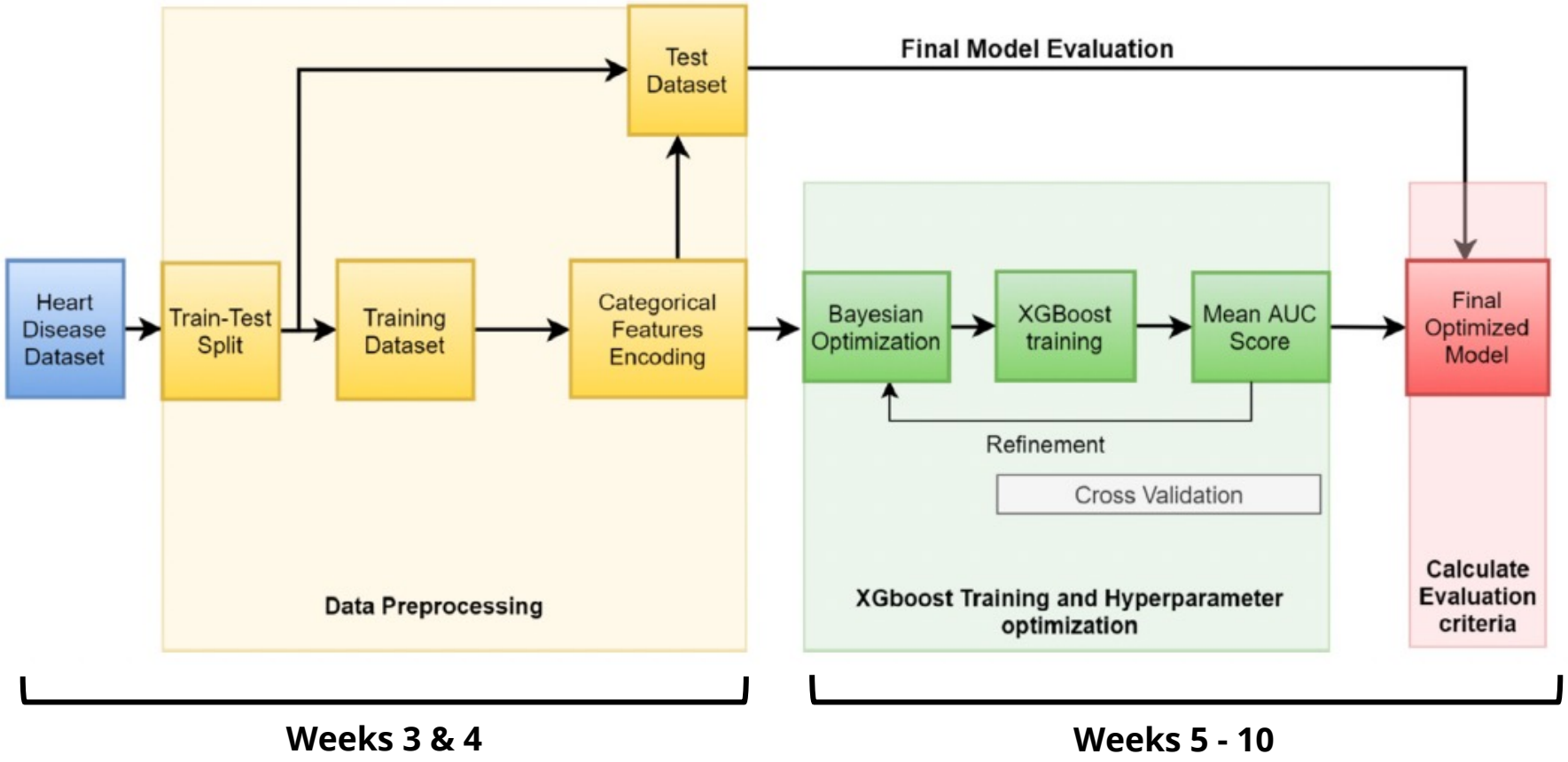
