NAME :- NAMAN SAINI

ENROLLMENT NO.- 21112074

Automated hyperparameter optimisation:

➢ Introduction

1. Fine-tuning machine learning models is significantly enhanced by hyperparameter optimization.
2. Hyperparameters are adjustable settings that control the model's learning from data.
3. These settings are fixed before training starts, unlike model parameters which are learned during training.
4. Skilful hyperparameter tuning can greatly boost a model's performance.
5. The Bayesian Optimization method for hyperparameter refinement is the focus of this document.
6. Additionally, the Tree-structured Parzen Estimator (TPE) method has also been utilized for hyperparameter optimization.
7. A comparison has been made between Hyper opt, and Bayesian optimization techniques, including an analysis of their learning rates.

➢ Hyperparameters

1. Hyperparameters are configuration settings for a machine learning algorithm.
2. They are set before the training process begins.
3. Hyperparameters guide the training algorithm.
4. They significantly impact the model's performance.
5. Examples include learning rate, number of trees in a random forest, and number of layers in a neural network.

I have used Random Forest Classifier as my base model over which I have applied many different HPO's such as Bayesian Optimization, (TPE) Tree-Parzen Estimator and lastly Hyper-opt to compare above techniques with in built libraries

➢ Why Random Forest Classifier is used as base model

- **Supervised Learning Algorithm**: The Random Forest, also known as a Random Decision Forest, is a supervised machine learning algorithm that leverages multiple decision trees for tasks like classification and regression.
- **Versatile and Scalable**: It is particularly effective for handling large and complex datasets, making it suitable for high-dimensional feature spaces.
- **Feature Importance Insights**: This algorithm provides valuable insights into the significance of different features in the dataset.
- **High Predictive Accuracy**: Random Forests are renowned for their ability to deliver high predictive accuracy while minimizing the risk of overfitting.
- **Broad Applicability**: Its robustness and reliability make it a popular choice in various domains, including finance, healthcare, and image analysis.

➢ **Key Hyperparameters for Optimization in Random Forest Classifier**:

- **n_estimators**:
    - Controls the number of decision trees in the forest.
    - A higher number of trees generally improves model accuracy but increases computational complexity.
    - Finding the optimal number of trees is crucial for balancing performance and training time.
- **max_depth**:
    - Sets the maximum depth for each tree in the forest.
    - Crucial for enhancing model accuracy; deeper trees capture more complexity.
    - However, excessively deep trees can lead to overfitting, so setting an appropriate depth is vital to maintain generalization.
- **max_features**:
    - Determines the number of features to consider when looking for the best split at each node.
    - Balancing this helps the model to avoid overfitting and improves its performance by ensuring diverse decision boundaries.
- **criterion**:
    - Defines the function used to measure the quality of a split.
    - The choices are **"gini"** for Gini impurity and **"entropy"** for information gain.
    - Selecting the appropriate criterion can influence the effectiveness of the splits and overall model performance.

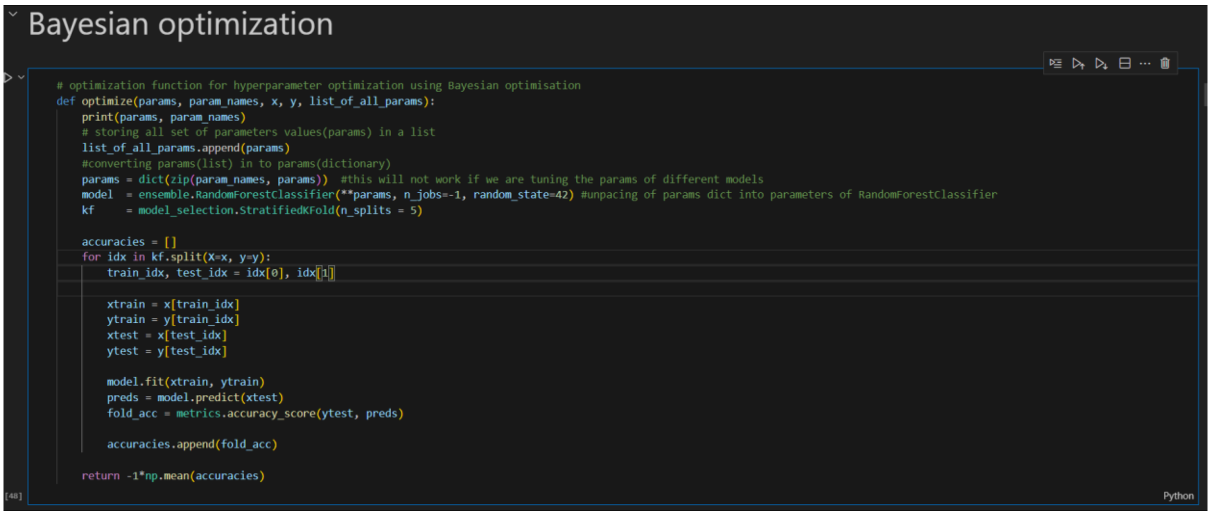
➢ **Bayesian Optimization**:

