Automated Hyperparameter Optimization using TPE

UTKARSH GUPTA 22113157

Hyperparameters vs Parameters

Hyperparameter are parameters whose values control the learning process and determine the value of model parameters that a learning algorithm ends up learning, The prefix 'hyper' suggests that they are 'top-level' parameters that control the learning process and the model parameters that result from it.

Hyperparameters are used by the learning algorithm when it is learning but they are not part of the resulting model. At the end of the learning process, we have the trained model parameters which effectively is what we refer to as the model. The hyperparameters that were used during training are not part of this model. We cannot for instance know what hyperparameter values were used to train a model from the model itself, we only know the model parameters that were learned.

Basically, anything in machine learning and deep learning that you decide their values or choose their configuration before training begins and whose values or configuration will remain the same when training ends is a hyperparameter.

Parameters on the other hand are internal to the model. That is, they are learned or estimated purely from the data during training as the algorithm used tries to learn the mapping between the input features and the labels or targets.

Model training typically starts with parameters being initialized to some values (random values or set to zeros). As training/learning progresses the initial values are updated using an optimization algorithm (e.g. gradient descent). The learning algorithm is continuously updating the parameter values as learning progress but hyperparameter values set by the model designer remain unchanged.

Hyperparameter Tuning

When you're training machine learning models, each dataset and model needs a different set of hyperparameters, which are a kind of variable. The only way to determine these is through multiple experiments, where you pick a set of hyperparameters and run them through your model. This is called *hyperparameter tuning*. You're training your model sequentially with different sets of hyperparameters. This process can be manual, or you can pick one of several automated hyperparameter tuning methods.

Whichever method you use, you need to track the results of your experiments. You'll have to apply some form of statistical analysis, such as the loss function, to determine which set of hyperparameters gives the best result. Hyperparameter tuning is an important and computationally intensive process.

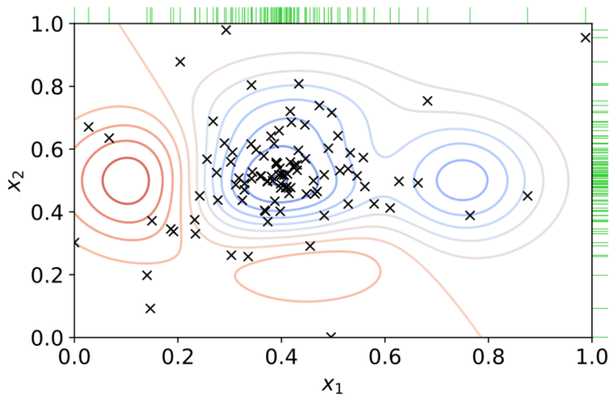
Hyperparameters directly control model structure, function, and performance. Hyperparameter tuning allows data scientists to tweak model performance for optimal results. This process is an essential part of machine learning, and choosing appropriate hyperparameter values is crucial for success.

For example, assume you're using the learning rate of the model as a hyperparameter. If the value is too high, the model may converge too quickly with suboptimal results. Whereas if the rate is too low, training takes too long, and results may not converge. A good and balanced choice of hyperparameters results in accurate models and excellent model performance.

There are many different techniques through which hyperparameter optimisation can be achieved, most commonly used are Bayesian Optimisation, Grid Search, Random Search, Tree Parzen Estimator.

For this project I have used **Tree-Parzen Estimator (TPE)** to optimize the hyperparameter an make it compatible with various models.

TREE-PARZEN ESTIMATOR (TPE)



TPE is a Bayesian optimization algorithm. This means it allows us to start with some initial beliefs about what our best model hyperparameters are, and update these beliefs in a principled way as we learn how different hyperparameters impact model performance. This is already a significant improvement over Grid Search and Random Search! Instead of determining the best hyperparameter set through trial and error, over time we can try more combinations of hyperparameters which lead to good models and fewer that do not.

