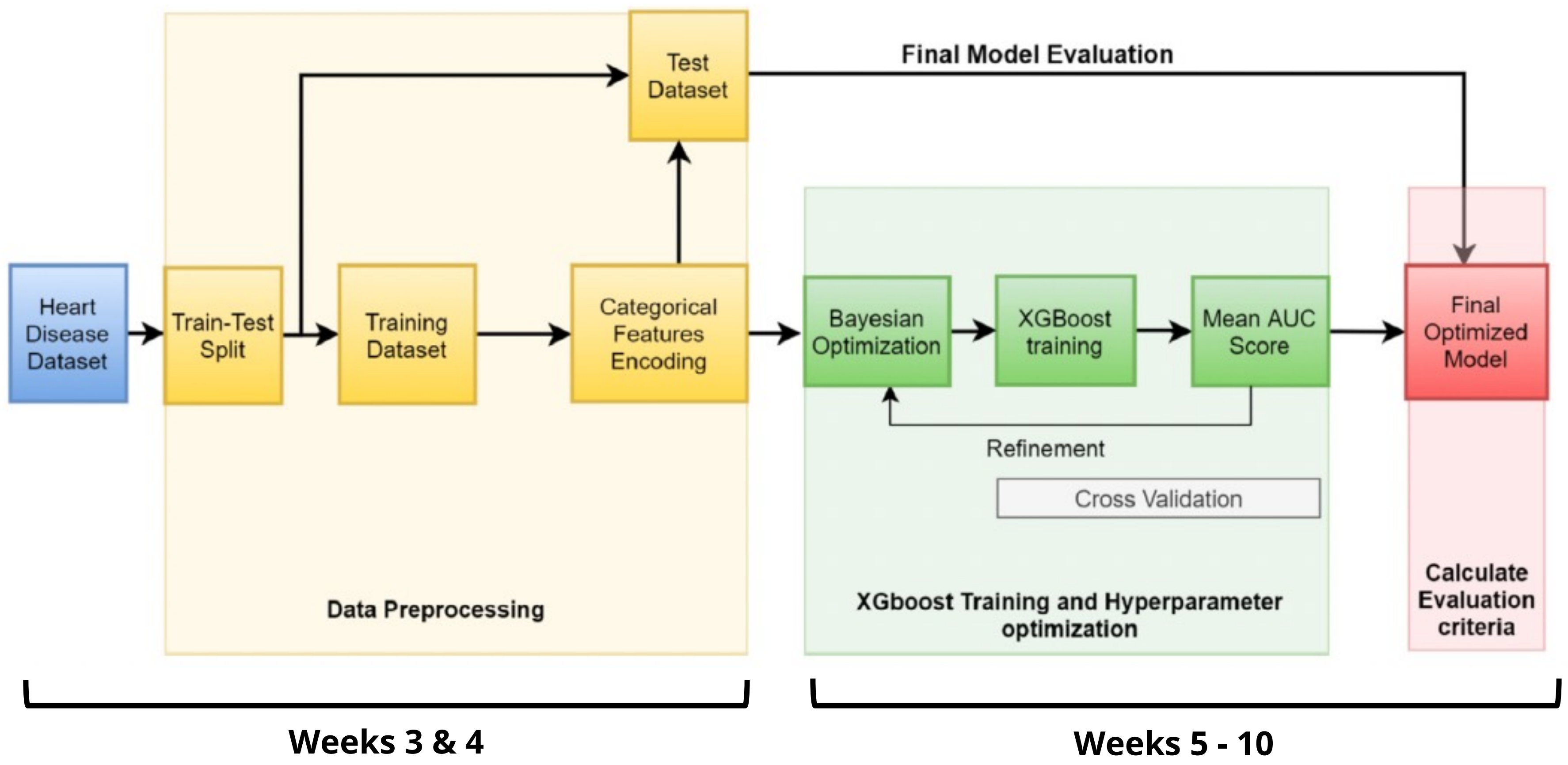
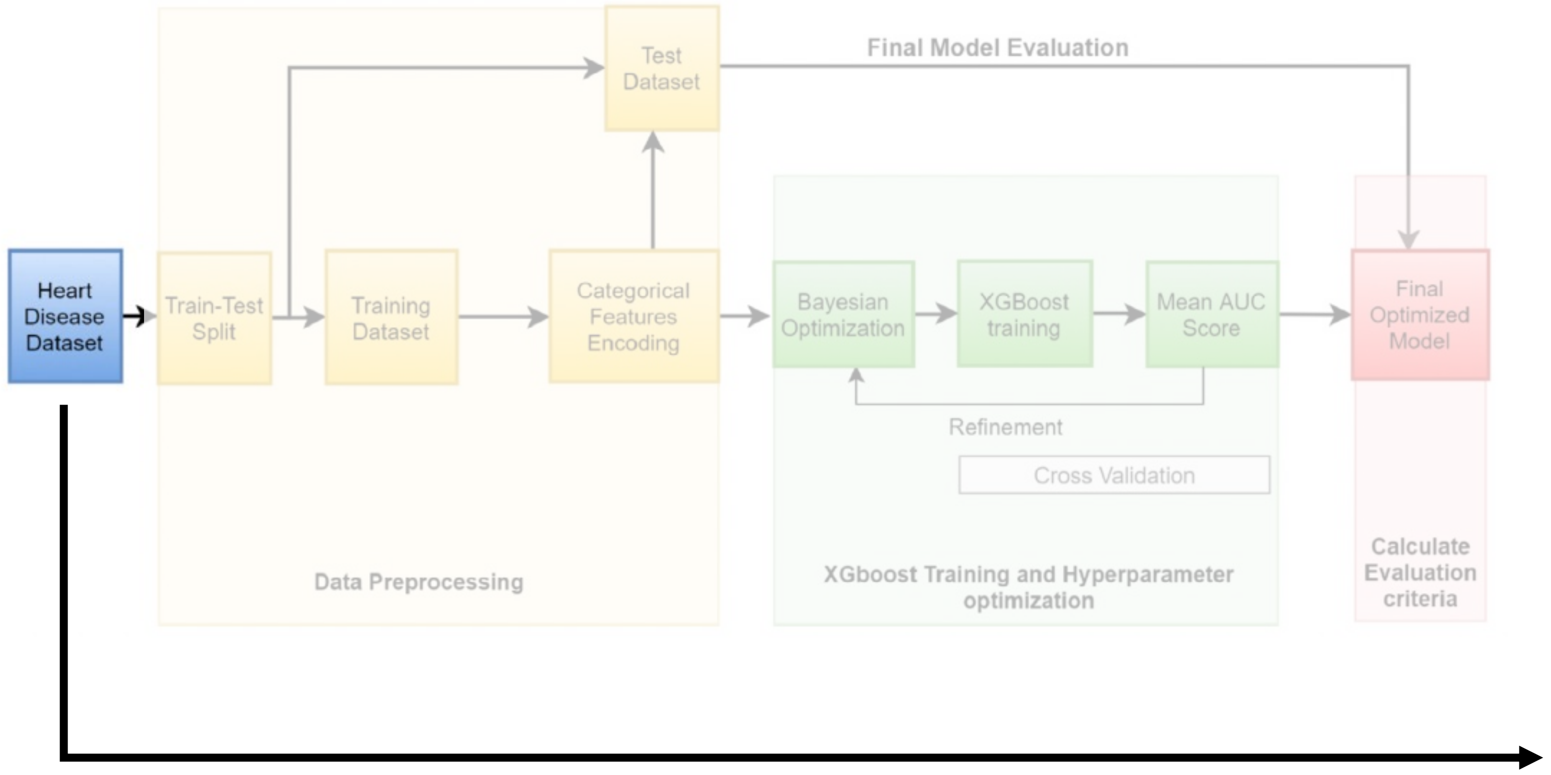
The Authors' Framework