- **Purpose**:
    - An iterative method to minimize or maximize an objective function, especially useful when evaluations are expensive.

- **Initialization**:
    - Start with a small, randomly selected set of hyperparameter values.
    - Evaluate the objective function at these initial points to establish a starting dataset.
- **Surrogate Model**:
    - Construct a probabilistic model, typically a Gaussian Process, based on the initial evaluations.
    - This model serves as an approximation of the objective function, providing estimates and uncertainty measures.
- **Acquisition Function**:
    - Use the surrogate model to decide the next set of hyperparameters.
    - Optimize an acquisition function to balance exploring new areas and exploiting known promising regions.
- **Evaluation**:
    - Assess the objective function with the hyperparameters chosen by the acquisition function.
    - This involves running the model and recording the performance metrics for these hyperparameters.
- **Update**:
    - Integrate the new evaluation data into the surrogate model.
    - Refine the model's approximation of the objective function with the updated information.
- **Iteration**:
    - Repeat the steps of modelling, acquisition, and evaluation iteratively.
    - Continue the process until a stopping criterion, like a set number of iterations or a target performance level, is reached.

➢ Implementation

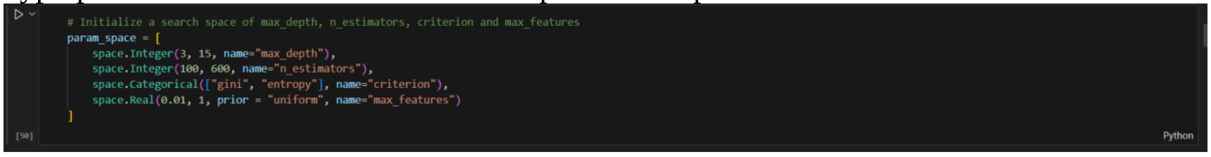
- **Step 1: Define the Objective Function**:
    - Our goal for optimization is to minimize the negative mean accuracy of a **`Random Forest Classifier`**.
    - This means our objective function will measure and return the negative of the mean accuracy to align with the minimization process. Below is a code snippet illustrating the objective function