Generally, the algorithm consists of the following steps:

1. Define a domain of hyperparameter search space,
2. Create an objective function which takes in hyperparameters and outputs a score (e.g., loss, root mean squared error, cross-entropy) that we want to minimize,
3. Get couple of observations (score) using randomly selected set of hyperparameters,
4. Sort the collected observations by score and divide them into two groups based on some quantile. The first group (x1) contains observations that gave the best scores and the second one (x2) - all other observations,
5. Two densities l(x1) and g(x2) are modelled using Parzen Estimators (also known as kernel density estimators) which are a simple average of kernels centered on existing data points,
6. Draw sample hyperparameters from l(x1), evaluating them in terms of l(x1)/g(x2), and returning the set that yields the minimum value under l(x1)/g(x1) corresponding to the greatest expected improvement. These hyperparameters are then evaluated on the objective function.
7. Update the observation list from step 3
8. Repeat step 4-7 with a fixed number of trials or until time limit is reached

HPO System Review

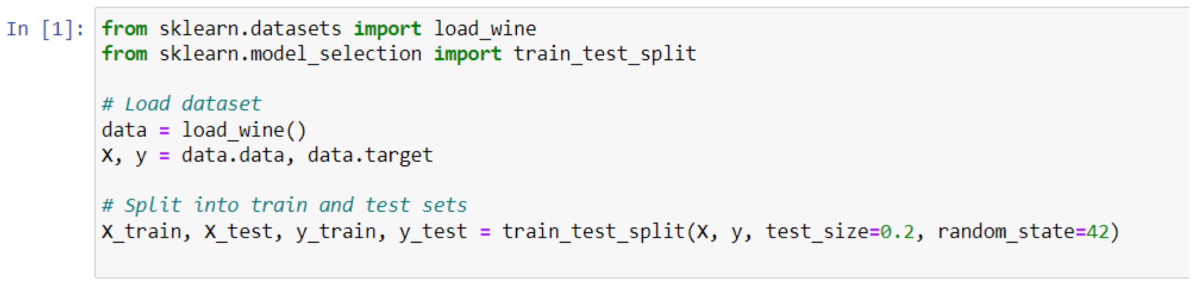
System has been optimized for three models i.e. **Random Forest Classifier, Gradient Boost Classifier and Logistic Regression.** Basic structure of the system remains same in all three models with some minor changes in system for Logistic Regression.

1. Introduction

The performance of a machine learning model is highly dependent on its hyperparameters. Traditionally, selecting the best hyperparameters involves manual tuning, which can be time-consuming and suboptimal. In this project, I have develop an automated hyperparameter optimization (HPO) system using custom AutoML techniques that can efficiently identify the best hyperparameter configuration for a given machine learning model and dataset. The system utilizes methods like Kernel Density Estimation (KDE) for modelling and sampling, similar to techniques used in Hyperopt, but is implemented from scratch for flexibility and understanding.

2. Data Preparation

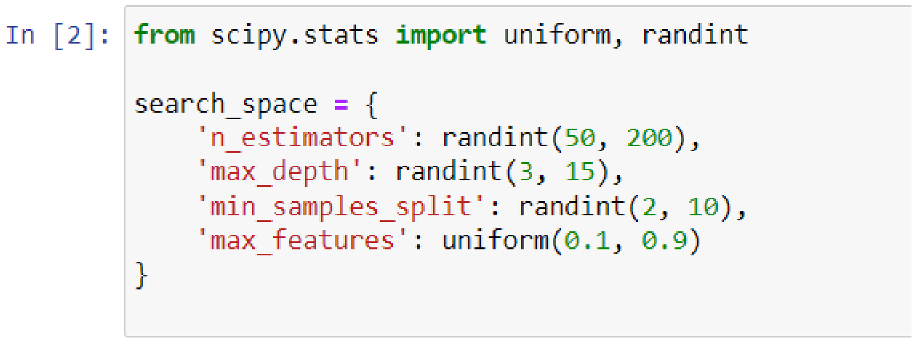
I have used the Wine dataset from the UCI Machine Learning Repository for this project. The dataset consists of 178 instances with 13 features and 3 target classes.



```
In [1]: from sklearn.datasets import load_wine
        from sklearn.model_selection import train_test_split

        # Load dataset
        data = load_wine()
        X, y = data.data, data.target

        # Split into train and test sets
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3. Hyperparameter Search Space Definition

We define the search space for hyperparameters using distributions that are suitable for each parameter of the RandomForestClassifier.

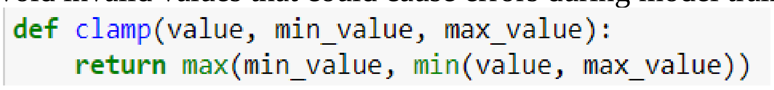


