efficiently navigating the hyperparameter space and optimizing model performance.

II. THEORETICAL BASIS

A. *Uniform Cost Search*

Uniform Cost Search (UCS) is a search algorithm that addresses the problem of finding the shortest path in a graph where the steps or edges have varying costs. Unlike Breadth-First Search (BFS) and Iterative Deepening Search (IDS), which are designed to find the path with the fewest steps, UCS is tailored to find the path with the lowest cumulative cost, making it ideal when the number of steps does not correlate directly with the path cost. For instance, if the goal is to travel from point A to point B through various intermediate points (A-S-F-B), UCS ensures that the sum of the distances (or costs) along the path is minimized, rather than merely minimizing the number of steps.

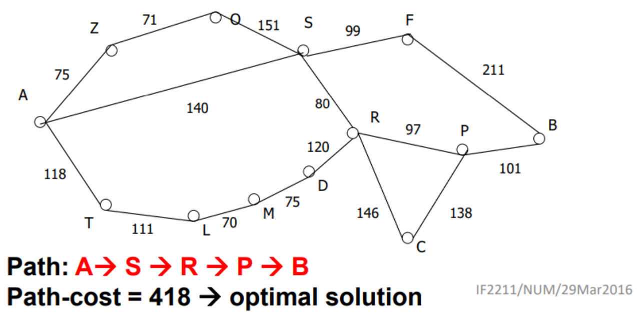


Figure 1. UCS Illustration in Pathfinding
(Source: [1])

To achieve this, UCS uses a priority queue where each node is prioritized based on the cumulative cost. By always expanding the node with the lowest g(n) value, UCS systematically explores paths in order of their cumulative cost, ensuring that the first time it reaches the goal node, it has found the shortest possible path in terms of total distance or cost. This makes UCS particularly relevant for problems where the cost of traveling between nodes varies, providing an optimal solution based on the sum of distances rather than the number of steps. In Uniform Cost Search (UCS), the function below is used to determine the priority of each node in the search process.

$$f(n) = g(n)$$

where g(n) is the cumulative cost from the start node to the current node n. In hyperparameter tuning scenario, this cost is effectively the negative of the model's accuracy score. The rationale behind using the negative accuracy score is to allow the priority queue to prioritize configurations that yield higher accuracies (since a higher accuracy would correspond to a lower negative value).

Although typically used for route finding, Uniform Cost Search (UCS) also works well for hyperparameter tuning in machine learning models. By prioritizing parameter configurations based on their performance (cost), UCS systematically explores the most promising settings first. This approach ensures that the search efficiently navigates the hyperparameter space, focusing on combinations that enhance model accuracy.

Uniform Cost Search (UCS) can operate in two modes: it can either stop after finding the first complete set of parameters with the lowest cost or continue searching to explore other potential solutions. In the first mode, UCS identifies and returns the optimal solution based on cumulative cost and halts further exploration, ensuring a quick resolution when the first good enough solution is acceptable. In the second mode, UCS continues to traverse the search space even after finding an initial solution, checking other nodes to ensure that no better solution exists. This exhaustive approach guarantees finding the globally optimal solution, though it may require more time. This dual capability allows UCS to balance between time efficiency and thoroughness, making it versatile for different optimization needs.

UCS does not typically continue searching after finding the optimal solution, as the algorithm inherently ensures that the first solution found is the optimal one due to its nature of expanding the least-cost node first. However, UCS can be modified to continue exploring the search space even after finding an initial solution to check for other potentially optimal solutions.

Although at first glance it might seem that modifying Uniform Cost Search (UCS) to continue searching after finding the first solution makes it similar to brute force, there are significant differences. UCS uses a priority queue to always expand the least-cost node first, ensuring that the search is more directed and efficient compared to the exhaustive enumeration of brute force. This prioritization allows UCS to systematically reduce the number of nodes it needs to explore, avoiding paths that are guaranteed to be more expensive than the current best path. While both methods will eventually consider all possible solutions, UCS retains its efficiency advantage by focusing on the most promising paths first. Thus, even with continued searching, UCS remains more efficient and targeted than a traditional brute force approach, highlighting its robustness in finding optimal solutions.

B. Machine Learning

Machine learning is a field of artificial intelligence that focuses on the development of algorithms and statistical models that enable computers to perform tasks without explicit instructions. Instead, these systems learn from data, identifying patterns and making decisions based on their learning. The theoretical basis of machine learning encompasses several core concepts, including:

1. Supervised Learning

   This concepts involves training a model on a labeled dataset, meaning the data includes both input features and the corresponding target labels. The model learns to predict the output from the input data. Common techniques include linear regression, logistic regression, support vector machines, and neural networks.

2. Unsupervised Learning

   This concepts involves training a model on data that does not have labeled responses. The goal is to find hidden patterns or intrinsic structures within the data. Techniques include clustering (e.g., k-means, hierarchical clustering) and dimensionality reduction (e.g., PCA, t-SNE).

3. Reinforcement Learning

   This concepts involves training an agent to make a sequence of decisions by rewarding desired behaviors. The agent learns to achieve a goal in an uncertain, potentially complex environment. Techniques include Q-learning and policy gradient methods.

4. Semi-Supervised Learning

   This concepts combines labeled and unlabeled data to improve learning accuracy. It leverages a small amount of labeled data to better understand the structure of the unlabeled data.

Data modeling in machine learning involves creating structured representations of data to develop predictive models that can provide valuable insights and make accurate predictions. The process begins with data preprocessing, where raw data is cleaned and transformed to make it suitable for modeling. This includes handling missing values, outliers, and inconsistencies, normalizing and scaling numerical features, and encoding categorical variables. The data is then split into training, validation, and test sets to ensure robust evaluation of the model's performance.