Bayesian optimization

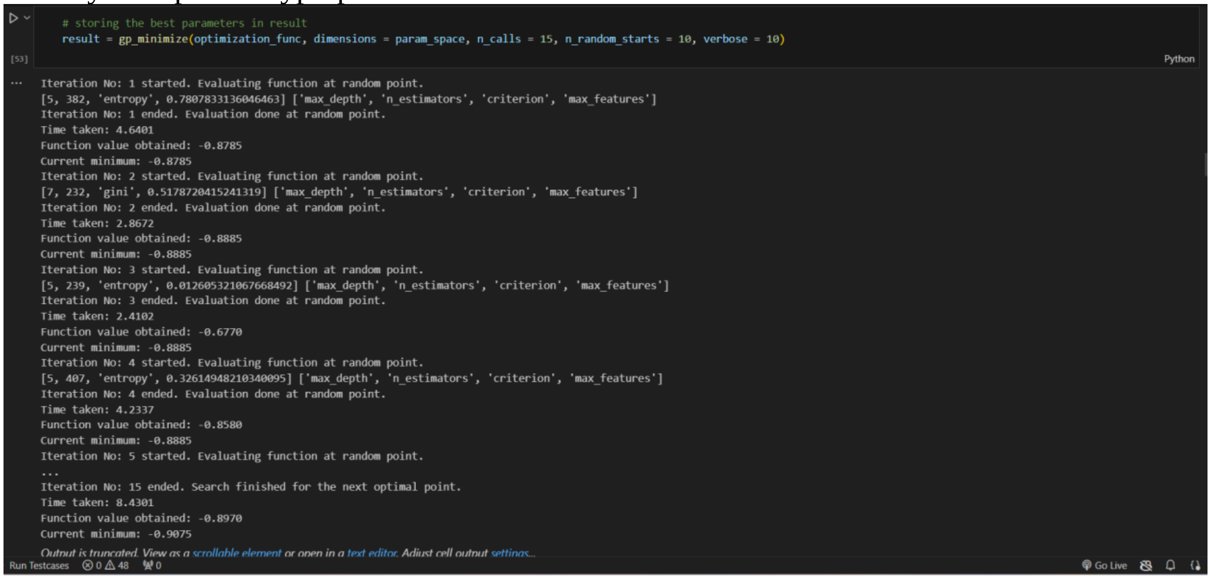
```python
# optimization function for hyperparameter optimization using Bayesian optimisation
def optimize(params, param_names, x, y, list_of_all_params):
    print(params, param_names)
    # storing all set of parameters values(params) in a list
    list_of_all_params.append(params)
    #converting params(list) in to params(dictionary)
    params = dict(zip(param_names, params))  #this will not work if we are tuning the params of different models
    model  = ensemble.RandomForestClassifier(**params, n_jobs=-1, random_state=42) #unpacing of params dict into parameters of RandomForestClassifier
    kf     = model_selection.StratifiedKFold(n_splits = 5)

    accuracies = []
    for idx in kf.split(X=x, y=y):
        train_idx, test_idx = idx[0], idx[1]

        xtrain = x[train_idx]
        ytrain = y[train_idx]
        xtest = x[test_idx]
        ytest = y[test_idx]

        model.fit(xtrain, ytrain)
        preds = model.predict(xtest)
        fold_acc = metrics.accuracy_score(ytest, preds)

        accuracies.append(fold_acc)

    return -1*np.mean(accuracies)
```

Step 2: Define the Hyperparameter Space:

- We need to outline the range and possible values for the hyperparameters we want to optimize.
- The following code snippet demonstrates the search space for various hyperparameters that will be used in the optimization process.
- 

```python
# Initialize a search space of max_depth, n_estimators, criterion and max_features
param_space = [
    space.Integer(3, 15, name="max_depth"),
    space.Integer(100, 600, name="n_estimators"),
    space.Categorical(["gini", "entropy"], name="criterion"),
    space.Real(0.01, 1, prior = "uniform", name="max_features")
]
```

Step 3: Execute the Optimization Algorithm:

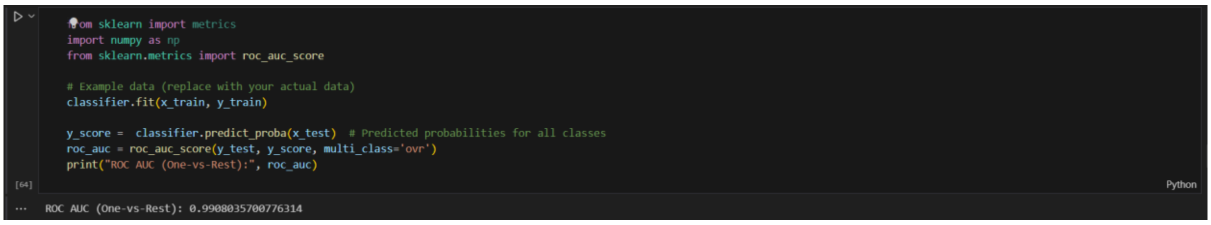
- Use the optimization algorithm to search for the best possible hyperparameters within the defined search space.
- The following code snippet illustrates how to run the optimization algorithm to identify the optimal hyperparameters.
- 

```python
# storing the best parameters in result
result = gp_minimize(optimization_func, dimensions = param_space, n_calls = 15, n_random_starts = 10, verbose = 10)
```

```
Iteration No: 1 started. Evaluating function at random point.
[5, 382, 'entropy', 0.7807833136046463] ['max_depth', 'n_estimators', 'criterion', 'max_features']
Iteration No: 1 ended. Evaluation done at random point.
Time taken: 4.6401
Function value obtained: -0.8785
Current minimum: -0.8785
Iteration No: 2 started. Evaluating function at random point.
[7, 232, 'gini', 0.5178720415241319] ['max_depth', 'n_estimators', 'criterion', 'max_features']
Iteration No: 2 ended. Evaluation done at random point.
Time taken: 2.8672
Function value obtained: -0.8885
Current minimum: -0.8885
Iteration No: 3 started. Evaluating function at random point.
[5, 239, 'entropy', 0.012605321067668492] ['max_depth', 'n_estimators', 'criterion', 'max_features']
Iteration No: 3 ended. Evaluation done at random point.
Time taken: 2.4102
Function value obtained: -0.6770
Current minimum: -0.8885
Iteration No: 4 started. Evaluating function at random point.
[5, 407, 'entropy', 0.32614948210340095] ['max_depth', 'n_estimators', 'criterion', 'max_features']
Iteration No: 4 ended. Evaluation done at random point.
Time taken: 4.2337
Function value obtained: -0.8580
Current minimum: -0.8885
Iteration No: 5 started. Evaluating function at random point.
...
Iteration No: 15 ended. Search finished for the next optimal point.
Time taken: 8.4301
Function value obtained: -0.8970
Current minimum: -0.9075
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Step 4: Evaluate the Results:

- Once optimization is complete, assess the performance of the best-found model.
- This involves calculating metrics like ROC-AUC scores and conducting cross-validation to ensure robust evaluation.