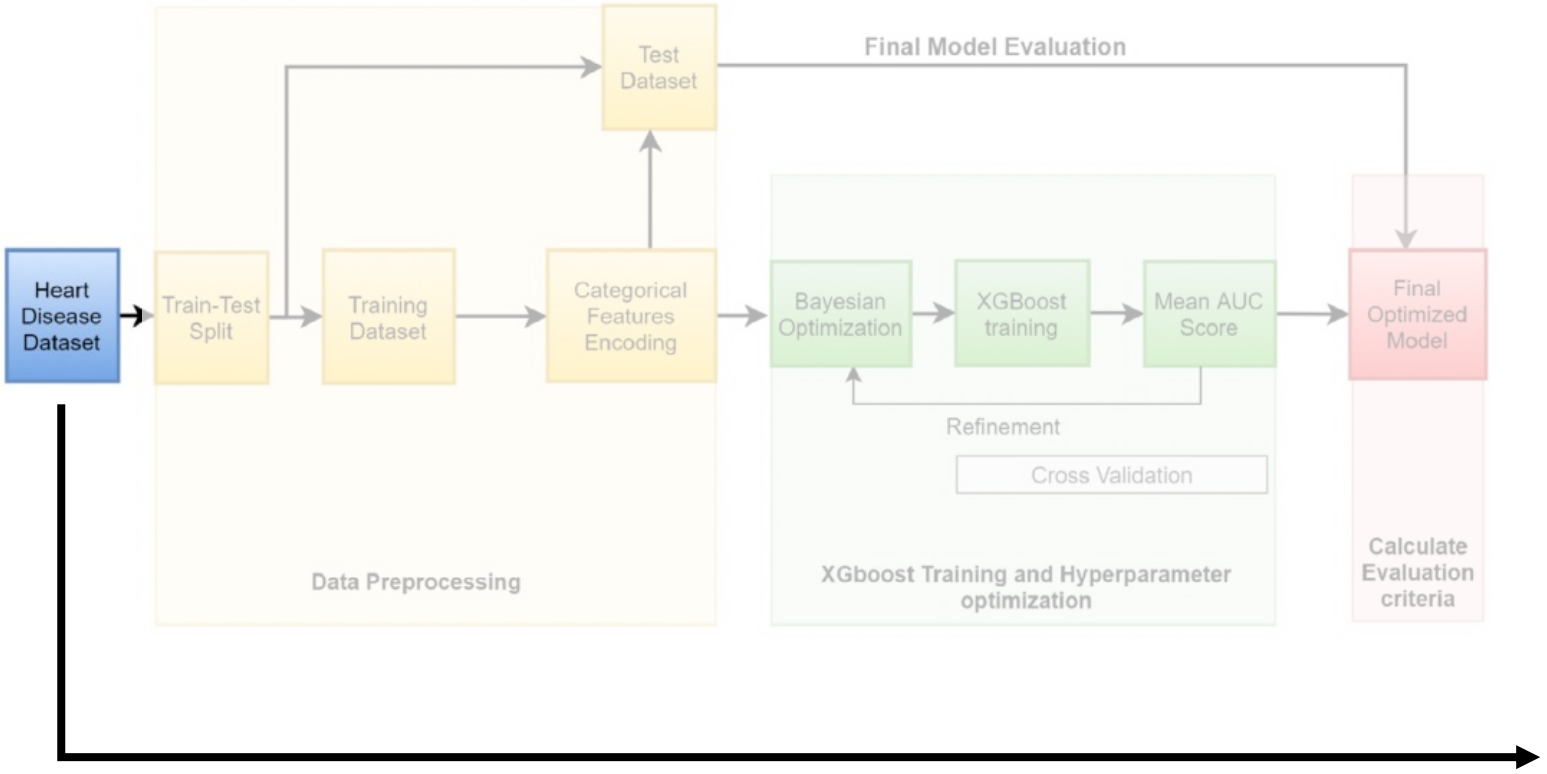
The Authors' Framework