Feature selection and engineering are crucial steps that follow, where the most relevant features are identified and new features are created to enhance model performance. This may involve selecting variables that contribute significantly to the prediction task and creating interaction terms or polynomial features. Once the features are prepared, the next step is model selection and training. Various algorithms can be used depending on the nature of the problem and the data. Common models include linear regression, decision trees, random forests, support vector machines, and neural networks. The chosen model is trained on the training dataset by adjusting its parameters to minimize prediction errors.

C. Hyperparameter Tuning

Hyperparameters are external configurations to a model that must be set before training begins. Examples include the learning rate in gradient descent, the number of trees in a random forest, and the maximum depth of a decision tree. The choice of hyperparameters can affect the model's convergence speed, its ability to generalize, and its overall performance. Proper tuning of these hyperparameters is essential because they control the complexity and capacity of the model, and thereby directly influence its predictive power and efficiency.

Several traditional strategies exist for hyperparameter tuning, each with its own advantages and limitations:

1. Grid Search: This method exhaustively searches through a manually specified subset of the hyperparameter space. Although simple and effective, grid search can be computationally expensive and time-consuming, especially with large datasets and complex models.
2. Random Search: Instead of exhaustively searching all possible combinations, random search samples a fixed number of hyperparameter configurations from a defined search space. Research has shown that random search can be more efficient than grid search, often finding good hyperparameter configurations more quickly.
3. Bayesian Optimization: This method uses probabilistic models to predict the performance of hyperparameters and select the most promising candidates for evaluation. It balances exploration and exploitation to efficiently navigate the hyperparameter space.
4. Evolutionary Algorithms: These are population-based optimization algorithms inspired by natural selection. They iteratively evolve a population of candidate solutions using operations like mutation, crossover, and selection.
5. Gradient-Based Optimization: Some advanced techniques use gradient information to optimize hyperparameters. These methods require differentiable hyperparameters and can be very efficient for certain types of models.

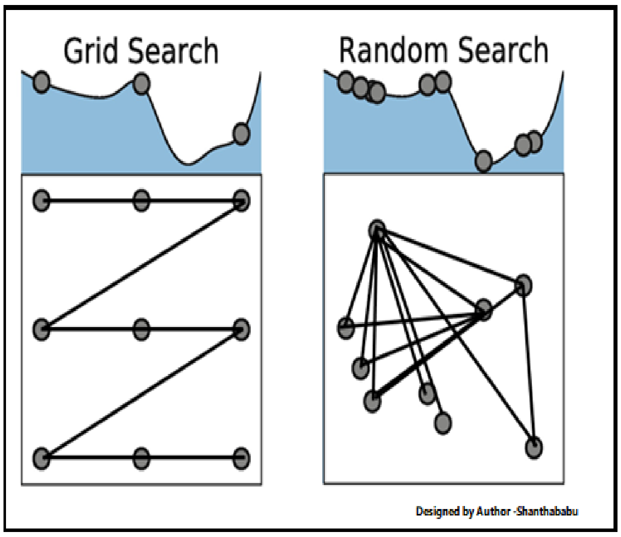


Figure 2. Grid search and Random search illustration (Source : [3])

The performance of different hyperparameter configurations is typically evaluated using cross-validation on a validation set. Common metrics include accuracy, precision, recall, F1 score, and mean squared error, depending on the type of problem (classification or regression). The chosen metric guides the search process, helping to identify the best hyperparameter set.

III. IMPLEMENTATION

In this section, we will outline the process of preparing the scenario for our testing. We will begin by discussing the steps involved in cleaning and engineering the dataset to ensure it is suitable for hyperparameter tuning. Following this, we will delve into the implementation of Uniform Cost Search (UCS) for finding optimal hyperparameter configurations. Finally, we will compare the performance of UCS with RandomizedSearchCV to evaluate its effectiveness in hyperparameter optimization.

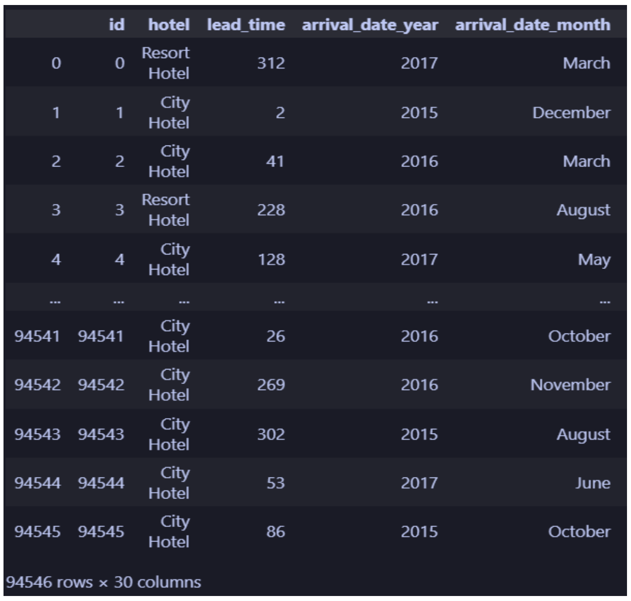
A. Limitations

In terms of limitations, there are several considerations to take into account regarding this research and its implementation. Firstly, accuracy might vary across test cases due to the use of the random shuffle splitting method, which can introduce variability in the results. Additionally, while the UCS method aims to find the set of parameters that yields the highest accuracy, tuning does not always guarantee better results than using untuned parameters, as overtuning can occur. This research focuses on demonstrating that among various tuning methods, UCS can identify the highest accuracy configuration.