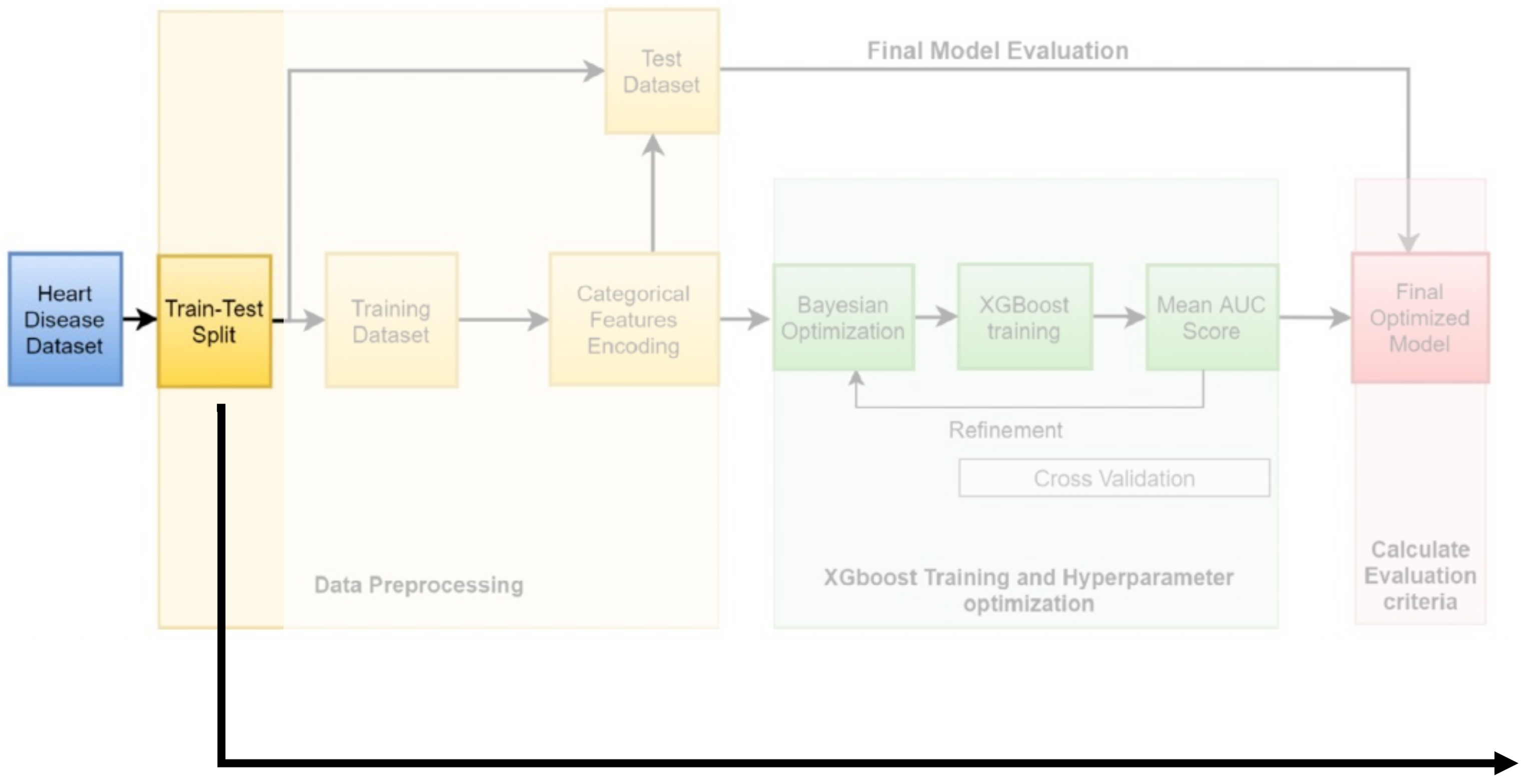
Step 1: Explore the Data



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

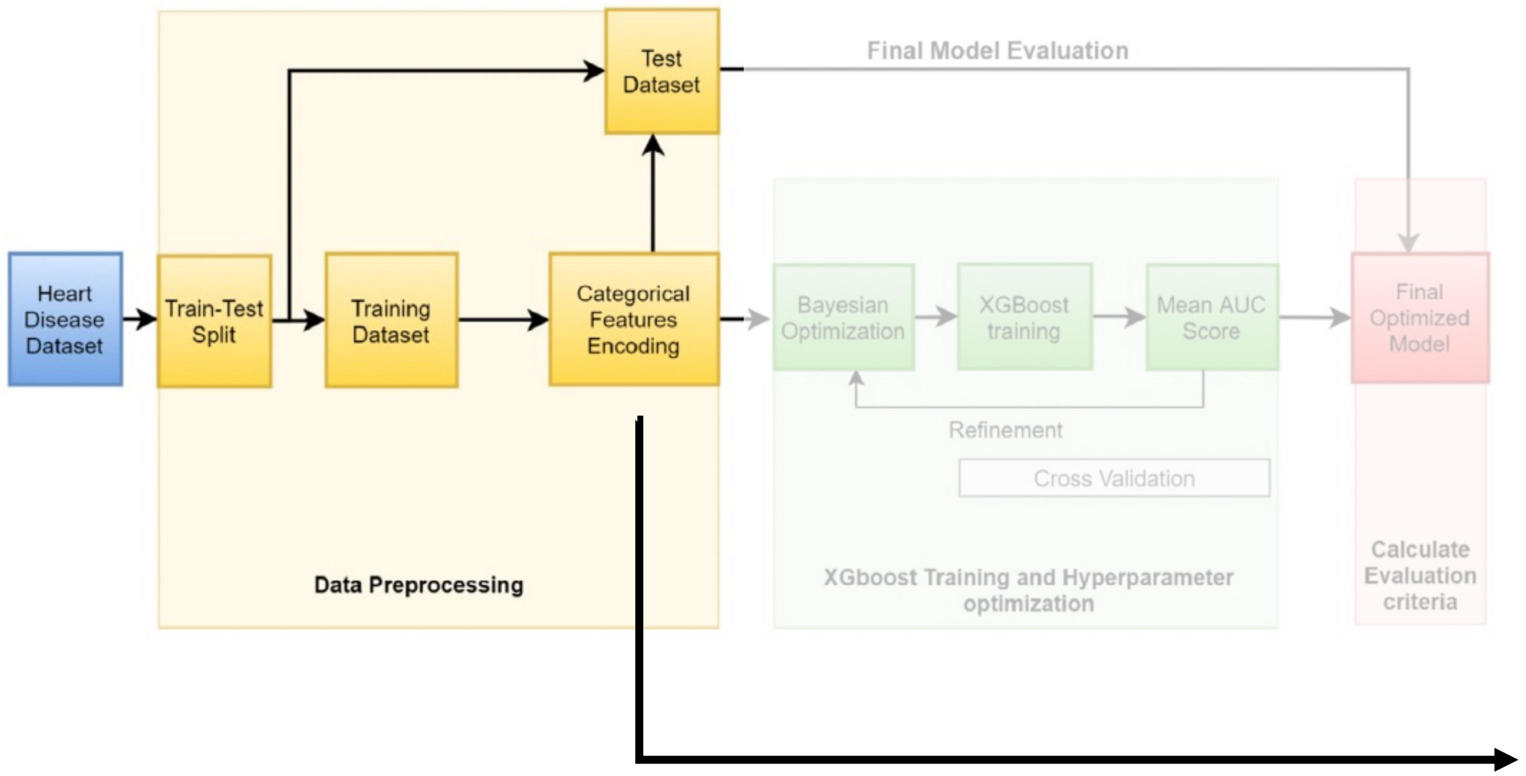


We need to split the data into a training set and a testing set so that we can build and evaluate our model. The authors use an 80/20 split.

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.