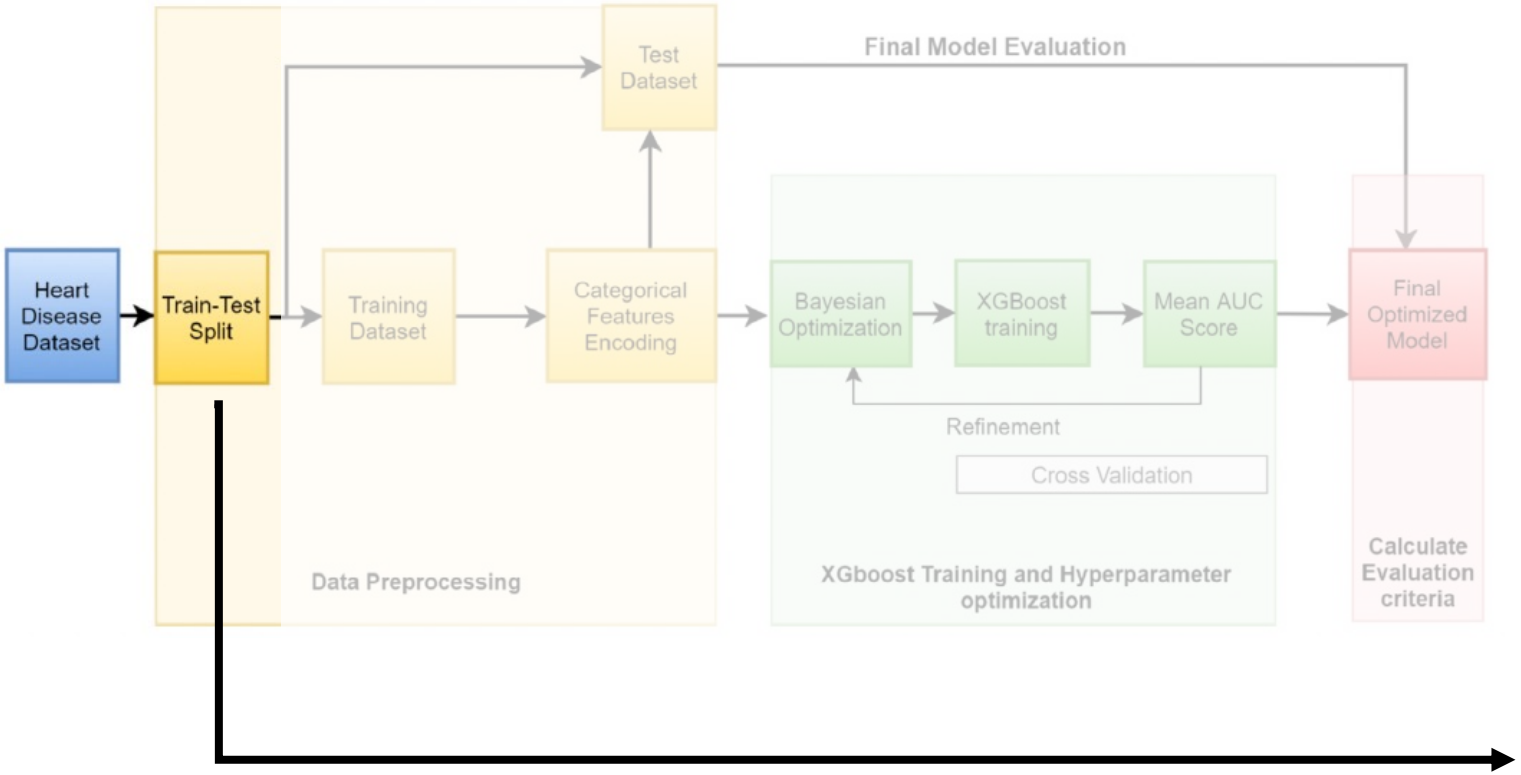
Step 1: Explore the Data



Before doing any modeling or model prep, it's ~~a good idea~~ essential to explore your data. You should:

- Take a look at each variable in the raw data to understand its composition and distribution.
- Explore the response variable to see if you need to transform it in any way + get a sense of any class imbalance
- Create some charts to visualize aspects of the data that you think might be interesting
- Review the data dictionary!