- 

```
from sklearn import metrics
import numpy as np
from sklearn.metrics import roc_auc_score

# Example data (replace with your actual data)
classifier.fit(x_train, y_train)

y_score =  classifier.predict_proba(x_test)  # Predicted probabilities for all classes
roc_auc = roc_auc_score(y_test, y_score, multi_class='ovr')
print("ROC AUC (One-vs-Rest):", roc_auc)
```

```
ROC AUC (One-vs-Rest): 0.9908035700776314
```

➢ Tree-structured Parzen Estimator (TPE) Optimization:

• Purpose:

- TPE optimizes an objective function iteratively, aiming to maximize or minimize it efficiently, especially beneficial when function evaluations are costly.

• Initialization:

- Initialize empty lists `params` and `results` to store sampled hyperparameters and their corresponding objective function scores.

• Iterations:

- For `n_calls` iterations:
  - Sample hyperparameters (`next_params`) from the defined `space` using random choice.
  - Evaluate the objective function (`objective_function`) with `next_params` to obtain a score (`score`).
  - Store `next_params` and `score` in `params` and `results`, respectively.

• Best Hyperparameters:

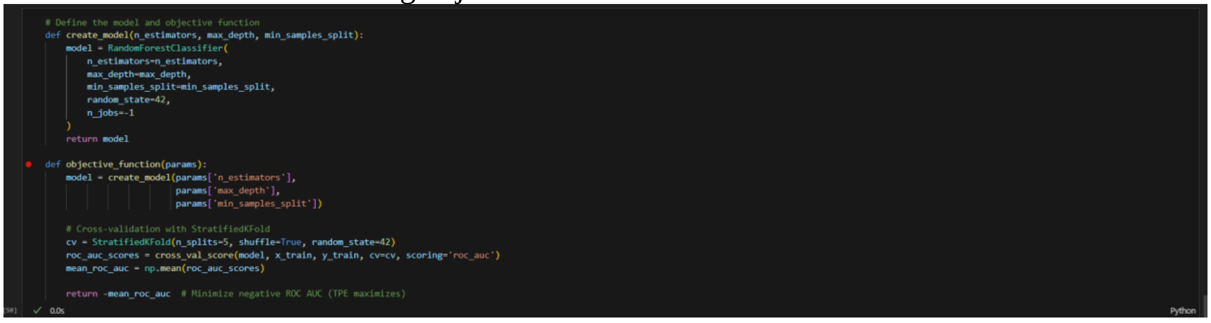
- Identify the index (`best_index`) of the highest score (`np.argmax(results)`), indicating the best-performing hyperparameters.
- Retrieve and return the best hyperparameters (`best_params`) based on `best_index`.

• Output:

- Print and return the best hyperparameters (`best_params`) found by the optimization process.

Below code snippet can be used to get the best parameters values:-

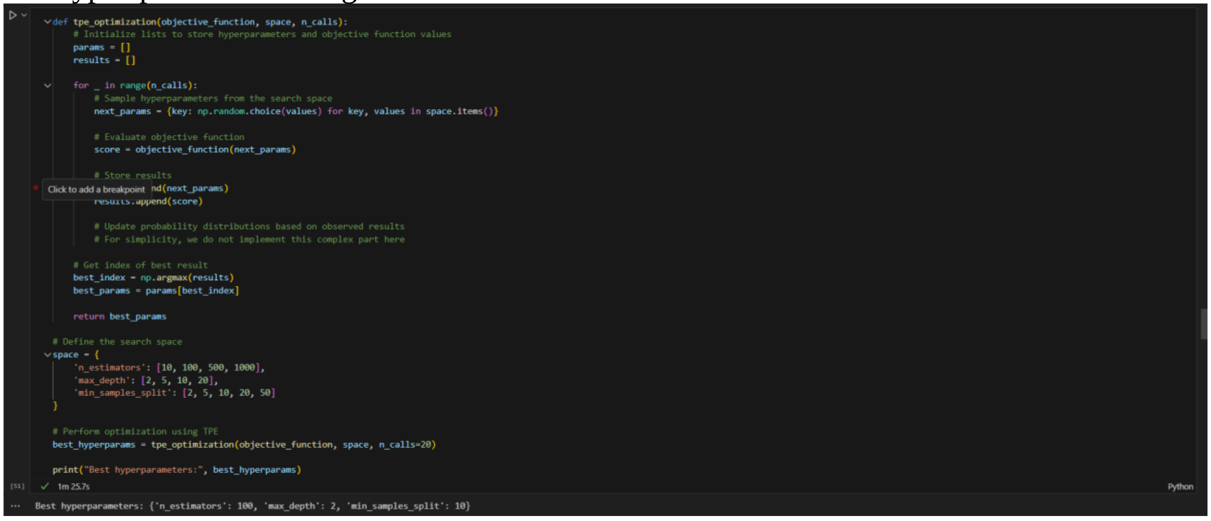
➢ For model creation and defining objective function