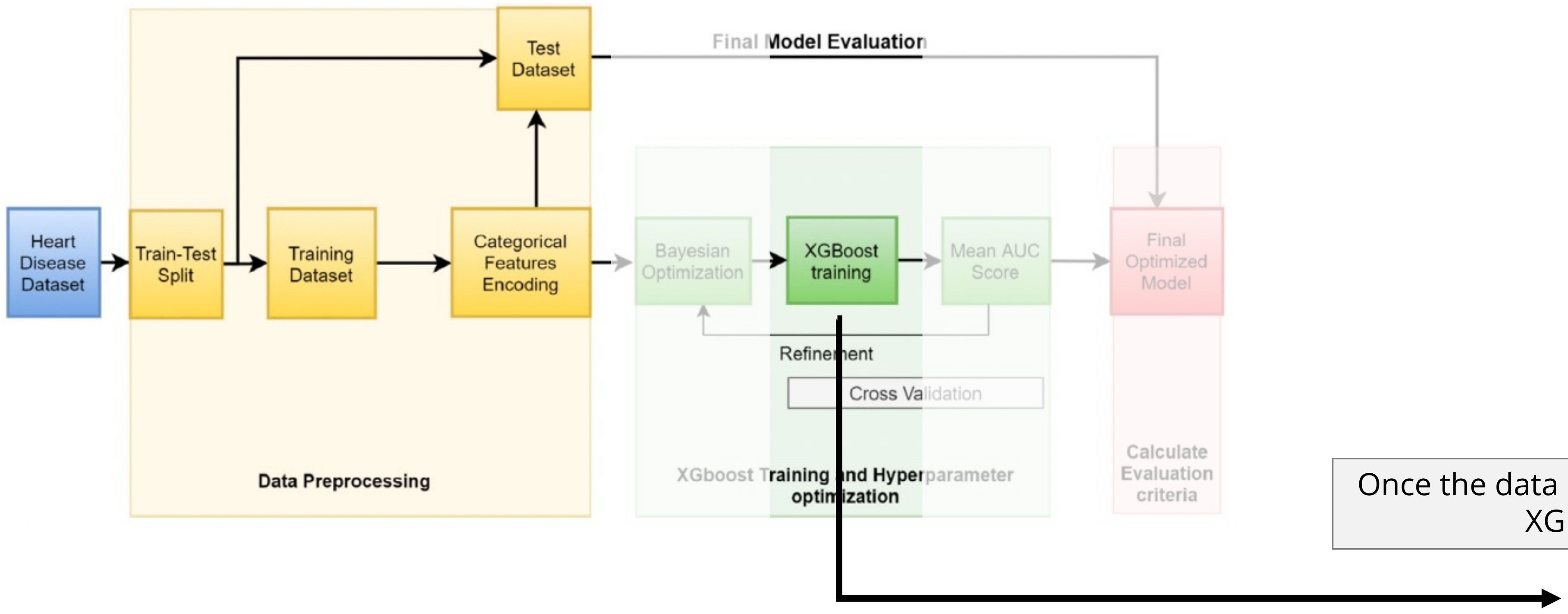
Step 3: Encode the Categorical Features



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).

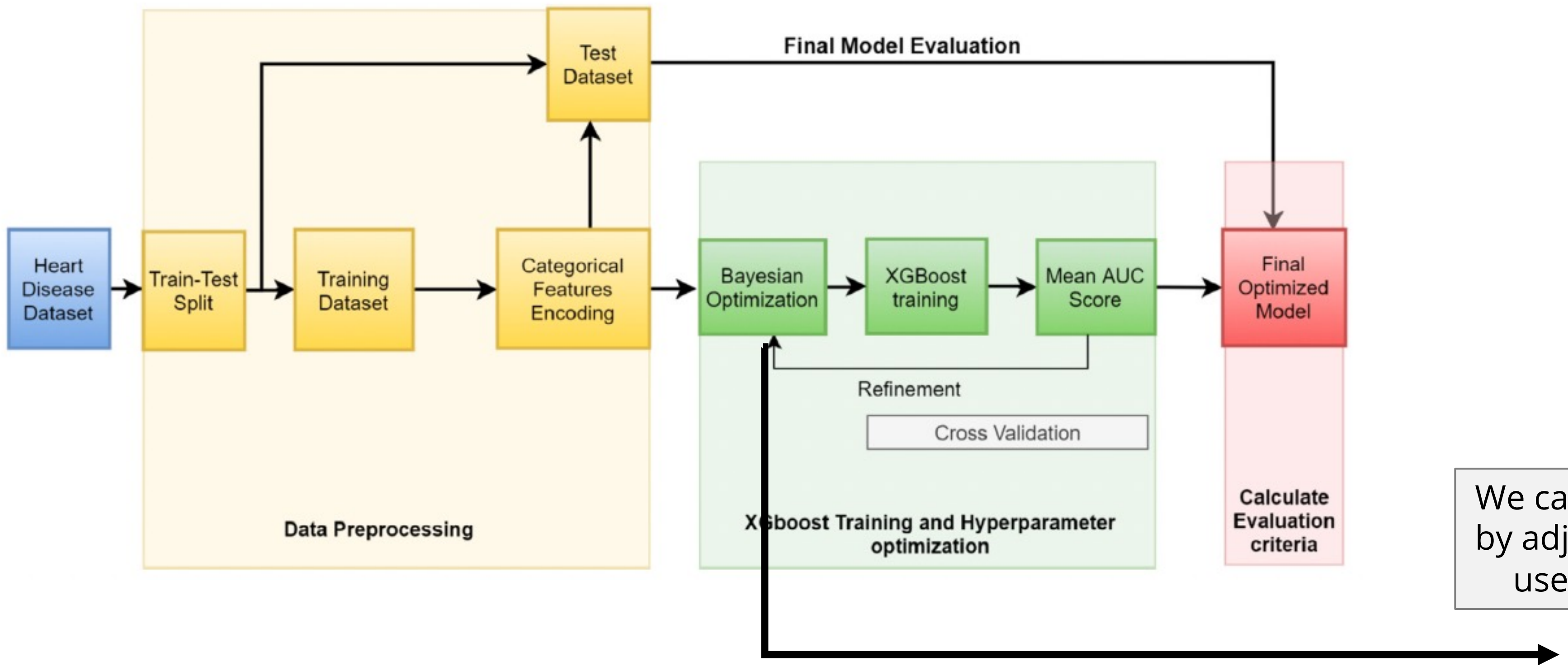
In the case of the heart data, we also need to transform the response variable (`num`) to binary.