```
In [2]: from scipy.stats import uniform, randint

        search_space = {
            'n_estimators': randint(50, 200),
            'max_depth': randint(3, 15),
            'min_samples_split': randint(2, 10),
            'max_features': uniform(0.1, 0.9)
        }
```

Step 4: Objective Function for Optimization

The objective function is a critical part of the hyperparameter optimization process. It evaluates a given set of hyperparameters by training a model and measuring its performance. In this project, we use cross-validation to compute the mean ROC AUC score as the performance metric. The steps in the objective function are as follows:

1. **Parameter Clamping**:
    - Ensure that each hyperparameter is within its valid range. This is done to avoid invalid values that could cause errors during model training.

      ```
      def clamp(value, min_value, max_value):
          return max(min_value, min(value, max_value))
      ```
    -
2. **Model Initialization**:
    - Initialize the RandomForestClassifier with the given set of hyperparameters. This includes the number of estimators (n_estimators), maximum depth of trees

(max_depth), minimum number of samples required to split a node (min_samples_split), and the maximum number of features considered for splitting (max_features).

```
def objective_function(params, X, y):
    n_estimators = int(clamp(params['n_estimators'], 50, 200))
    max_depth = int(clamp(params['max_depth'], 3, 15))
    min_samples_split = int(clamp(params['min_samples_split'], 2, 10))
    max_features = clamp(params['max_features'], 0.1, 1.0)

    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        max_features=max_features,
        random_state=42
    )
```

3. **Label Binarization**:
    - Convert the target labels into a binary format. This step is necessary for computing the ROC AUC score, especially in a multiclass setting.

```
lb = LabelBinarizer()
y_bin = lb.fit_transform(y)
cv_scores = []
```

4. **Cross-Validation Setup**:
    - Set up a StratifiedKFold cross-validator to ensure that each fold has a similar distribution of target classes. This helps in getting a more reliable estimate of model performance.
5. **Cross-Validation Loop**:
    - Loop over each fold of the cross-validation split.
    - Train the model on the training set of the fold.
    - Predict probabilities on the validation set of the fold.
    - Compute the ROC AUC score for the fold and store it.

```
skf = StratifiedKFold(n_splits=5)
for train_idx, test_idx in skf.split(X, y):
    model.fit(X[train_idx], y[train_idx])
    y_prob = model.predict_proba(X[test_idx])
```

6. **Compute Mean ROC AUC Score**:
    - Compute the mean ROC AUC score across all folds. This average score is used as the objective function value, which we aim to minimize during optimization.

```
if y_bin.shape[1] == 1:
    roc_auc = roc_auc_score(y[test_idx], y_prob[:, 1])
else:
    roc_auc = roc_auc_score(y_bin[test_idx], y_prob, average='macro')

cv_scores.append(roc_auc)
```

Step 5: Hyperparameter Optimization Process

The hyperparameter optimization process involves generating initial observations, fitting Kernel Density Estimators (KDE) to model promising and less promising regions, and sampling new hyperparameters to evaluate.

1. **Helper Function to Generate Random Hyperparameters**:

- get_random_hyperparameters generates random hyperparameters within the specified search space using their respective distributions.

```python
def get_random_hyperparameters(search_space):
    return {key: dist.rvs() for key, dist in search_space.items()}
```

NOTE: for Logistic Regression we use following code, since params in Logistic Regression are in List.

```python
def get_random_hyperparameters(search_space):
    params = {}
    for key, dist in search_space.items():
        if isinstance(dist, list):
            params[key] = random.choice(dist)

    return params
```

2. **Generate Initial Observations**:

- initial_observations generates a specified number (n) of initial observations by evaluating randomly selected sets of hyperparameters. Each observation consists of a set of hyperparameters and the corresponding score obtained from the objective_function.