```
# Define the model and objective function
def create_model(n_estimators, max_depth, min_samples_split):
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        random_state=42,
        n_jobs=-1
    )
    return model

def objective_function(params):
    model = create_model(params['n_estimators'],
                         params['max_depth'],
                         params['min_samples_split'])

    # Cross-validation with StratifiedKFold
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    roc_auc_scores = cross_val_score(model, x_train, y_train, cv=cv, scoring='roc_auc')
    mean_roc_auc = np.mean(roc_auc_scores)

    return -mean_roc_auc  # Minimize negative ROC AUC (TPE maximizes)
```

[50] ✓ 0.0s

➢ For hyper-parameter tuning



```
def tpe_optimization(objective_function, space, n_calls):
    # Initialize lists to store hyperparameters and objective function values
    params = []
    results = []

    for _ in range(n_calls):
        # Sample hyperparameters from the search space
        next_params = {key: np.random.choice(values) for key, values in space.items()}

        # Evaluate objective function
        score = objective_function(next_params)

        # Store results
  Click to add a breakpoint nd(next_params)
        results.append(score)

        # Update probability distributions based on observed results
        # For simplicity, we do not implement this complex part here

    # Get index of best result
    best_index = np.argmax(results)
    best_params = params[best_index]

    return best_params

# Define the search space
space = {
    'n_estimators': [10, 100, 500, 1000],
    'max_depth': [2, 5, 10, 20],
    'min_samples_split': [2, 5, 10, 20, 50]
}