Furthermore, only basic cleaning and preprocessing were applied to the data, which might have limited the achievable accuracy. Additional preprocessing steps and gaining deeper insights into the data could further enhance accuracy. Finally, while the UCS method accepts a range of variables and parameter values, the broader the range and the more variables included, the longer the search process will take. This research's implementation aims not only to find a complete parameter set but also to identify the optimal one, which can be time-consuming.

B. *Dataset*

The dataset used in this study contains detailed information on hotel bookings, comprising 94,546 rows and 30 columns. It provides a comprehensive overview of each booking, including the booking ID, average daily rate, number of adults, children, and babies, arrival details (day, month, week number, and year), room type assigned and reserved, booking changes, and customer type. Additionally, it includes data on waiting list duration, deposit type, distribution channel, country of origin, previous cancellations, parking space requirements, and the final reservation status. The purpose of this dataset is to provide information to determine whether a user has checked out or canceled their reservation.



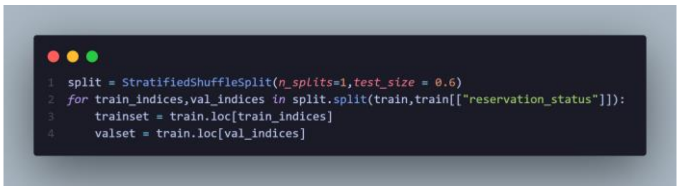
	id	hotel	lead_time	arrival_date_year	arrival_date_month
0	0	Resort Hotel	312	2017	March
1	1	City Hotel	2	2015	December
2	2	City Hotel	41	2016	March
3	3	Resort Hotel	228	2016	August
4	4	City Hotel	128	2017	May
...	...	...	...	...	...
94541	94541	City Hotel	26	2016	October
94542	94542	City Hotel	269	2016	November
94543	94543	City Hotel	302	2015	August
94544	94544	City Hotel	53	2017	June
94545	94545	City Hotel	86	2015	October

94546 rows × 30 columns

Figure 3. Snippet of Dataset
(Source : Code written by the author)

C. *Data Preparation*

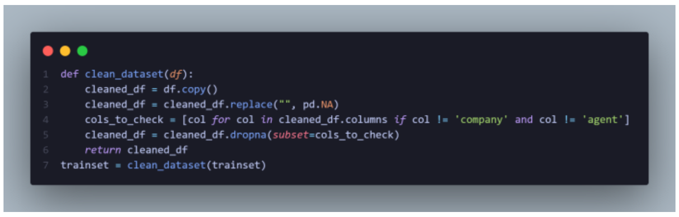
The first step of the preprocessing involves splitting the data into a training set and a validation set. The training set is used for model training, allowing the algorithm to learn from the data, while the validation set is reserved for evaluation, enabling the assessment of the model's performance on unseen data.



```
split = StratifiedShuffleSplit(n_splits=1,test_size = 0.6)
for train_indices,val_indices in split.split(train,train[["reservation_status"]]):
    trainset = train.loc[train_indices]
    valset = train.loc[val_indices]
```

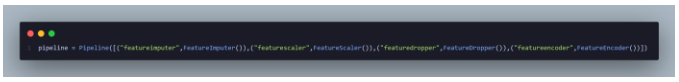
Figure 4. Snippet of data shuffle splitter
(Source : Code written by the author)

Before moving to the pipeline, the dataset is cleaned of null and empty values to ensure data quality and consistency. However, the 'agents' and 'country' columns are retained despite containing a significant number of NaN values, as removing them at this stage would substantially reduce the size of the training data. These columns will be handled later in the pipeline, where rows containing NaN values in 'agents' and 'country' will be imputed altogether to ensure the integrity of the final dataset.



```
def clean_dataset(df):
    cleaned_df = df.copy()
    cleaned_df = cleaned_df.replace("", pd.NA)
    cols_to_check = [col for col in cleaned_df.columns if col != 'company' and col != 'agent']
    cleaned_df = cleaned_df.dropna(subset=cols_to_check)
    return cleaned_df
trainset = clean_dataset(trainset)
```

Figure 5. Snippet of Data cleaning function
(Source : Code written by the author)



```
pipeline = Pipeline([("featureimputer",FeatureImputer()),("featurescaler",FeatureScaler()),("featuredropper",FeatureDropper()),("featureencoder",FeatureEncoder())])
```

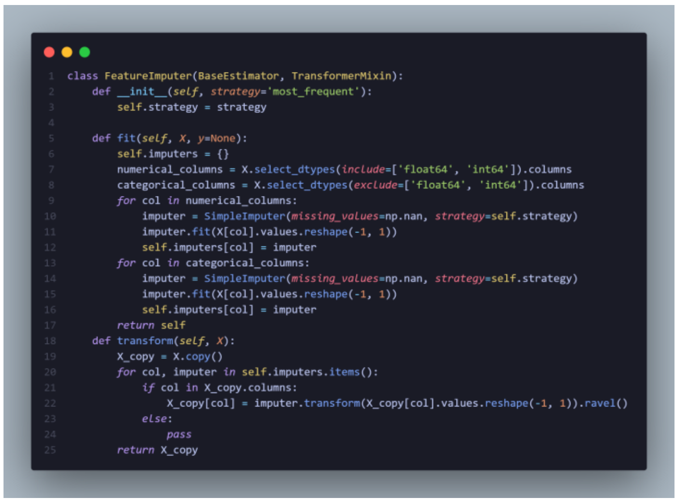
Figure 6. Snippet of data pipeline
(Source : Code written by the author**)**

No outlier cleaning was performed in this study due to the imbalanced and varied nature of the data, which could lead to excessive deletion when attempting to remove outliers. To maintain accuracy, it was decided to retain the outliers and address them by scaling later in the data processing pipeline. This approach ensures that valuable data points are not discarded, while still managing their impact on the model's performance through subsequent scaling techniques.