```python
def initial_observations(n, search_space, X, y):
    observations = []
    for _ in range(n):
        params = get_random_hyperparameters(search_space)
        score = objective_function(params, X, y)
        observations.append((params, score))
    return observations
```

NOTE: Since params of Logistic Regression are in List so we have to first encode them in order to fit KDE on Observation

```python
def encode_params(params, search_space):
    encoded = []
    for key, dist in search_space.items():
        value = params[key]
        if isinstance(dist, list):
            encoded.append(dist.index(value))
        else:
            encoded.append(value)
    return encoded

def decode_params(encoded, search_space):
    params = {}
    for i, key in enumerate(search_space.keys()):
        dist = search_space[key]
        if isinstance(dist, list):
            index = int(encoded[i])
            index = max(0, min(index, len(dist) - 1))
            params[key] = dist[index]
        else:
            params[key] = encoded[i]
    return params
```

3. **Fit KDE on Observations**:

- fit_kde fits a Kernel Density Estimator (KDE) on the observed hyperparameters. KDE is used to model the distribution of promising hyperparameters.

```python
def fit_kde(observations):
    samples = np.array([list(obs[0].values()) for obs in observations])
    kde = KernelDensity(kernel='gaussian').fit(samples)
    return kde
```

4. **Sample New Hyperparameters from KDE**:

- sample_from_kde samples new hyperparameters from the KDE. This step allows the generation of new candidate hyperparameters based on the distribution learned from the promising observations.

```python
def sample_from_kde(kde, search_space):
    samples = kde.sample()
    return {key: samples[0][i] for i, key in enumerate(search_space.keys())}
```

5. **Main Optimization Function**:

- optimize_hyperparameters performs the hyperparameter optimization by iteratively updating the observations and sampling new hyperparameters.

```python
def optimize_hyperparameters(X, y, search_space, n_initial=10, n_iterations=50):
    observations = initial_observations(n_initial, search_space, X, y)

    for _ in range(n_iterations - n_initial):
        sorted_observations = sorted(observations, key=lambda x: x[1])
        split_point = int(len(sorted_observations) * 0.2)
        x1 = sorted_observations[:split_point]
        x2 = sorted_observations[split_point:]

        kde_x1 = fit_kde(x1)
        kde_x2 = fit_kde(x2)

        params = sample_from_kde(kde_x1, search_space)
        score = objective_function(params, X, y)
        observations.append((params, score))

    best_params = sorted(observations, key=lambda x: x[1])[0][0]
    return best_params
```

Detailed Explanation of the Optimization Process:

1. **Initial Observations**:
    - Generate n_initial observations by randomly sampling hyperparameters and evaluating their performance using the objective_function.

    observations = initial_observations(n_initial, search_space, X, y)

2. **Iterative Optimization**:
    - For the remaining iterations (n_iterations - n_initial), perform the following steps:

    3. **Sort Observations**:
        - Sort the observations based on their scores in ascending order. This allows us to identify the most promising hyperparameters.

        sorted_observations = sorted(observations, key=lambda x: x[1])

    4. **Split Observations**:
        - Split the sorted observations into two groups: the top 20% (promising region, x1) and the bottom 80% (less promising region, x2).

        split_point = int(len(sorted_observations) * 0.2)
        x1 = sorted_observations[:split_point]
        x2 = sorted_observations[split_point:]

    5. **Fit KDE**:
        - Fit KDEs on both the promising region (x1) and the less promising region (x2). This models the distribution of hyperparameters in both regions.

        kde_x1 = fit_kde(x1)
        kde_x2 = fit_kde(x2)

    6. **Sample New Hyperparameters**:
        - Sample new hyperparameters from the KDE fitted on the promising region (x1). This step leverages the learned distribution to generate new promising hyperparameter candidates.

```
params = sample_from_kde(kde_x1, search_space)
```

7. **Evaluate New Hyperparameters**:
   - Evaluate the new set of hyperparameters using the objective_function and append the result to the observations.