Step 2: Create a Train and Test Set

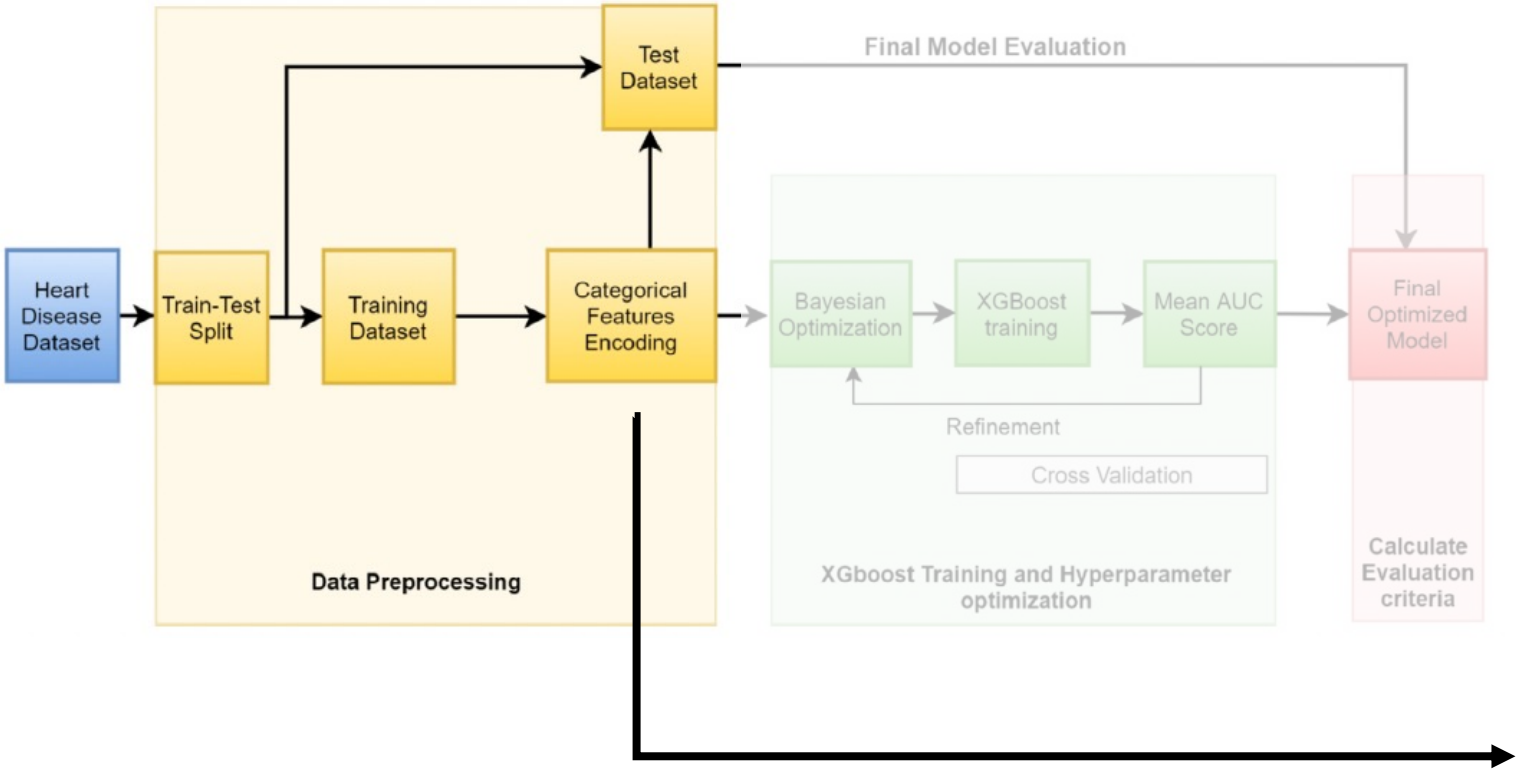


We need to split the data into a training set and a testing set so that we can build and evaluate our model. The authors use an 80/20 split.

One way to do this is to use the `test_train_split` function from `sklearn.model_selection`

NOTE: The authors split their data before the encoding step, but that's not necessary. You could transform your features using the full dataset and then execute the test/train split.