After clearing the trainset of the NaN values, a pipeline is constructed to process the data further. The pipeline includes several steps:

1. FeatureImputer

   This step handles missing values in the dataset. The fills in or replaces missing data points, ensuring that the dataset is complete and can be used for training without issues related to incomplete data.

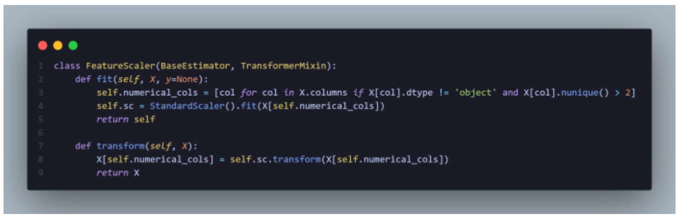


```python
class FeatureImputer(BaseEstimator, TransformerMixin):
    def __init__(self, strategy='most_frequent'):
        self.strategy = strategy

    def fit(self, X, y=None):
        self.imputers = {}
        numerical_columns = X.select_dtypes(include=['float64', 'int64']).columns
        categorical_columns = X.select_dtypes(exclude=['float64', 'int64']).columns
        for col in numerical_columns:
            imputer = SimpleImputer(missing_values=np.nan, strategy=self.strategy)
            imputer.fit(X[col].values.reshape(-1, 1))
            self.imputers[col] = imputer
        for col in categorical_columns:
            imputer = SimpleImputer(missing_values=np.nan, strategy=self.strategy)
            imputer.fit(X[col].values.reshape(-1, 1))
            self.imputers[col] = imputer
        return self
    def transform(self, X):
        X_copy = X.copy()
        for col, imputer in self.imputers.items():
            if col in X_copy.columns:
                X_copy[col] = imputer.transform(X_copy[col].values.reshape(-1, 1)).ravel()
            else:
                pass
        return X_copy
```

Figure 7. Snippet of FeatureImputer
(Source : Code written by the author)

2. FeatureScaler

   The steps standardizes the range of independent variables or features of data, which is crucial for many machine learning algorithms that are sensitive to the scale of data.

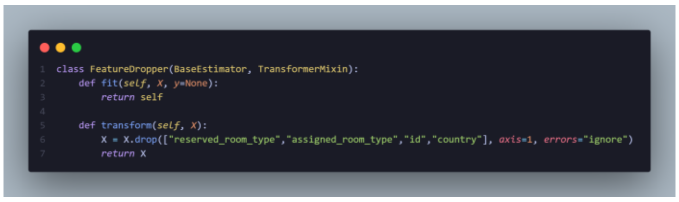


```python
class FeatureScaler(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        self.numerical_cols = [col for col in X.columns if X[col].dtype != 'object' and X[col].nunique() > 2]
        self.sc = StandardScaler().fit(X[self.numerical_cols])
        return self

    def transform(self, X):
        X[self.numerical_cols] = self.sc.transform(X[self.numerical_cols])
        return X
```

Figure 8. Snippet of FeatureScaler
(Source : Code written by the author)

3. FeatureDropper

   This step removes columns that are not useful for the model training process, such as columns with too many missing values or irrelevant information.

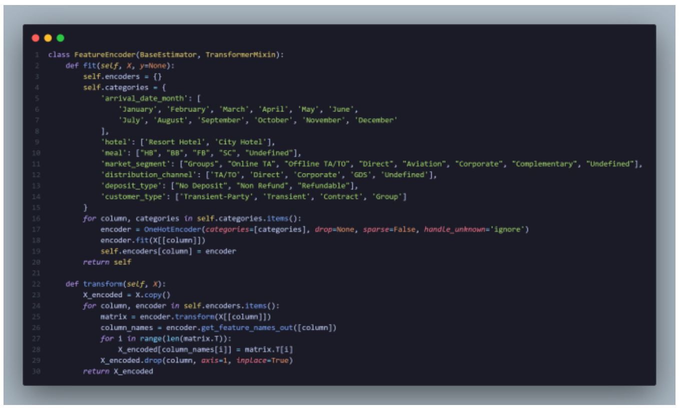


```python
class FeatureDropper(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X = X.drop(["reserved_room_type","assigned_room_type","id","country"], axis=1, errors="ignore")
        return X
```

Figure 9. Snippet of FeatureDropper
(Source : Code written by the author)

4. FeatureEncoder.

   This step encodes categorical variables. It transforms categorical features into a format that can be provided to the machine learning algorithm to do a better job in prediction.



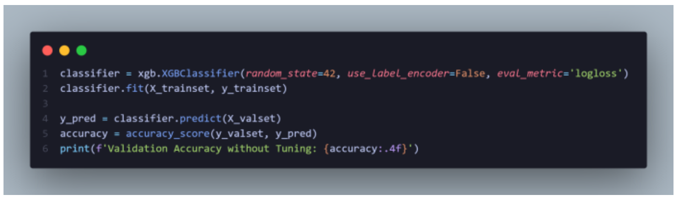
```python
class FeatureEncoder(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        self.encoders = {}
        self.categories = {
            'arrival_date_month': [
                'January', 'February', 'March', 'April', 'May', 'June',
                'July', 'August', 'September', 'October', 'November', 'December'
            ],
            'hotel': ['Resort Hotel', 'City Hotel'],
            'meal': ["HB", "BB", "FB", "SC", "Undefined"],
            'market_segment': ["Groups", "Online TA", "Offline TA/TO", "Direct", "Aviation", "Corporate", "Complementary", "Undefined"],
            'distribution_channel': ['TA/TO', 'Direct', 'Corporate', 'GDS', 'Undefined'],
            'deposit_type': ["No Deposit", "Non Refund", "Refundable"],
            'customer_type': ['Transient-Party', 'Transient', 'Contract', 'Group']
        }
        for column, categories in self.categories.items():
            encoder = OneHotEncoder(categories=[categories], drop=None, sparse=False, handle_unknown='ignore')
            encoder.fit(X[[column]])
            self.encoders[column] = encoder
        return self

    def transform(self, X):
        X_encoded = X.copy()
        for column, encoder in self.encoders.items():
            matrix = encoder.transform(X[[column]])
            column_names = encoder.get_feature_names_out([column])
            for i in range(len(matrix.T)):
                X_encoded[column_names[i]] = matrix.T[i]
            X_encoded.drop(column, axis=1, inplace=True)
        return X_encoded
```