# Perform optimization using TPE
best_hyperparams = tpe_optimization(objective_function, space, n_calls=20)

print("Best hyperparameters:", best_hyperparams)
```

[51] ✓ 1m 25.7s

```
Best hyperparameters: {'n_estimators': 100, 'max_depth': 2, 'min_samples_split': 10}
```

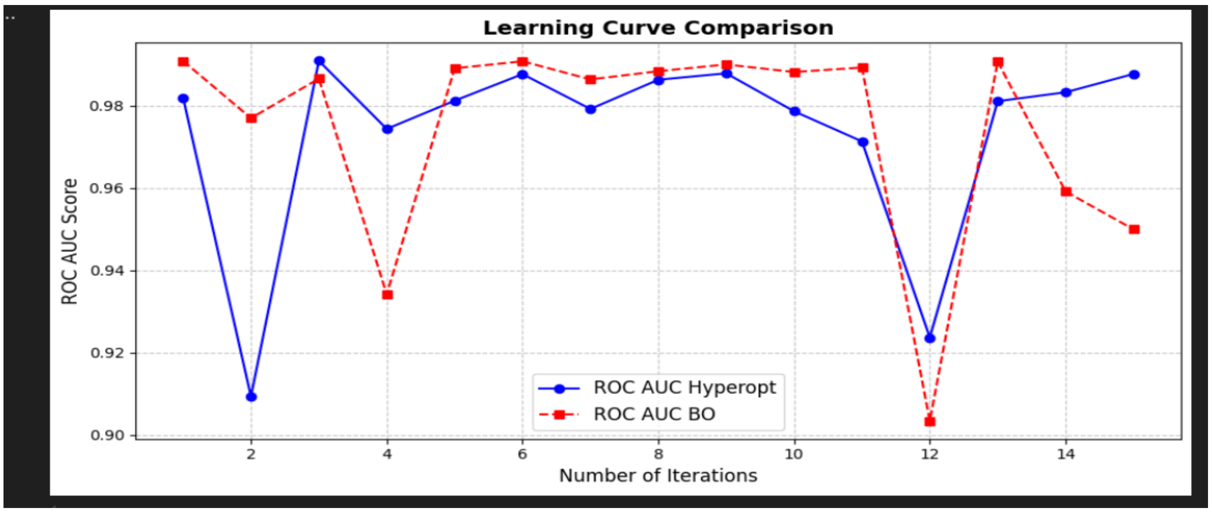
This iterative approach efficiently explores the hyperparameter space, leveraging a surrogate model to guide the search towards optimal configurations, suitable for enhancing the performance of machine learning models and similar complex systems.

## Also used hyper-opt library and random forest classifier with default parameters to compare above techniques

➢ **Results**

Type of Optimization	Cross validation score	ROC-AUC
Random forest with default hyper-parameter	0.8055	0.9328482734454095
Tree-structured Parzen Estimator (TPE)	0.899375	0.9907218678503817
Bayesian Optimization	0.9085000000000001	0.9908662816302052
Hyper-opt	0.909	0.9909808792964891

➢ Learning Curve Distribution of Bayesian v/s Hyperopt



➢ Observations

After hyperparameter tuning, notable improvements are observed in model performance metrics. Both the cross-validation scores and ROC-AUC show significant increases. Additionally:

- The **Cross-Validation Score** indicates improved generalization capability of the model across different datasets or folds, suggesting enhanced reliability.
- The **ROC-AUC (Receiver Operating Characteristic - Area Under the Curve)** score reflects improved discriminatory power of the model in classification tasks, indicating better separation of classes.
- **Comparison of Optimization Methods**:
  - Scores obtained using **Hyperopt** are generally better than those from **Bayesian Optimization**, which in turn also outperform results from **Tree-structured Parzen Estimator (TPE)** and straightforward **Random Forest** models.

These observations highlight the effectiveness of systematic hyperparameter tuning in enhancing model performance, with different optimization algorithms yielding varying degrees of improvement.

➢ Conclusion

• **Bayesian Optimization** emerges as a robust method for hyperparameter tuning, effectively navigating the hyperparameter space through a balanced approach of exploration and exploitation.

• This method often outperforms traditional techniques like grid search and random search by efficiently directing search efforts towards promising areas of the parameter space, thereby achieving superior results with fewer iterations.

• Throughout this report, we demonstrated the implementation of Bayesian Optimization using the scikit-optimize library to optimize hyperparameters for a RandomForestClassifier. By formulating a well-defined objective function and specifying a suitable hyperparameter space, we successfully identified a set of parameters that maximized the model's accuracy.

• Additionally, we compared the performance of our Bayesian Optimization approach with that of Hyperopt using a learning distribution curve. This comparison illustrated the effectiveness of Bayesian Optimization in achieving competitive results in a resource-efficient manner.

• **Tree-structured Parzen Estimator (TPE)**, while similar to Bayesian Optimization, operates by modeling and refining the probability distributions of hyperparameters based on their performance. In comparison to traditional Bayesian methods, TPE often exhibits faster convergence to optimal solutions, leveraging its adaptive search strategy to more effectively navigate complex parameter spaces.

• Bayesian Optimization and TPE are versatile techniques applicable across various machine learning models and performance metrics, making them indispensable tools for optimizing model performance in diverse real-world applications.