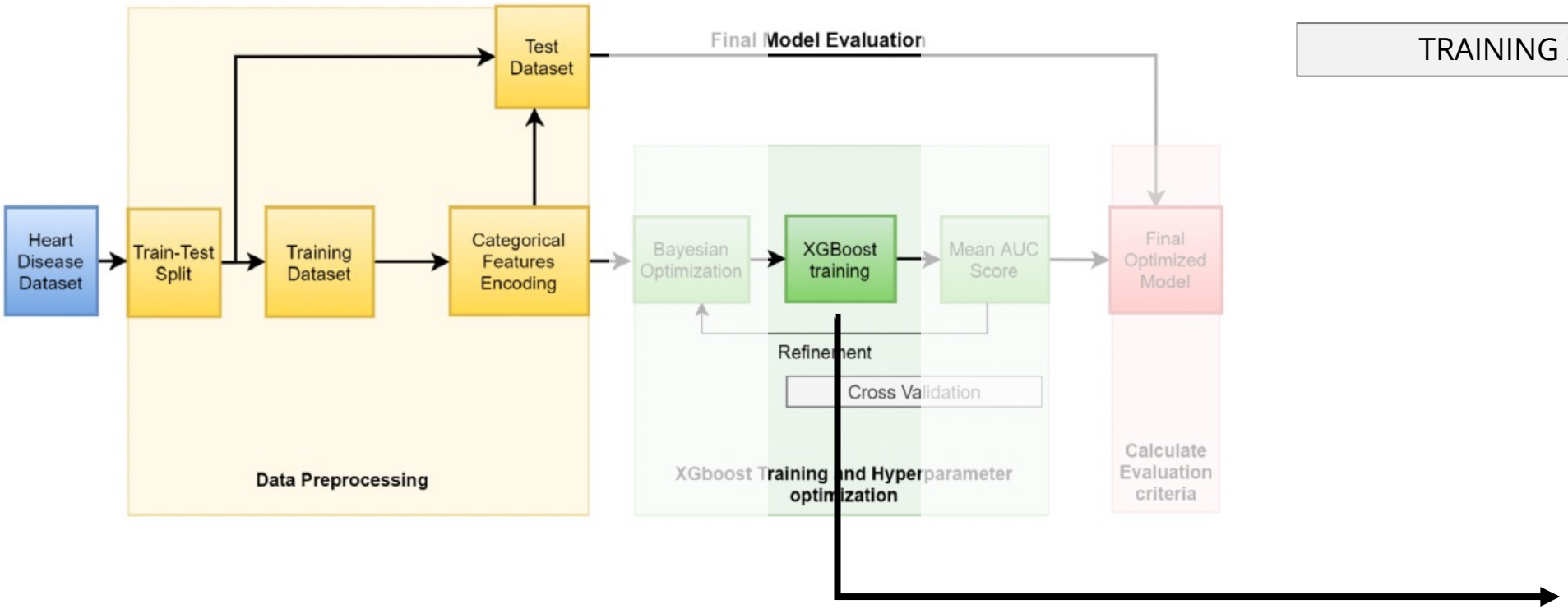
Step 3: Encode the Categorical Features



We need to transform the categorical features in the dataset into numerical features, and one hot encoding is a standard approach for doing this (assuming that your categorical features don't have inherent order and don't have too many unique values).

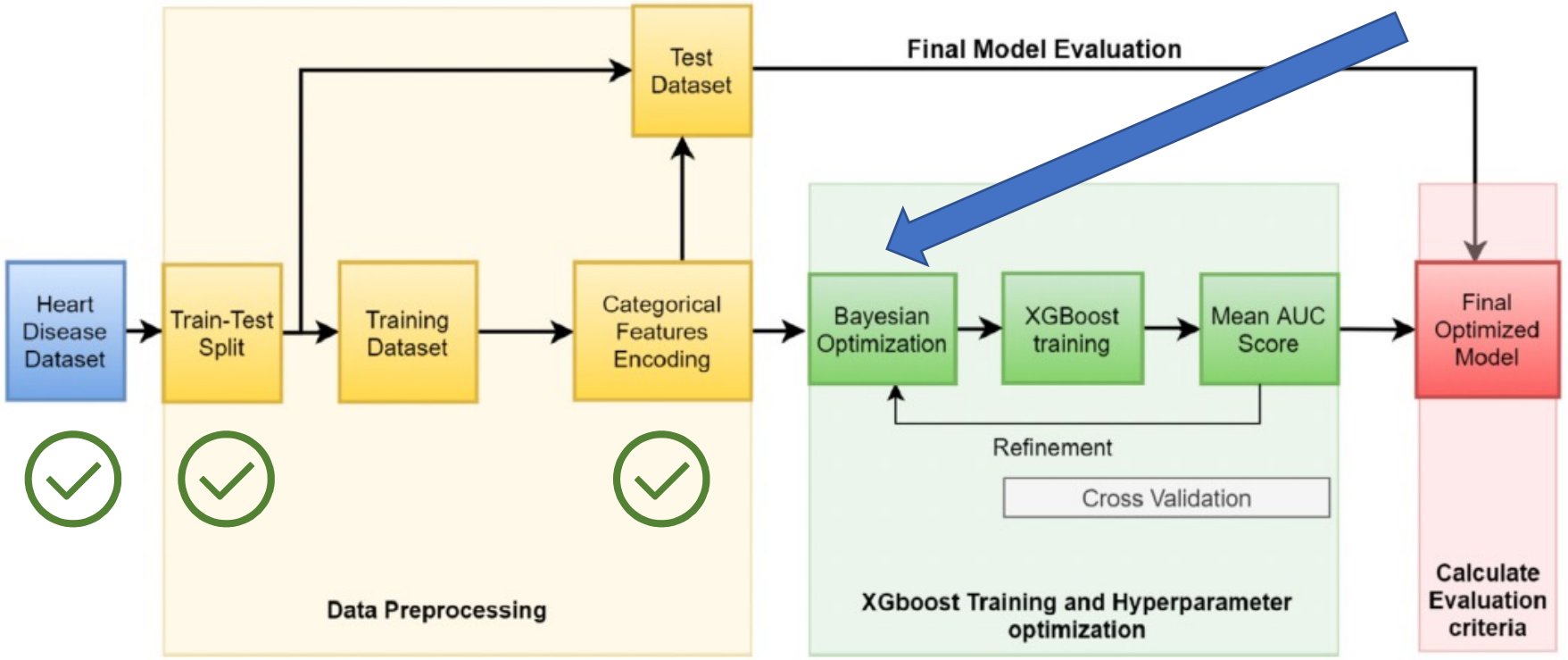
In the case of the heart data, we also need to transform the response variable (`num`) to binary.