Step 4: Build the Classifier



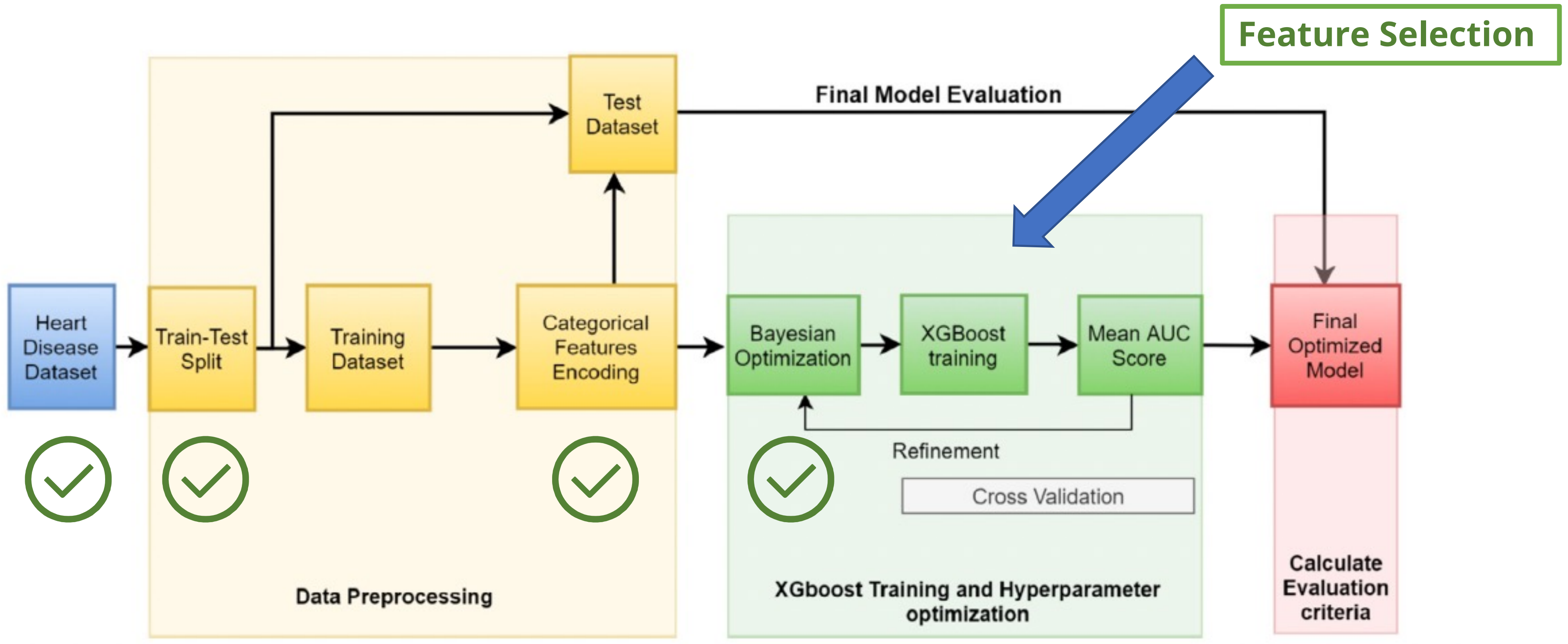
Once the data is prepared we can train an XGBoost classifier.

Step 4: Tune Hyperparameters



We can improve the classifier performance by adjusting model hyperparameters. We'll use a Bayesian optimization approach

Additional Step: Feature Selection



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)