Figure 10. Snippet of FeatureEncoder
(Source : Code written by the author)

D. *Data Modelling*

After preprocessing the data to ensure it is suitable for modeling, the modeling process is divided into four distinct scenarios, all utilizing the XGBoost model. This approach allows for a comprehensive evaluation of different hyperparameter tuning methods on model performance.

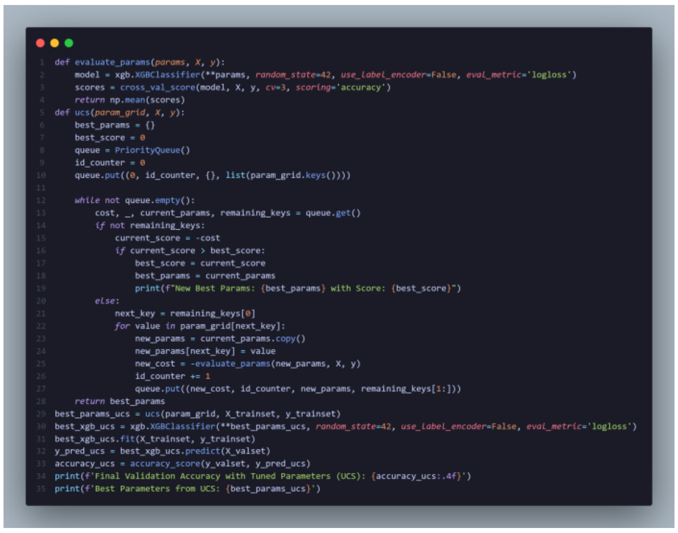
The first scenario involves using XGBoost with its default hyperparameters. This serves as a baseline model to understand the performance of XGBoost without any hyperparameter optimization. By evaluating the model in its default state, we can establish a reference point for comparison with the other tuned models. This baseline helps to highlight the impact of hyperparameter tuning on model performance.



```python
classifier = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
classifier.fit(X_trainset, y_trainset)

y_pred = classifier.predict(X_valset)
accuracy = accuracy_score(y_valset, y_pred)
print(f'Validation Accuracy without Tuning: {accuracy:.4f}')
```

Figure 11. Snippet of pure XGBoost prediction
(Source : Code written by the author)

In the second scenario, XGBoost is employed with hyperparameters optimized using Uniform Cost Search (UCS). UCS is a best-first search algorithm that systematically explores the hyperparameter space by prioritizing configurations based on their potential to improve model performance. By assigning costs inversely related to predicted accuracy, UCS effectively identifies promising hyperparameter settings. This scenario aims to demonstrate the efficiency and effectiveness of UCS in navigating the hyperparameter space to find configurations that enhance model accuracy.



```python
def evaluate_params(params, X, y):
    model = xgb.XGBClassifier(**params, random_state=42, use_label_encoder=False, eval_metric='logloss')
    scores = cross_val_score(model, X, y, cv=3, scoring='accuracy')
    return np.mean(scores)
def ucs(param_grid, X, y):
    best_params = {}
    best_score = 0
    queue = PriorityQueue()
    id_counter = 0
    queue.put((0, id_counter, {}, list(param_grid.keys())))

    while not queue.empty():
        cost, _, current_params, remaining_keys = queue.get()
        if not remaining_keys:
            current_score = -cost
            if current_score > best_score:
                best_score = current_score
                best_params = current_params
                print(f"New Best Params: {best_params} with Score: {best_score}")
        else:
            next_key = remaining_keys[0]
            for value in param_grid[next_key]:
                new_params = current_params.copy()
                new_params[next_key] = value
                new_cost = -evaluate_params(new_params, X, y)
                id_counter += 1
                queue.put((new_cost, id_counter, new_params, remaining_keys[1:]))
    return best_params
best_params_ucs = ucs(param_grid, X_trainset, y_trainset)
best_xgb_ucs = xgb.XGBClassifier(**best_params_ucs, random_state=42, use_label_encoder=False, eval_metric='logloss')
best_xgb_ucs.fit(X_trainset, y_trainset)
y_pred_ucs = best_xgb_ucs.predict(X_valset)
accuracy_ucs = accuracy_score(y_valset, y_pred_ucs)
print(f'Final Validation Accuracy with Tuned Parameters (UCS): {accuracy_ucs:.4f}')
print(f'Best Parameters from UCS: {best_params_ucs}')
```

Figure 12. Snippet of UCS generated parameters and its XGBoost prediction
(Source : Code written by the author)

The algorithm begins by initializing an empty set for the best parameters (best_params) and setting the best score (best_score) to zero. It also initializes a priority queue to manage the exploration of parameter sets, where each item in the queue is a tuple containing the cost, a unique identifier, the current set of parameters, and the list of remaining parameter keys to explore.