```
score = objective_function(params, X, y)
observations.append((params, score))
```

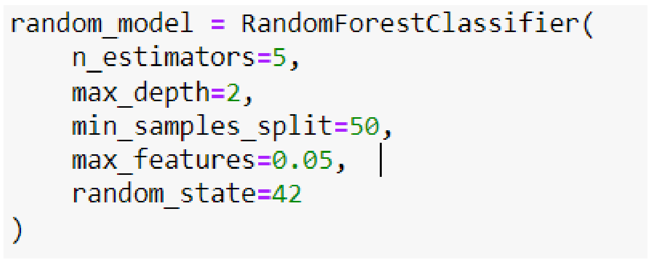
8. **Return Best Hyperparameters**:
   - After completing all iterations, return the best hyperparameters found based on the scores.

```
best_params = sorted(observations, key=lambda x: x[1])[0][0]
return best_params
```

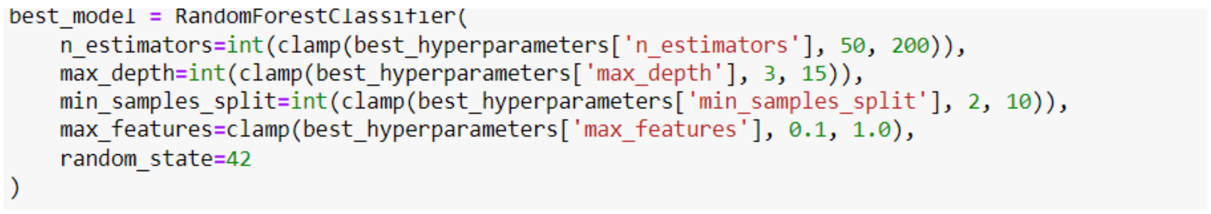
Step 6: Train Final Model with Optimized Hyperparameters

Once the best hyperparameters have been identified through the optimization process, we train a final model using these hyperparameters. This step ensures that we fully leverage the optimized hyperparameters to achieve the best possible model performance on our training data.

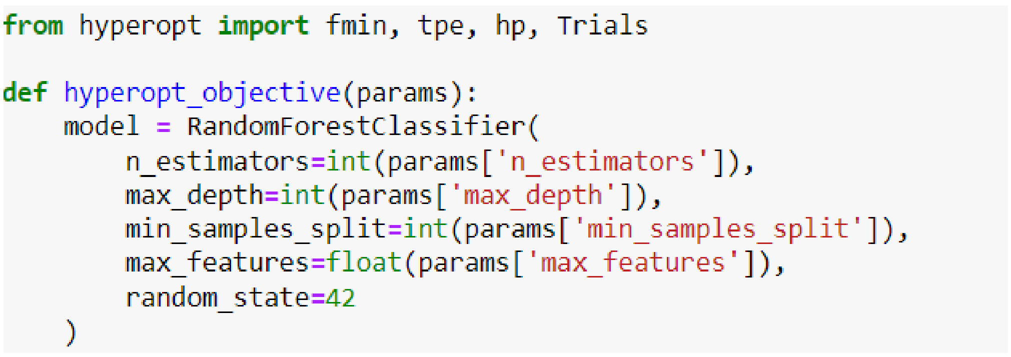
In order to appreciate the efficiency of model we will be training one model with random hyperparameter and other one with optimized hyperparameters using Hyperopt.



```python
random_model = RandomForestClassifier(
    n_estimators=5,
    max_depth=2,
    min_samples_split=50,
    max_features=0.05,
    random_state=42
)
```



```python
best_model = RandomForestClassifier(
    n_estimators=int(clamp(best_hyperparameters['n_estimators'], 50, 200)),
    max_depth=int(clamp(best_hyperparameters['max_depth'], 3, 15)),
    min_samples_split=int(clamp(best_hyperparameters['min_samples_split'], 2, 10)),
    max_features=clamp(best_hyperparameters['max_features'], 0.1, 1.0),
    random_state=42
)
```



