mlpack / **mlpack**

Branch: **master** **mlpack** / src / mlpack / methods / decision_stump / **decision_stump_main.cpp** **Find file** **Copy path**

kaushik-rohit switch to serialize() and use boost::serialization 494a52f on 5 Oct

4 contributors

197 lines (173 sloc) 7.19 KB

```cpp
/**
 * @file decision_stump_main.cpp
 * @author Udit Saxena
 *
 * Main executable for the decision stump.
 *
 * mlpack is free software; you may redistribute it and/or modify it under the
 * terms of the 3-clause BSD license.  You should have received a copy of the
 * 3-clause BSD license along with mlpack.  If not, see
 * http://www.opensource.org/licenses/BSD-3-Clause for more information.
 */
#include <mlpack/prereqs.hpp>
#include <mlpack/core/util/cli.hpp>
#include <mlpack/core/data/normalize_labels.hpp>
#include <mlpack/core/util/mlpack_main.hpp>
#include "decision_stump.hpp"

using namespace mlpack;
using namespace mlpack::decision_stump;
using namespace std;
using namespace arma;

```

```
PROGRAM_INFO("Decision Stump",
    "This program implements a decision stump, which is a single-level decision"
    " tree.  The decision stump will split on one dimension of the input data, "
    "and will split into multiple buckets.  The dimension and bins are selected"
    " by maximizing the information gain of the split.  Optionally, the minimum"
    " number of training points in each bin can be specified with the " +
    PRINT_PARAM_STRING("bucket_size") + " parameter."
    "\n\n"
    "The decision stump is parameterized by a splitting dimension and a vector "
    "of values that denote the splitting values of each bin."
    "\n\n"
    "This program enables several applications: a decision tree may be trained "
    "or loaded, and then that decision tree may be used to classify a given set"
    " of test points.  The decision tree may also be saved to a file for later "
    "usage."
    "\n\n"
    "To train a decision stump, training data should be passed with the " +
    PRINT_PARAM_STRING("training") + " parameter, and their corresponding "
    "labels should be passed with the " + PRINT_PARAM_STRING("labels") + " "
    "option.  Optionally, if " + PRINT_PARAM_STRING("labels") + " is not "
    "specified, the labels are assumed to be the last dimension of the "
    "training dataset.  The " + PRINT_PARAM_STRING("bucket_size") + " "
    "parameter controls the minimum number of training points in each decision "
    "stump bucket."
    "\n\n"
    "For classifying a test set, a decision stump may be loaded with the " +
    PRINT_PARAM_STRING("input_model") + " parameter (useful for the situation "
    "where a stump has already been trained), and a test set may be specified "
    "with the " + PRINT_PARAM_STRING("test") + " parameter.  The predicted "
    "labels can be saved with the " + PRINT_PARAM_STRING("predictions") + " "
    "output parameter."
    "\n\n"
    "Because decision stumps are trained in batch, retraining does not make "
    "sense and thus it is not possible to pass both " +
    PRINT_PARAM_STRING("training") + " and " +
    PRINT_PARAM_STRING("input_model") + "; instead, simply build a new "
    "decision stump with the training data."
    "\n\n"
    "After training, a decision stump can be saved with the " +
    PRINT_PARAM_STRING("output_model") + " output parameter.  That stump may "
    "later be re-used in subsequent calls to this program (or others).");

// Datasets we might load.
PARAM_MATRIX_IN("training", "The dataset to train on.", "t");
PARAM_UROW_IN("labels", "Labels for the training set. If not specified, the "
    "labels are assumed to be the last row of the training data.", "l");
PARAM_MATRIX_IN("test", "A dataset to calculate predictions for.", "T");

// Output.
PARAM_UROW_OUT("predictions", "The output matrix that will hold the "