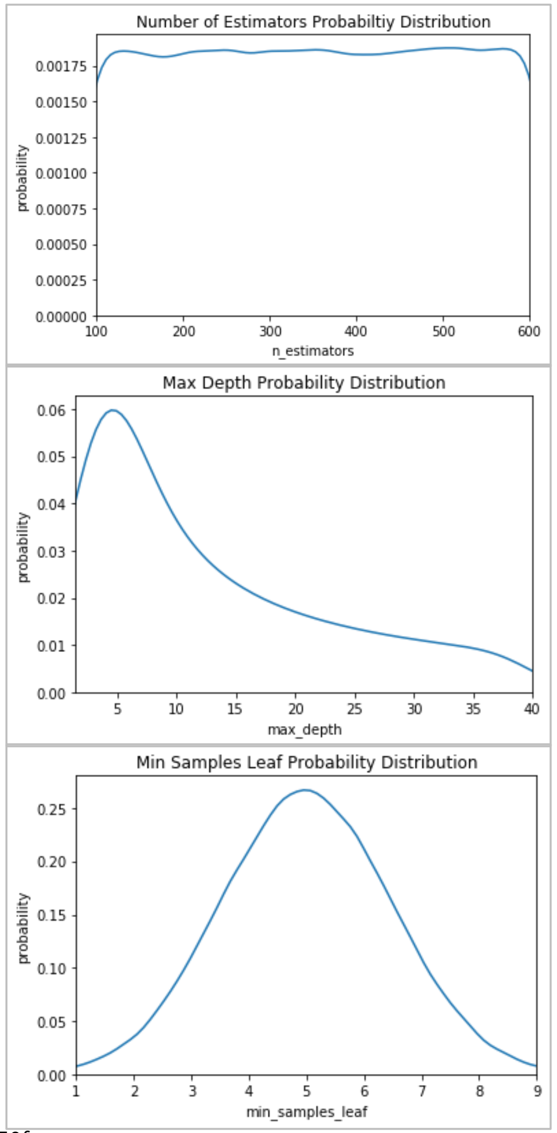
Hyperparameter Tuning Methods

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

XGBoost Hyperparameters

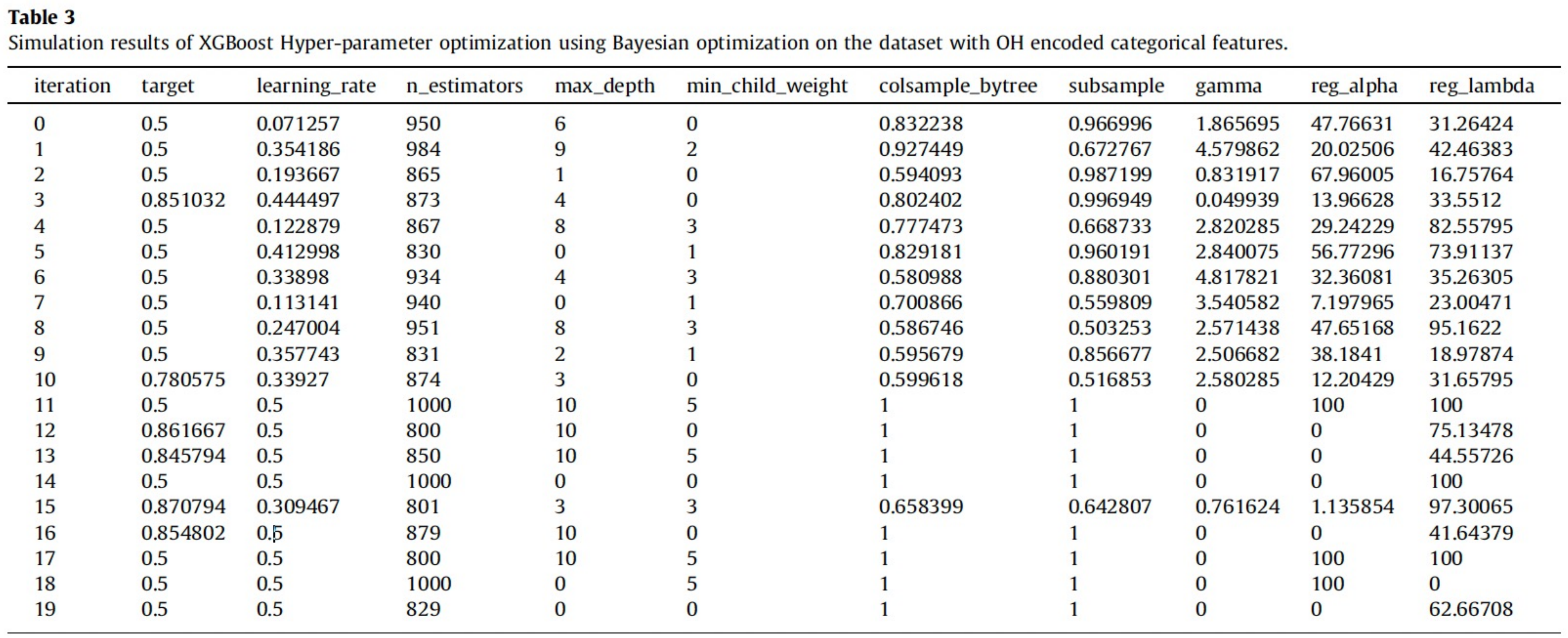


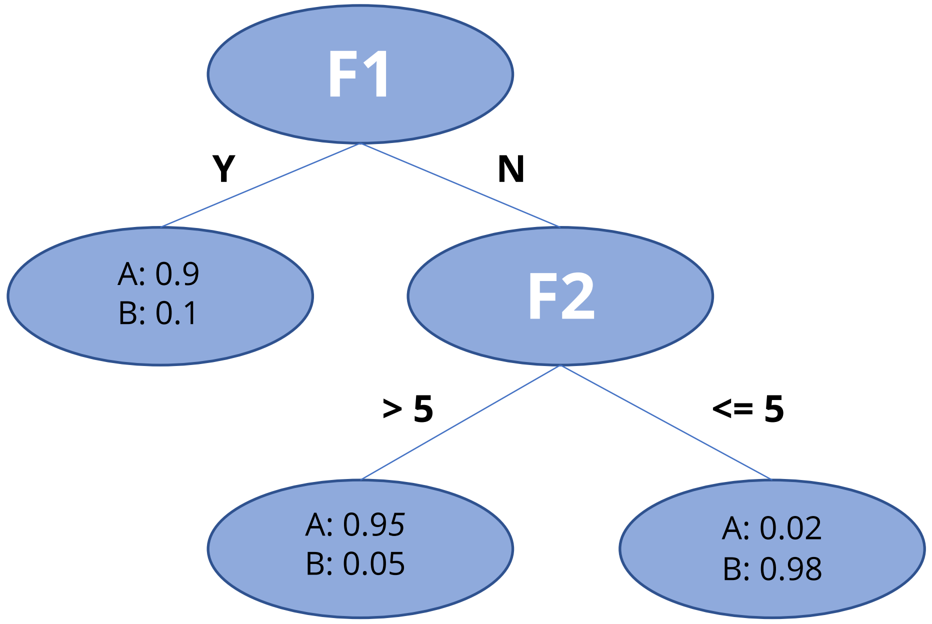
Table 3
Simulation results of XGBoost Hyper-parameter optimization using Bayesian optimization on the dataset with OH encoded categorical features.

iteration	target	learning_rate	n_estimators	max_depth	min_child_weight	colsample_bytree	subsample	gamma	reg_alpha	reg_lambda
0	0.5	0.071257	950	6	0	0.832238	0.966996	1.865695	47.76631	31.26424
1	0.5	0.354186	984	9	2	0.927449	0.672767	4.579862	20.02506	42.46383
2	0.5	0.193667	865	1	0	0.594093	0.987199	0.831917	67.96005	16.75764
3	0.851032	0.444497	873	4	0	0.802402	0.996949	0.049939	13.96628	33.5512
4	0.5	0.122879	867	8	3	0.777473	0.668733	2.820285	29.24229	82.55795
5	0.5	0.412998	830	0	1	0.829181	0.960191	2.840075	56.77296	73.91137