```python
from hyperopt import fmin, tpe, hp, Trials

def hyperopt_objective(params):
    model = RandomForestClassifier(
        n_estimators=int(params['n_estimators']),
        max_depth=int(params['max_depth']),
        min_samples_split=int(params['min_samples_split']),
        max_features=float(params['max_features']),
        random_state=42
    )
```

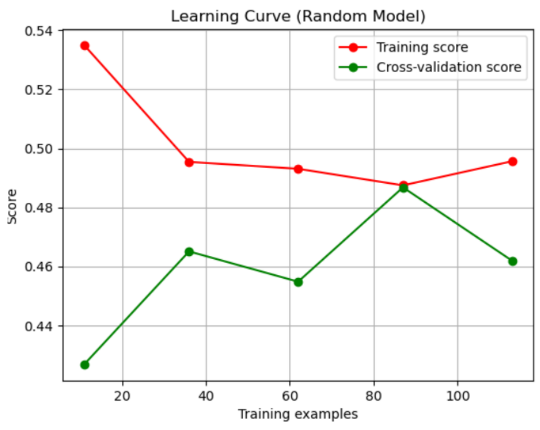
```
lb = LabelBinarizer()
y_bin = lb.fit_transform(y_train)
cv_scores = []

skf = StratifiedKFold(n_splits=5)
for train_idx, test_idx in skf.split(X_train, y_train):
    model.fit(X_train[train_idx], y_train[train_idx])
    y_prob = model.predict_proba(X_train[test_idx])

    if y_bin.shape[1] == 1:
        roc_auc = roc_auc_score(y_train[test_idx], y_prob[:, 1])
    else:
        roc_auc = roc_auc_score(y_bin[test_idx], y_prob, average='macro')

    cv_scores.append(roc_auc)