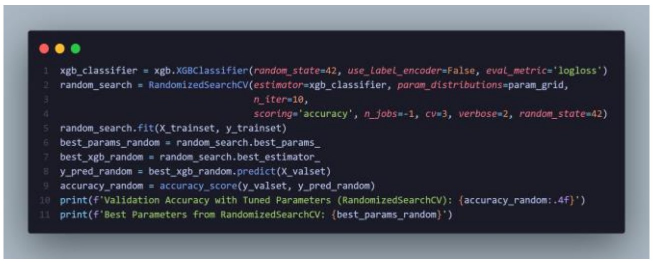
The algorithm enters a loop that continues until the queue is empty. In each iteration, it dequeues the parameter set with the lowest cost (highest accuracy). If there are no more parameters to explore (remaining_keys is empty), it compares the current score (negative of the cost) with the best score found so far. If the current score is higher, it updates the best parameters and best score.

If there are still parameters to explore, the algorithm takes the next parameter key from the list (next_key) and iterates over all possible values for this key. For each value, it creates a new parameter set (new_params) by copying the current parameters and adding the new value for the key. It then evaluates this new parameter set using the evaluate_params function, which performs cross-validation to obtain an accuracy score.

The new parameter set, along with its cost (negative accuracy score), unique identifier, and remaining keys to explore, is then added to the priority queue. The process continues until all possible parameter sets have been explored, and the queue is empty. The best set of parameters found during the search is returned.

The UCS method initially finds the first complete set of parameters with the lowest cost and prints it out. However, it does not stop there; another modification ensures that it continues its exhaustive search while still applying the UCS concept, starting from other nodes. Every time a new node yields better results, it prints a statement indicating a change from the initial node path. This approach ensures that UCS applies both concepts: stopping at the first found parameters if time efficiency is preferred, or continuing to exhaustively search for optimal parameters if better performance is desired. This makes comparison with RandomizedSearchCV and GridSearchCV viable, as UCS can be evaluated for both time efficiency and optimality.

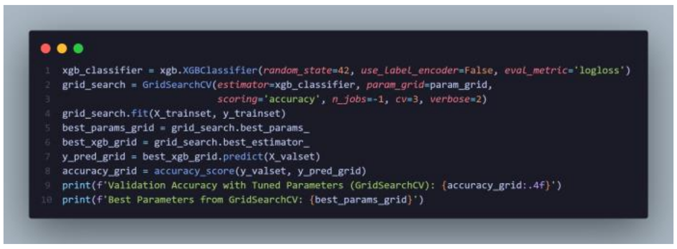
The third scenario applies XGBoost with hyperparameters determined through RandomizedSearchCV (a sklearn model selection library). RandomizedSearchCV randomly samples a specified number of hyperparameter combinations from the defined parameter space and evaluates them. This approach can often find good hyperparameters more quickly than exhaustive grid search, making it a popular choice for hyperparameter tuning.



```python
xgb_classifier = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
random_search = RandomizedSearchCV(estimator=xgb_classifier, param_distributions=param_grid,
                                   n_iter=10,
                                   scoring='accuracy', n_jobs=-1, cv=3, verbose=2, random_state=42)
random_search.fit(X_trainset, y_trainset)
best_params_random = random_search.best_params_
best_xgb_random = random_search.best_estimator_
y_pred_random = best_xgb_random.predict(X_valset)
accuracy_random = accuracy_score(y_valset, y_pred_random)
print(f'Validation Accuracy with Tuned Parameters (RandomizedSearchCV): {accuracy_random:.4f}')
print(f'Best Parameters from RandomizedSearchCV: {best_params_random}')
```

Figure 13. Snippet of XGBoost prediction using RandomizedSearchCV parameters
(Source : Code written by the author)

The fourth scenario applies XGBoost with hyperparameters determined through GridSearchCV, a model selection library from scikit-learn. GridSearchCV performs an exhaustive search over a specified parameter grid, evaluating all possible combinations of the provided hyperparameters. This approach systematically explores the entire parameter space to identify the best combination of hyperparameters that maximizes model performance.



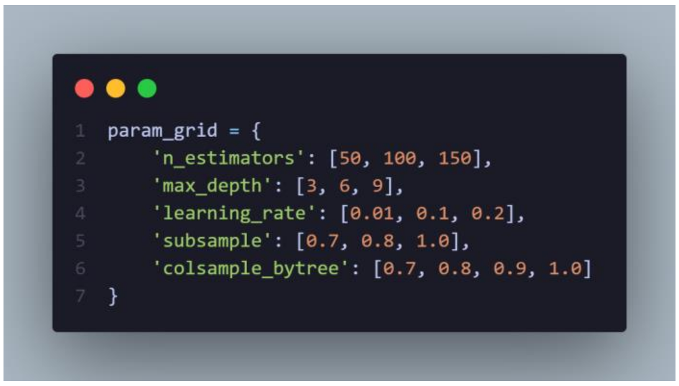
```python
xgb_classifier = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
grid_search = GridSearchCV(estimator=xgb_classifier, param_grid=param_grid,
                           scoring='accuracy', n_jobs=-1, cv=3, verbose=2)
grid_search.fit(X_trainset, y_trainset)
best_params_grid = grid_search.best_params_
best_xgb_grid = grid_search.best_estimator_
y_pred_grid = best_xgb_grid.predict(X_valset)
accuracy_grid = accuracy_score(y_valset, y_pred_grid)
print(f'Validation Accuracy with Tuned Parameters (GridSearchCV): {accuracy_grid:.4f}')
print(f'Best Parameters from GridSearchCV: {best_params_grid}')
```

Figure 14. Snippet of XGBoost prediction using GridSearchCV parameters
(Source : Code written by the author)

The purpose of the third and fourth scenarios is to evaluate whether UCS is effective for hyperparameter tuning. The third scenario uses RandomizedSearchCV, which prioritizes time efficiency by sampling a specified number of hyperparameter combinations. Therefore, UCS, with its longer search time, should at least achieve the same or better results as RandomizedSearchCV. The fourth scenario employs GridSearchCV, which performs an exhaustive search to find the optimal parameters. Hence, UCS should yield results that are at least as good as, or very close to, those obtained by GridSearchCV. Meeting these benchmarks would demonstrate that UCS is a viable and effective method for hyperparameter tuning.

