StructureBoost

Aug 28, 2020

Contents

CHAPTER 1

Overview

StructureBoost is a machine learning package that implements Gradient Boosted Trees. What sets StructureBoost apart is its ability to exploit the **structure** of categorical variables.

1.1 Getting Started with StructureBoost

1.1.1 Installation

Installing StructureBoost is straightforward using pip:

```
pip install structureboost
```

You can also access the source code on Github

1.1.2 Feature Configuration

StructureBoost allows some options to be configured at a feature-specific level. These options are contained in a dictionary object where the keys of the dictionary contain the names of the features for the model. An example configuration dictionary is below:

```
{'county': {'feature_type': 'categorical_str',
 'graph': <graphs.graph_undirected at 0x13cf2d080>,
 'split_method': 'span_tree',
 'num_span_trees': 1},
 'month': {'feature_type': 'numerical', 'max_splits_to_search': 25}}
```

In this example, there are two features: ‘county’ and ‘month’. The feature ‘county’ has feature type ‘categorical_str’, with ‘split_method’ set to ‘span_tree’ and ‘num_span_trees’ set to 1. It also has a ‘graph’ associated with it. The feature ‘month’ is of feature_type ‘numerical’ and has the parameter ‘max_splits_to_search’ set to 25.

1.1.3 Feature Configuration Parameters

This page will walk through some of the parameters associated with each feature.

Feature Type: The most important aspect of a feature that must be defined is its type. StructureBoost admits the following values for the *feature_type*:

- **numerical**: This is a standard numeric variable. Missing values are permitted.
- **categorical_str**: A Categorical variable, where the values are of type *str*. Missing values are permitted. This is the more flexible categorical option, though it can be slightly slower than *categorical_int*. A *graph* must be provided to indicate the structure of the variable. If the structure is not known, we recommend using a complete graph (all values are pairwise adjacent).
- **categorical_int**: A Categorical variable where the different values are non-negative integers. Missing values are *not* permitted (primarily due to fact that numpy's nan does not have an integer option). This will be slightly faster than categorical_int, so in some cases it may be worth re-coding a string valued variable into integers. A *graph* must be provided to indicate the structure of the variable. If the structure is not known, we recommend using a complete graph (all values are pairwise adjacent).
- **graphical_voronoi**: This is a more complicated type which converts multiple numerical variables into a single categorical feature, with the structure determined by the Voronoi graph. See this example for details.

Depending on the feature_type, there will be further parameters that must be specified:

- For *numerical* feature type:
  - *max_splits_to_search*: This parameter must be specified to determine the maximum number of splits that should be checked. This must be set to a positive integer or np.Inf (to search all splits). Results are generally better when keeping this to a smaller value (between 10 and 50). In addition to being faster, it serves to regularize the model. Splits are chosen randomly.
- For *categorical_int* and *categorical_str* feature types:
  - *split_method*: Options are *span_tree*, *contraction*, and *one_hot*. This determines how to arrive at the space of splits to search for a categorical variable. We describe them below:
    - *span_tree*: This is the recommended method. A graph must be provided. A number of spanning trees of the graph will be chosen uniformly from the space of all spanning trees. Then, the algorithm will consider the splits induced by removing each possible edge from the spanning tree. This method requires the specification of *num_span_trees* - the number of spanning trees to check for this feature at each node. The recommended value of *num_span_trees* is 1.
    - *contraction*: This method also requires a graph. The graph is reduced in size by randomly contracting edges until it reaches a size <= the specified parameter *contraction_size*. At that point, it enumerates all possible binary splits into two connected components of the contracted graph. A number *max_splits_to_search* of these a randomly selected to be evaluated. Enumerating all possible (allowable) binary splits is computationally intensive, therefore it is recommended to choose a contraction_size around 9. Similarly, we recommend setting *max_splits_to_search* in the range of 10 to 50. If the graph is a cycle or similarly sparse, then larger numbers can be chosen. To enumerate all allowable binary splits, choose the *contraction_size* to be larger than the size of the graph. To evaluate all enumerated splits, choose *max_splits_to_search*
    - *one_hot*: This method will search all "one-hot" splits – that is, splits where one value goes left and the rest go right. This is not a recommended method, but provided for comparison purposes. If this method is chosen, a graph is not required to be specified.

1.1.4 Model Configuration Parameters

In addition to the feature-specific configurations, there are additional parameters set at the model level. These are similar to those available in other boosting packages.