return -np.mean(cv_scores)
```

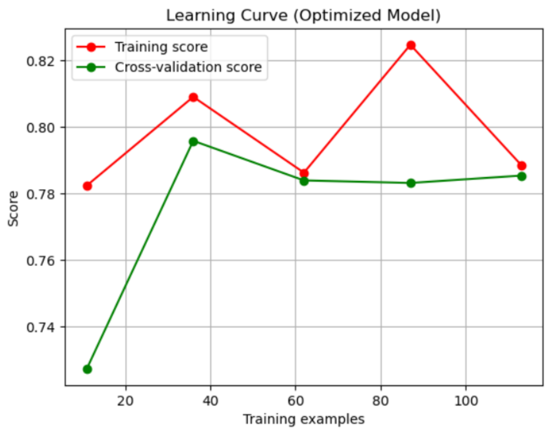
Step 7: Evaluate and Compare Models

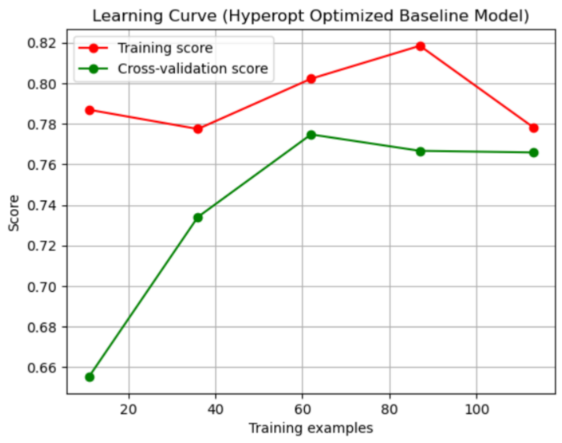
In this step, we evaluate and compare the performance of different models: the random model, the optimized model, and the Hyperopt model. This involves plotting the learning curves for these models to understand their performance better.

Learning curves show the performance of a model over time or as the training progresses. They are useful for diagnosing the bias-variance tradeoff and identifying whether the model is overfitting or underfitting.

Learning Curves for Logistic Regression are obtained as follows:



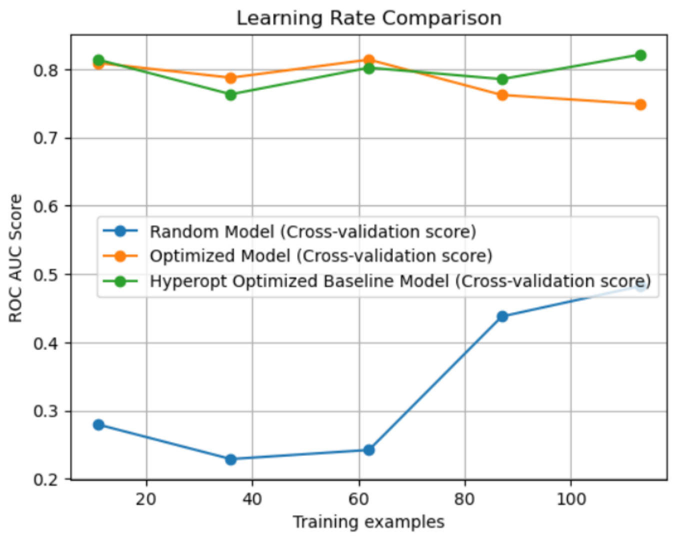




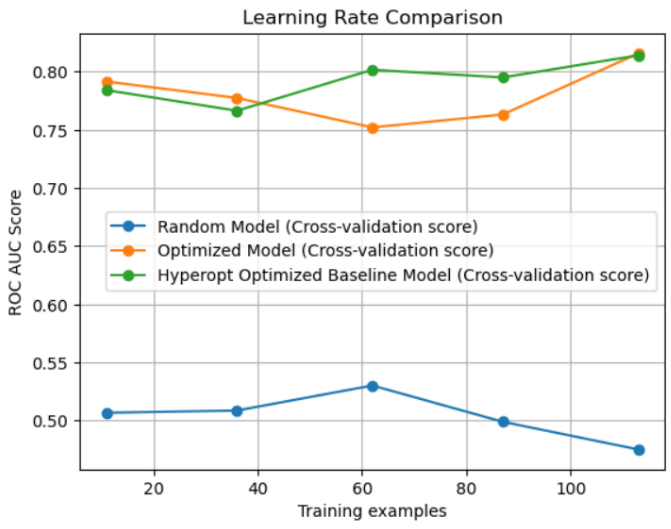
8. Compare Learning Rate Distribution Curves

We compare the learning rate distribution curves for the three models.

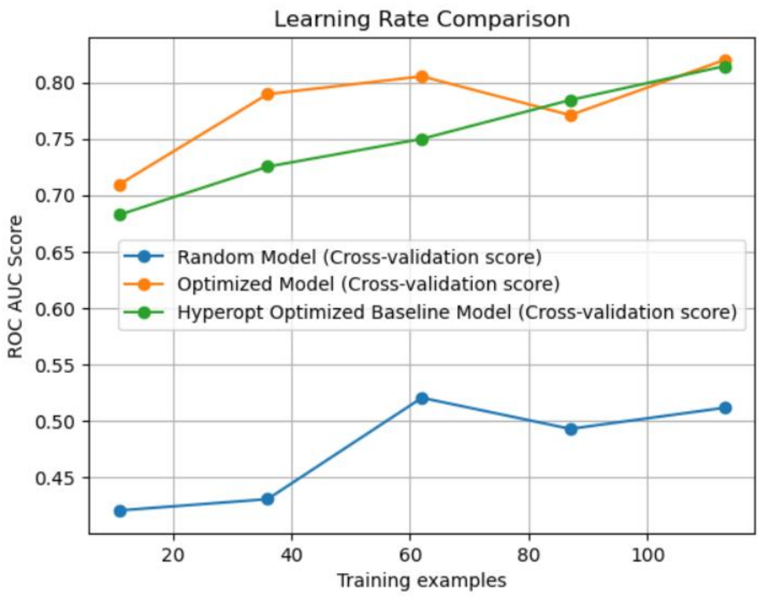
1. Random Forest Classifier



2. Gradient Boosting Classifier



3. Logistic Regression



Conclusions

Hyperparameter tuning is a critical part of the modeling process. While Grid Search and Random Search approaches are easy to implement, TPE as an alternative provides a more principled way of tuning hyperparameters and is pretty simple from a conceptual perspective. Several Python libraries exist with very good implementations of TPE including Hyperopt and Optuna .