6	0.5	0.33898	934	4	3	0.580988	0.880301	4.817821	32.36081	35.26305
7	0.5	0.113141	940	0	1	0.700866	0.559809	3.540582	7.197965	23.00471
8	0.5	0.247004	951	8	3	0.586746	0.503253	2.571438	47.65168	95.1622
9	0.5	0.357743	831	2	1	0.595679	0.856677	2.506682	38.1841	18.97874
10	0.780575	0.33927	874	3	0	0.599618	0.516853	2.580285	12.20429	31.65795
11	0.5	0.5	1000	10	5	1	1	0	100	100
12	0.861667	0.5	800	10	0	1	1	0	0	75.13478
13	0.845794	0.5	850	10	5	1	1	0	0	44.55726
14	0.5	0.5	1000	0	0	1	1	0	0	100
15	0.870794	0.309467	801	3	3	0.658399	0.642807	0.761624	1.135854	97.30065
16	0.854802	0.5	879	10	0	1	1	0	0	41.64379
17	0.5	0.5	800	10	5	1	1	0	100	100
18	0.5	0.5	1000	0	5	1	1	0	100	0
19	0.5	0.5	829	0	0	1	1	0	0	62.66708

Feature Importance

Feature importance is a measure of how substantial the influence of a given feature /predictor is on the response variable. That seems straightforward enough. But how do we measure 'influence'?