Step 4: Build the Classifier

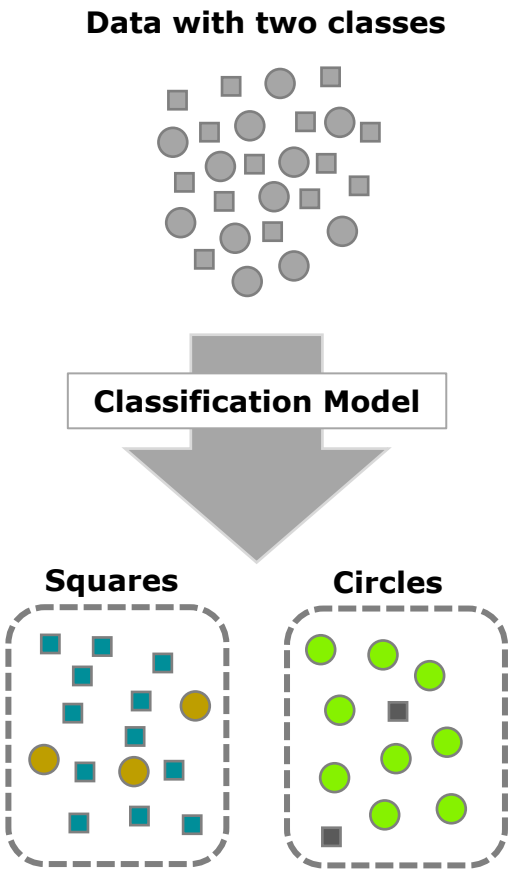


TRAINING XGBOOST MODEL

What's Next?


Heart Disease Dataset
Train-Test Split
Test Dataset
Training Dataset
Categorical Features Encoding
Data Preprocessing
Final Model Evaluation
Bayesian Optimization
XGBoost training
Mean AUC Score
Refinement
Cross Validation
XGboost Training and Hyperparameter optimization
Final Optimized Model
Calculate Evaluation criteria

The Confusion Matrix



	Actually Squares	Actually Circles
Predicted to be Squares	**True Negatives** Our model correctly identified these squares as squares.	**False Negatives** Our model incorrectly labeled these circles as squares.
Predicted to be Circles	**False Positives** Our model incorrectly identified these squares as circles.	**True Positives** Our model correctly labeled these circles as circles.

Evaluation Metrics

Accuracy:	What percentage of all circles and squares were labeled correctly? Higher accuracy indicates that the model can more readily distinguish circles from squares.
Sensitivity (Recall):	What percentage of all circles were labeled as circles? Higher recall indicates that the model can more readily identify circles in the data.
Specificity:	What percentage of all squares were labeled as squares? Higher specificity indicates that the model can readily identify squares in the data.
Precision:	What percentage of all shapes predicted as circles were actually circles? Higher precision indicates that the model can more readily distinguish circles.

	Actually Squares	**Actually Circles**
Predicted to be Squares	**True Negatives** Our model correctly identified these squares as squares.	**False Negatives** Our model incorrectly labeled these circles as squares.
Predicted to be Circles	**False Positives** Our model incorrectly identified these squares as circles.	**True Positives** Our model correctly labeled these circles as circles.