- **mode**: This can be 'classification' or 'regression'. StructureBoost uses a single class to handle both situations.
- **num_trees**: The number of trees to build. We recommend using an eval_set with early stopping so that the number of trees built is learned dynamically. When using that option, you can set num_trees to a large value. However, one can also specify a set number of trees.
- **max_depth**: The maximum depth to build the trees. The larger the number, the more likely you are to overfit, and the less likely to underfit. Default is 3.
- **learning_rate**: The "step size" to use when adding each tree. We recommend erring on the side of having smaller steps and more trees (and using early stopping with an eval_set to optimize model size). StructureBoost works particularly well under these conditions since it will have multiple opportunities to visit the space of possible splits.
- **subsample**: How much of the training data to use at each tree. Will be interpreted as a number of rows if given an integer >1 and a percentage of the data if given a float between 0 and 1. Rows can be chosen with or without replacement, depending on the value of *replace*.
- **replace**: If True, the data set for each tree will be chosen with replacement (as in the bootstrap). If False, the rows for each tree will chosen without replacement.
- **loss_fn**: By default, we will use log loss (aka entropy, categorical cross-entropy, maximum likelihood) for classification and mean squared error for regression. To specify a custom loss_fn, one can pass a tuple containing two (vectorized) functions that return the first and second derivatives of the loss_fn.
- **feat_sample_by_tree**: The fraction (or number) of features to sample from at each tree. Will be interpreted as a number of features for integers>1 and a fraction for floats between 0 and 1.
- **feat_sample_by_node**: The fraction (or number) of features to sample from at each node. Will be interpreted as a number of features for integers>1 and a fraction for floats between 0 and 1. Effect is cumulative with feat_sample_by_tree - chooses a subset relative to the size of the subset passed to it for each tree.
- **gamma**: Regularization parameter as in XGBoost. If the best split results in a gain less than gamma, it will not be executed.
- **reg_lambda**: L2 Regularization parameter as in XGBoost. Serves to "shrink" the size of the value at each leaf.
- **na_unseen_action**: How to choose which way Missing Values should be sent in the tree, if there are no Missing Values at that node at the time of choosing a split. Default is "weighted_random", which randomly fixes the direction in proportion to the number of data points that went into each direction. Alternative is "random", which chooses a direction with equal probability regardless of how many data points went each direction.
- **min_sample_split**: How many data points a node must have to consider splitting it further. Default is 2.

1.1.5 StructureBoost Class

To get started with StructureBoost, read through the docs, or just dive into some *Examples*

1.2 Categorical Structure

What do we mean by the *structure* of a categorical variable?

The standard approaches to dealing with categorical variables are "one-hot encoding" (which assumes the possible values are all different from one another, but with no particular similarities or differences) or "integer-encoding" (which assumes the values have an ordinal or linear structure).

However, in many cases the underlying structure of the values is not a strict ordering.

Here are a few examples:

- The contiguous 48 states in the USA have a structure based on which states border which others.
- The 12 months in the year have a circular structure: January is "next to" December in the same way that July is "next to" August.
- A survey question has a set of possible response given by "Poor", "Fair", "Good", "Excellent", "Not Applicable", "I prefer not to answer". The first four answers may have a linear structure, with the other two responses incomparable.
- The outcome variable of CIFAR-10 has the following possibilities: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. 5 of them are vehicles and 5 are animals. Of the 5 animals, 3 are mammals. This yields an ontological structure: a car is more similar to a truck than either is to a frog.

1.3 StructureBoost FAQ

1. **What is the** *feature_configs*?: StructureBoost uses a configuration dictionary to specify many parameters. For each feature, we must configure how to determine which splits to evaluate. For categorical variables, this can particularly detailed. An advantage of this is that StructureBoost allows a level of fine-tuned control not typically available in other packages. To reduce the overhead involved, StructureBoost provides tools for quickly defining an "start" configuration that can be easily modified. Furthermore, we can validate the configuration to reduce any unexpected errors or results. For more detailed documentation on the various settings visit *Feature Configuration*. Or better yet, go through this example notebook.

1.4 Examples

The best way to learn how to use StructureBoost is by working through examples. As such, we have created numerous Jupyter notebooks on a variety of data sets to illustrate the capabilities.

The full set of examples can be found here. The descriptions are below.

1. CA_County_Classification: A straightforward example of how to configure and use StructureBoost for a classification problem. Also contains a comparison to other boosting packages.
2. CA_County_Regression: Like the previous notebook, but for a regression example.
3. CA_County_Deep_Dive: A lengthier exploration of the CA County weather data, highlighting other ways that Categorical Structure can be extremely important.
4. MoCap_Spatial: An example using the **graphical_voronoi** feature type on 3d sensor data detecting hand postures.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search