Consider a (very) simple tree construction:



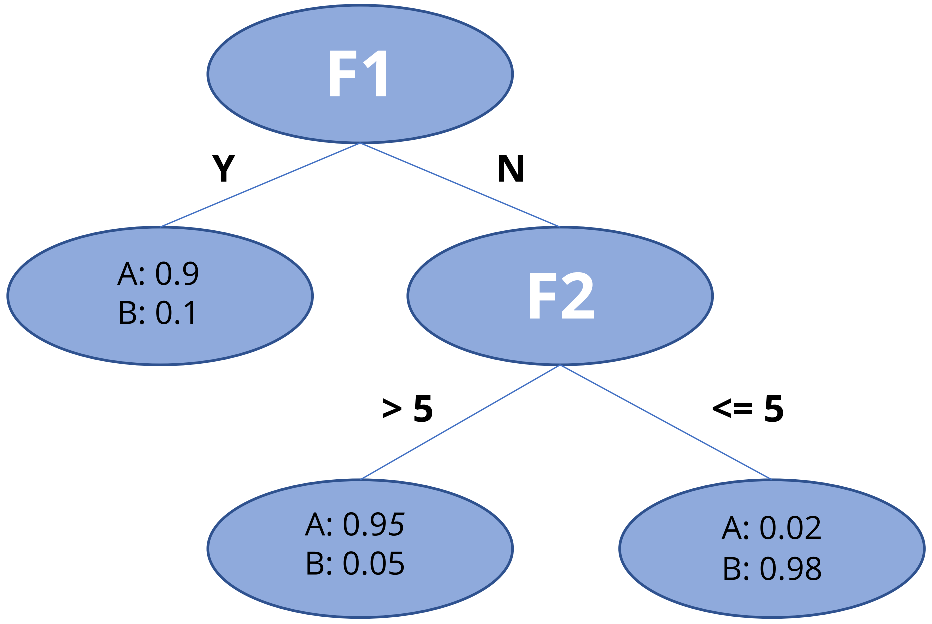
Importance is calculated for a single decision tree by the amount that each attribute split point improves the performance measure, weighted by the number of observations the node is responsible for.

At each split, we want to use the feature that results in the greatest **node purity** of the resulting branches. A node that contains a 50/50 split between two classes would be 100% impure. A node that contains all of one class would be 100% pure.

Feature Importance

Feature importance is a measure of how substantial the influence of a given feature /predictor is on the response variable. That seems straightforward enough. But how do we measure 'influence'?

Consider a (very) simple tree construction:



Thus, for a single tree we'll iterate through the nodes of the tree and calculate the weighted reduction in node purity resulting from the split. We'll attribute that reduction to the feature used for the split. When done, we'll divide those values by the total weight of the data (n observations).

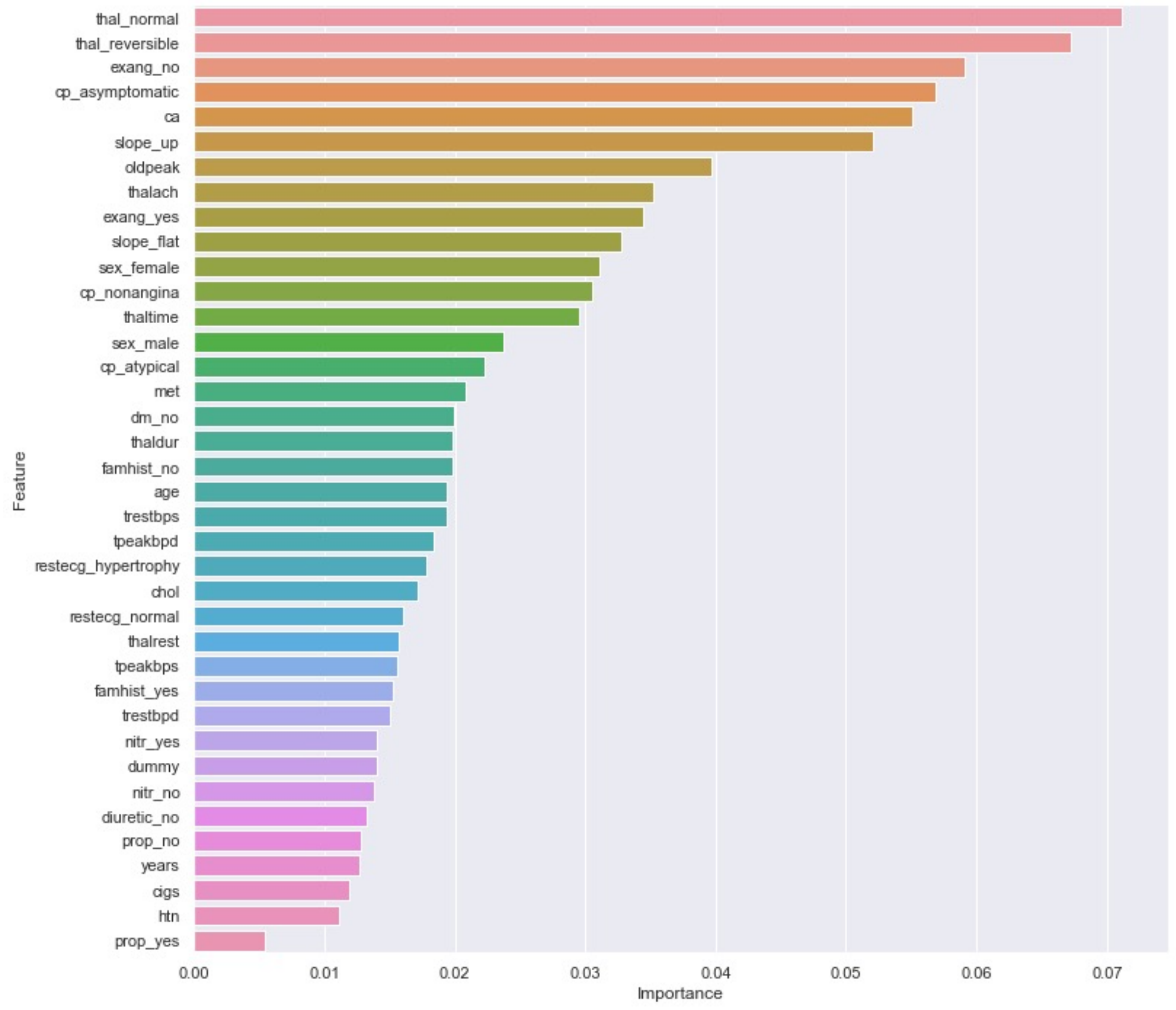
Now, in XGBoost we have many trees to consider. To get the importance of each feature across all trees, we simply average the importance values calculated for each tree.