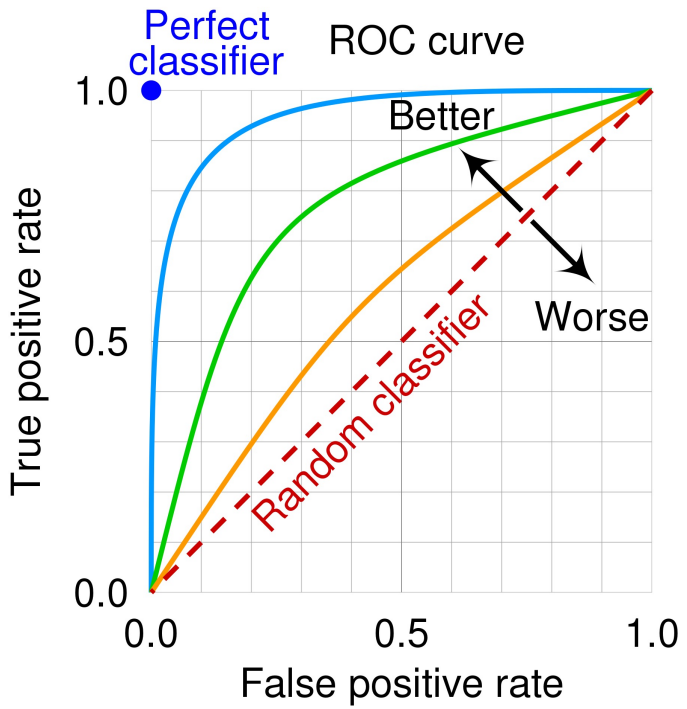
Evaluation Metrics

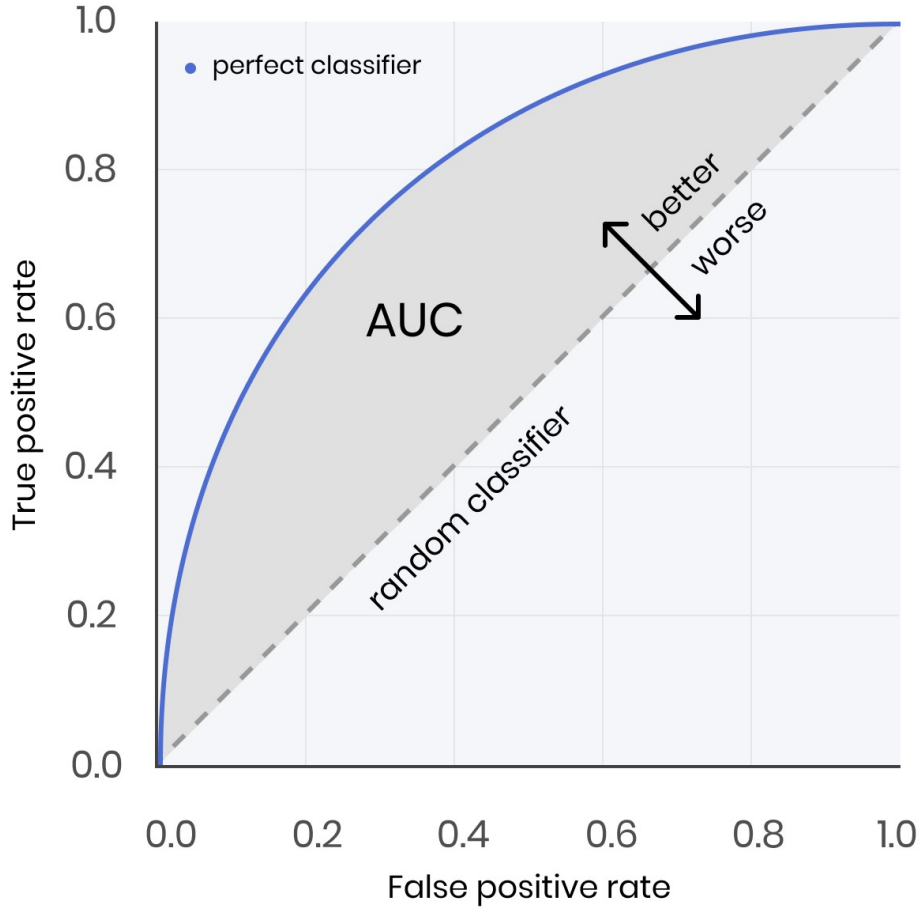
Accuracy:	$\frac{\sum TP + \sum TN}{\sum TP + \sum TN + \sum FP + \sum FN}$	**0.68**
Sensitivity (Recall):	$\frac{\sum TP}{\sum TP + \sum FN}$	**0.75**
Specificity:	$\frac{\sum TN}{\sum TN + \sum FP}$	**0.60**
Precision:	$\frac{\sum TP}{\sum TP + \sum FP}$	**0.65**

Consider a case where we had 20 squares and 20 circles in our dataset. We trained a classifier and here's the result:

	Actually Squares	Actually Circles
Predicted to be Squares	True Negatives: 12	False Negatives: 5
Predicted to be Circles	False Positives: 8	True Positives: 15

ROC Curve and AUC





True Positive Rate = Sensitivity = $\frac{\sum TP}{\sum TP + \sum FN}$

False Positive Rate = (1 – Specificity) = 1 - $\frac{\sum TN}{\sum TN + \sum FP}$

What is a Hyperparameter?

PARAMETERS

A **parameter** is a configuration variable that can be estimated from the data. Parameters are:

- required by the model when making predictions.
- estimated or learned from data.
- often not set manually by the practitioner.
- often saved as part of the learned model.

The coefficients estimated from fitting a linear regression model are parameters. These are learned from the data and are necessary to make predictions using new data.

HYPERPARAMETERS

In contrast, a **hyperparameter** is a configuration that is external to the model and whose value cannot be estimated from data. Hyperparameters are:

- often used in processes to help estimate model parameters.
- often specified by the practitioner.
- often be set using heuristics.
- **often tuned for a given predictive modeling problem**.

The value for k in a k-means clustering model is a hyperparameter. More complex models, such as XGBoost, have a multitude of tunable hyperparameters.

Borrowed from: https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/

XGBoost Hyperparameters

XGBoost has a number of hyperparameters. Here we'll focus on the 9 hyperparameters that the authors tune:

Hyperparameter	Description	Value Range
learning_rate	Boosting learning rate	0 – 1 (default == 0.3)
n_estimators	Number of trees in the ensemble (== to # of boosting rounds)	Int > 0 (default == 100)
min_child_weight	Minimum sum of instance weight needed in a child	Float > 0 (default == 1)
max_depth	Maximum tree depth for base learners	Int > 0 (default == 6)
subsample	Subsample ratio of the training instance	0 – 1 (default == 1)
colsample_by_tree	Subsample ratio of columns when constructing each tree	0 – 1 (default == 1)
gamma	Minimum loss reduction required to make additional leaf node partition	Float > 0 (default == 0)
reg_lambda	L2 regularization term on weights (Ridge)	Float > 0 (default == 1)
reg_alpha	L1 regularization term on weights (Lasso)	Float > 0 (default == 0)

Hyperparameter Tuning Methods

What are we trying to do?