IV. TESTING AND ANALYSIS

The model-ready dataset was then passed through three different scenarios. In this test case, the parameters used were based on the features of XGBoost, which included the following parameter grid: 'n_estimators' with values [50, 100, 150], 'max_depth' with values [3, 6, 9], 'learning_rate' with values [0.01, 0.1, 0.2], 'subsample' with values [0.7, 0.8, 1.0], and 'colsample_bytree' with values [0.7, 0.8, 0.9, 1.0].



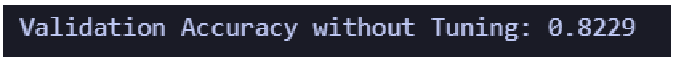
```
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [3, 6, 9],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.7, 0.8, 1.0],
    'colsample_bytree': [0.7, 0.8, 0.9, 1.0]
}
```

Figure 15. Snippet of parameters used as test cases
(Source : Code written by the author)

The choice of model may vary according to its intended usage, as different models have distinct parameters and requirements. It's important to remember that each model type comes with its own set of parameters that need to be optimized to achieve the best performance. In this test case, XGBoost is used as the model due to its ability to handle large datasets efficiently, its robustness in managing both regression and classification tasks, and its superior performance through gradient boosting techniques. XGBoost's regularization capabilities also help prevent overfitting, making it an ideal choice for achieving high accuracy and reliability in predictions.

The results of the three scenarios are then compared and evaluated together to gain additional insights and determine whether the search for the optimal parameters is returns a desirable results.

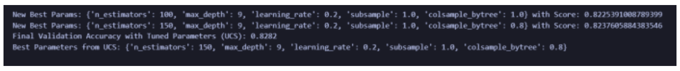
In this test case, with a dataset split of 60% for training and 40% for testing, and using the XGBoost model without hyperparameter tuning, the resulting accuracy is 0.8229.



```
Validation Accuracy without Tuning: 0.8229
```

Figure 16. Snippet of accuracy result using pure XGBoost
(Source : Code written by the author)

This shows a satisfactory result, as accuracy ranges from 70% to 90% are considered optimal when evaluating a model. However, another attempt will be made to further increase the accuracy by applying different sets of parameters.



```
New Best Params: {'n_estimators': 100, 'max_depth': 9, 'learning_rate': 0.2, 'subsample': 1.0, 'colsample_bytree': 1.0} with Score: 0.8225391008789399
New Best Params: {'n_estimators': 150, 'max_depth': 9, 'learning_rate': 0.2, 'subsample': 1.0, 'colsample_bytree': 0.8} with Score: 0.8237605884383546
Final Validation Accuracy with Tuned Parameters (UCS): 0.8282
Best Parameters from UCS: {'n_estimators': 150, 'max_depth': 9, 'learning_rate': 0.2, 'subsample': 1.0, 'colsample_bytree': 0.8}
```

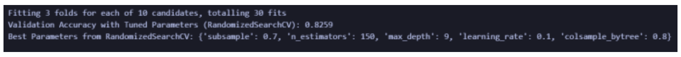
Figure 17. Snippet of accuracy result using XGBoost and UCS generated parameters
(Source : Code written by the author)

When evaluating the model using UCS, the search process took approximately 14 minutes. Despite the longer search time, it resulted in a better accuracy score of 0.8282. During the search, the model evaluated one parameter sets before identifying the final set as the best one. The evaluation was performed using cross-validation scores, and once the optimal parameters were found, they were used to predict accuracy on the test set.

As shown above, the print statement is triggered whenever a better score is found, indicating that the UCS method has identified a new node that yields improved results. This demonstrates the effectiveness of UCS in continually refining the search for the optimal parameters by prioritizing nodes with higher potential.

Had the problem been solely to find the first solution while maintaining its optimality, the UCS code could be adjusted to stop after returning the first complete set of parameters and it will have a much shorter search time while still maintaining its high accuracy (Figure 17 shows it stands slightly below the best value). This initial set would be considered a good solution within the scope of the explored branches, even if it does not encompass the entire parameter set. This adjustment would allow for faster results while still ensuring the parameters are effective, though they might not be the absolute best across the whole parameter space. However, to be able to compete with traditional strategies, it is necessary to enhance its optimality to the fullest, even if it means increasing the search time.

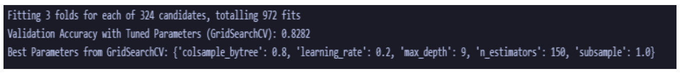
Compared to the previous accuracy, we can conclude that UCS indeed returns parameters that boost the accuracy of the model. To test whether this improvement represents the upper bound (the best result within the params range provided) , we will now compare it with the results from Randomized Cross-Validation (RandomizedCV). RandomizedCV is known to yield diverse results due to its stochastic nature, potentially identifying different optimal parameters that could further enhance accuracy.



```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
Validation Accuracy with Tuned Parameters (RandomizedSearchCV): 0.8259
Best Parameters from RandomizedSearchCV: {'subsample': 0.7, 'n_estimators': 150, 'max_depth': 9, 'learning_rate': 0.1, 'colsample_bytree': 0.8}
```