    "predicted labels for the test set.", "p");

/**
 * This is the structure that actually saves to disk.  We have to save the
 * label mappings, too, otherwise everything we load at test time in a future
 * run will end up being borked.
 */
struct DSModel
{
  //! The mappings.
  arma::Col<size_t> mappings;
  //! The stump.
  DecisionStump<> stump;

  //! Serialize the model.
  template<typename Archive>
  void serialize(Archive& ar, const unsigned int /* version */)
  {
    ar & BOOST_SERIALIZATION_NVP(mappings);
    ar & BOOST_SERIALIZATION_NVP(stump);
  }
};

// We may load or save a model.
PARAM_MODEL_IN(DSModel, "input_model", "Decision stump model to "
    "load.", "m");
PARAM_MODEL_OUT(DSModel, "output_model", "Output decision stump model to save.",
    "M");

PARAM_INT_IN("bucket_size", "The minimum number of training points in each "
    "decision stump bucket.", "b", 6);

void mlpackMain()
{
  // Check that the parameters are reasonable.
  if (CLI::HasParam("training") && CLI::HasParam("input_model"))
  {
    Log::Fatal << "Both --training_file and --input_model_file are specified, "
        << "but a trained model cannot be retrained.  Only one of these options"
        << " may be specified." << endl;
  }

  if (!CLI::HasParam("training") && !CLI::HasParam("input_model"))
  {
    Log::Fatal << "Neither --training_file nor --input_model_file are given; "
        << "one must be specified." << endl;
  }

  if (!CLI::HasParam("output_model") && !CLI::HasParam("predictions"))
  {
    Log::Warn << "Neither --output_model_file nor --predictions_file are "
        << "specified; no results will be saved!" << endl;
  }

  // We must either load a model, or train a new stump.
  DSModel model;
  if (CLI::HasParam("training"))
  {
    mat trainingData = std::move(CLI::GetParam<mat>("training"));

    // Load labels, if necessary.
    Row<size_t> labelsIn;
    if (CLI::HasParam("labels"))
    {
      labelsIn = std::move(CLI::GetParam<Row<size_t>>("labels"));
    }
    else
    {
      // Extract the labels as the last
      Log::Info << "Using the last dimension of training set as labels."
          << endl;

      labelsIn = arma::conv_to<arma::Row<size_t>>::from(
          trainingData.row(trainingData.n_rows - 1));
      trainingData.shed_row(trainingData.n_rows - 1);
    }

    // Normalize the labels.
    Row<size_t> labels;
    data::NormalizeLabels(labelsIn, labels, model.mappings);

    const size_t bucketSize = CLI::GetParam<int>("bucket_size");
    const size_t classes = labels.max() + 1;

    Timer::Start("training");
    model.stump.Train(trainingData, labels, classes, bucketSize);
    Timer::Stop("training");
  }
  else
  {
    model = std::move(CLI::GetParam<DSModel>("input_model"));
  }

  // Now, do we need to do any testing?
  if (CLI::HasParam("test"))
  {
    // Load the test file.
    mat testingData = std::move(CLI::GetParam<arma::mat>("test"));

    if (testingData.n_rows <= model.stump.SplitDimension())
      Log::Fatal << "Test data dimensionality (" << testingData.n_rows << ") "
          << "is too low; the trained stump requires at least "
          << model.stump.SplitDimension() << " dimensions!" << endl;

    Row<size_t> predictedLabels(testingData.n_cols);
    Timer::Start("testing");
    model.stump.Classify(testingData, predictedLabels);
    Timer::Stop("testing");

    // Denormalize predicted labels, if we want to save them.
    if (CLI::HasParam("predictions"))
    {
      Row<size_t> actualLabels;
      data::RevertLabels(predictedLabels, model.mappings, actualLabels);

      // Save the predicted labels as output.
      CLI::GetParam<Row<size_t>>("predictions") = std::move(actualLabels);
    }
  }

  // Save the model, if desired.
  if (CLI::HasParam("output_model"))
    CLI::GetParam<DSModel>("output_model") = std::move(model);
}
```