By tuning our hyperparameters we're attempting to find the "settings" of the model that return the best performance as measured by the test set. We can think of the goal expressed this way.

$$x^\star = \arg\min_{x \in \mathcal{X}} f(x)$$

We have our objective score that we're trying to minimize, f(x). X_star is the set of hyperparameters chosen from the set X that yields the lowest value of the score.

What are our options for optimizing our hyperparameters?

Well, we could just manually choose values for each hyperparameter in some kind of iterative way and see how those choices impact the model result. If we have one thing to tune, then this could be a possible approach, but you can probably see how this manual process becomes laughably inefficient (or impossible!) quickly.

Luckily, we have more automated options for searching through the possible values for our hyperparameters.

Borrowed from: https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f

Hyperparameter Tuning Methods

Grid Search

In a grid search, we specify the hyperparameters we're interested in tuning as well as the values we'd like to try. This obviously requires some research into what values are reasonable for our HPs, but we should do that anyway. After specifying our range of HPs and values, we build models iteratively with each combination of HPs and score them to determine which combination yields the best result.

Grid searches are a bit inefficient though since they don't take into account past results as the search progresses. For instance, we might notice that the model does not perform better as we increase the value of n_estimators, but our grid search approach won't take that knowledge into account (so, it can be a bit inefficient).

```
hyperparameter_grid = {
    'n_estimators': [100, 200, 300, 400, 500, 600],
    'max_depth': [2, 5, 10, 15, 20, 25, 30, 35, 40],
    'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8]
}
```

Borrowed from: https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f

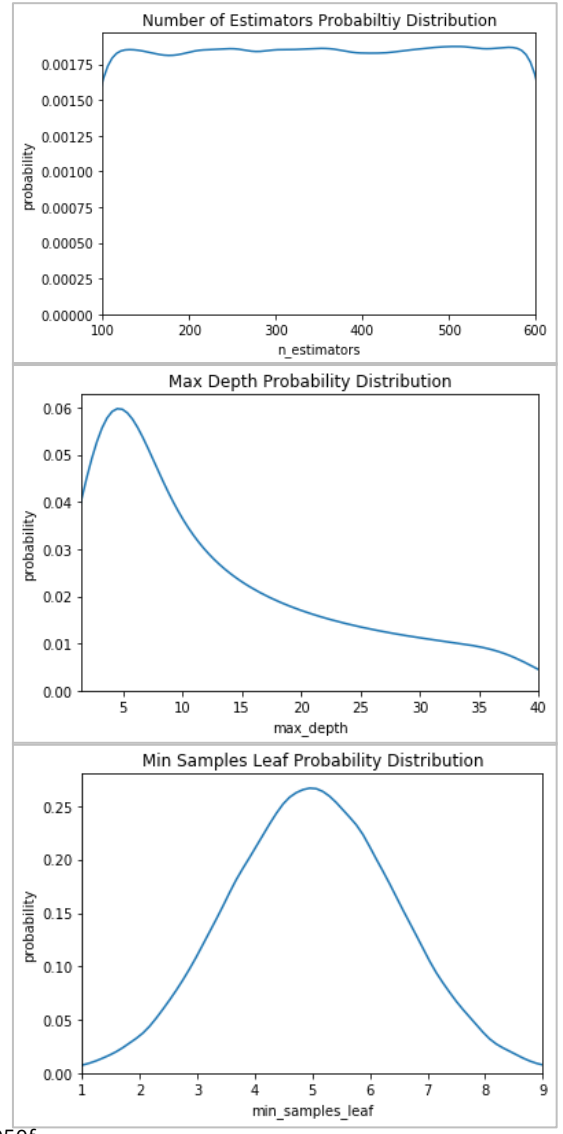
Hyperparameter Tuning Methods

Bayesian Optimization

In contrast to a grid search, Bayesian optimization approaches do keep track of past results in order to inform future sets of HPs to use. This is possible because these approaches construct a “surrogate probability model” of the objective function and then update that surrogate model as more information becomes available. In a stepwise fashion we end up with:

1. Build a surrogate probability model of the objective function
2. Find the hyperparameters that perform best on the surrogate
3. Apply these hyperparameters to the true objective function
4. Update the surrogate model incorporating the new results

P(score | hyperparameters)



Borrowed from: https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f