Feature Selection

We can extract the feature importance values from the trained XGBoost model, sort them, and plot them to visualize which features had the most substantial impact.

Here, we're plotting an importance metric called 'gain'. Gain implies the relative contribution of a given feature to the model and is calculated by taking each feature's contribution for each tree in the model.

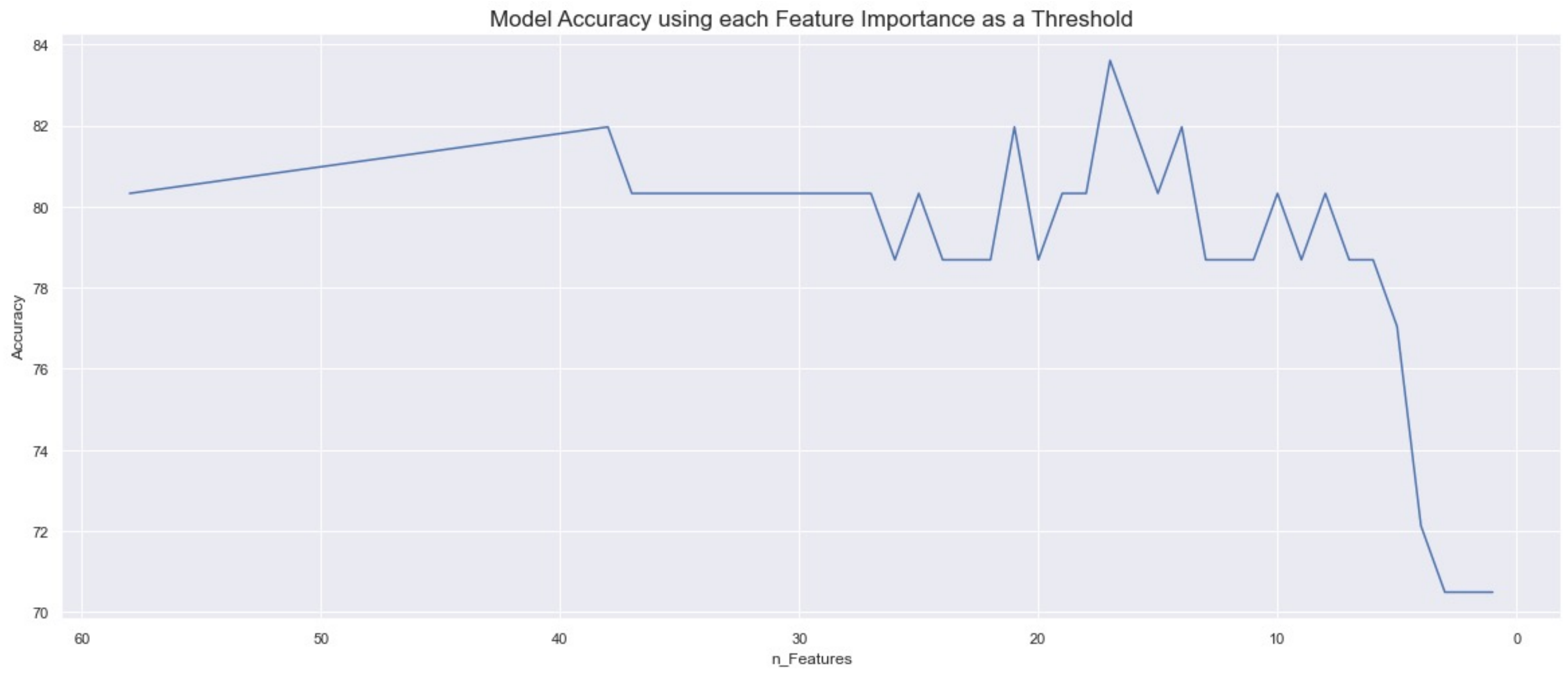
$$Gain = \frac{1}{2}\left[\underbrace{\frac{G_L^2}{H_L+\lambda}}_{\text{Score on new left leaf}} + \underbrace{\frac{G_R^2}{H_R+\lambda}}_{\text{Score on new right leaf}} - \underbrace{\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}}_{\text{Score on original leaf}}\right] - \underbrace{\gamma}_{\text{Regularization}}$$



Feature Selection

We can refit our model iteratively using a decreasing number of features (based on an increasing threshold for importance/gain) and see how the model accuracy changes.

Once we do this, we can select a set of features that minimizes model complexity and maximizes accuracy.


Model Accuracy using each Feature Importance as a Threshold
Accuracy
n_Features