Figure 18. Snippet of accuracy result using XGBoost and RandomizedCV parameters
(Source : Code written by the author)

The RandomizedCV returned an accuracy value of 0.8259, which is higher than the untuned model but lower than the accuracy achieved by the UCS method. It is important to note that RandomizedCV focuses on balancing between time and results, indicating that our UCS method has proven to be effective in determining the optimal parameters for achieving the best results.



```
Fitting 3 folds for each of 324 candidates, totalling 972 fits
Validation Accuracy with Tuned Parameters (GridSearchCV): 0.8282
Best Parameters from GridSearchCV: {'colsample_bytree': 0.8, 'learning_rate': 0.2, 'max_depth': 9, 'n_estimators': 150, 'subsample': 1.0}
```

Figure 19. Snippet of accuracy result using XGBoost and GridSearchCV parameters
(Source : Code written by the author)

The results above show that the accuracy and preferred parameters obtained using GridSearchCV are equal to those achieved with our UCS method. Since GridSearchCV employs an exhaustive approach to find the best solution, this parity in results demonstrates that our UCS method works effectively. It is important to note, however, that GridSearchCV might be faster due to its implementation as a library, which benefits from optimization techniques such as parallel processing. These built-in optimizations enable GridSearchCV to efficiently handle exhaustive searches, further validating the robustness and efficiency of our UCS approach in comparison.

The UCS method might provide better results than RandomizedCV because it focuses more on finding the optimal result rather than minimizing the time required. However, it still falls short compared to GridSearchCV, which, despite being an exhaustive search method, has its own optimization techniques. The purpose of demonstrating that the UCS method works is evident in how the program consistently selects nodes with higher priority, ensuring a thorough search for the best parameters. This approach has paid off, as shown by UCS outperforming RandomizedCV in terms of accuracy and have matching parameters and accuracy with GridSearchCV.

V. CONCLUSION

A key way to increase model accuracy is through hyperparameter tuning. Traditional methods of hyperparameter tuning include RandomizedCV and GridSearchCV, among others. However, in this research, the author challenged himself to apply the UCS method to find the best and most optimal parameters. This approach aims to test if the UCS method could be used as an effective way to provide optimal results, leveraging its systematic and prioritized search capabilities as an alternative to conventional hyperparameter tuning techniques.

Uniform Cost Search (UCS) is conceptually advantageous due to its flexibility. It can maintain its true UCS nature by stopping at the first iteration once an optimal solution is found, ensuring efficiency while still achieving good results. Alternatively, it can compete with traditional exhaustive methods by continuing the search to explore additional solutions, all while maintaining a cost-minimization approach. This dual capability allows UCS to either prioritize speed or thoroughness, making it a robust and adaptable method for finding the best parameters.

Based on testing and analysis, the UCS method has proven its effectiveness in finding the best parameters for hyperparameter tuning. This is evidenced by its ability to yield better results while maintaining its behavior of printing every time it finds nodes with better accuracy, ultimately achieving a higher score than RandomizedCV. Although UCS may seem inefficient compared to built-in traditional selection libraries like GridSearchCV, this paper demonstrates the fundamental concept of applying UCS as a method for identifying optimal parameters. With further optimization, UCS could potentially yield excellent results in a more efficient timeframe.

VI. APPENDIX

https://github.com/Razark-Y/UCS_Approach_For_Hyperparameter_Tuning

VIDEO LINK AT YOUTUBE

https://www.youtube.com/watch?v=DzqWGiAI670

ACKNOWLEDGMENT

The author wishes to extend heartfelt gratitude to the following individuals:

1. **The Author's Parents**: The constant support, encouragement, and belief in the author's abilities from their parents have been invaluable. Their unwavering presence and motivation have driven the completion of this work.

2. **Dr. Ir. Rinaldi Munir, M.T., Dr. Ir. Rila Mandala, and Dr. Nur Ulfa Maulidevi**: The author expresses deep appreciation for these esteemed professors of the IF2211 Algorithm Strategies course for their guidance and knowledge imparted during the lectures

3. **Friends and peers:** for their support and contributions in the drafting and refining of this paper.

Finally, the author extends thanks to the readers. Apologies are offered for any errors in writing or content. The author sincerely hopes that this paper will be beneficial to its readers.

REFERENCES

[1] R. Munir, "Penentuan rute (Route/Path Planning) - Bagian 1," [Online]. Available:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf [Accessed 12 July 2024]

[2] G.Aurelien, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" [Online].
Available:
https://powerunit-ju.com/wp-content/uploads/2021/04/Aurelien-Geron-Hands-On-Machine-Learning-with-Scikit-Learn-Keras-and-Tensorflow_-Concepts-Tools-and-Techniques-to-Build-Intelligent-Systems-OReilly-Media-2019.pdf [Accessed 11 July 2024]

[3] S.Pandian, "A Comprehensive Guide on Hyperparameter Tuning and its Techniques" [Online]
Available:
https://www.analyticsvidhya.com/blog/2022/02/a-comprehensive-guide-on-hyperparameter-tuning-and-its-techniques/ [Accessed 12 July 2024]

[4] S.Sryheni, "Obtaining the Path in the Uniform Cost Search Algorithm" [Online]
Available:
https://www.baeldung.com/cs/find-path-uniform-cost-search [Accessed 12 July 2024]

[5] B.Priya," Hyperparameter Tuning: GridSearchCV and RandomizedSearchCV, Explained" [Online]
Available:

https://www.kdnuggets.com/hyperparameter-tuning-gridsearchcv-and-randomizedsearchcv-explained [Accessed 12 July 2024]

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, 12 Juni 2024

[illegible]

Wilson Yusda (13522019)