基于决策回归树的缺失值插补技术 C++实现

1. 第三方库

1) GLFW 获取平台鼠标，键盘输入事件

2) imGUI 绘制图形窗口

2. 决策回归树

决策回归树（Decision Regression Tree）是一种常用的机器学习算法，用于预测连续型变量的取值。它基于树结构来对数据进行建模和预测，通过将数据集划分为不同的区域，并在每个区域内预测一个常数值来实现回归任务。

2.1 决策回归树原理

决策树回归通过构建一颗树结构来对数据进行建模和预测。树的每个内部节点表示一个属性/特征，每个叶节点表示一个输出值。决策树的构建过程是一个递归的过程，它通过选择最佳的属性/特征来进行数据划分，使得划分后子集的输出值尽可能接近真实值。

决策树的构建过程主要包括以下几个步骤：

1. 选择最佳划分属性/特征：通过某种指标（如信息增益、基尼系数）选择最佳的属性/特征来进行数据划分。

2. 划分数据集：根据选择的属性/特征将数据集划分为多个子集。

3. 递归构建子树：对每个子集递归地应用上述步骤，直到满足停止条件（如达到最大深度、节点中样本数量小于阈值等）为止。

2.2 决策回归树模型步骤

步骤 1：准备数据集。准备包含输入特征和对应输出值的数据集。

步骤 2：选择划分属性。根据某种指标（如均方误差、平方损失）选择最佳的划分属性/特征。

步骤 3：划分数据集。根据选择的划分属性将数据集划分为多个子集。

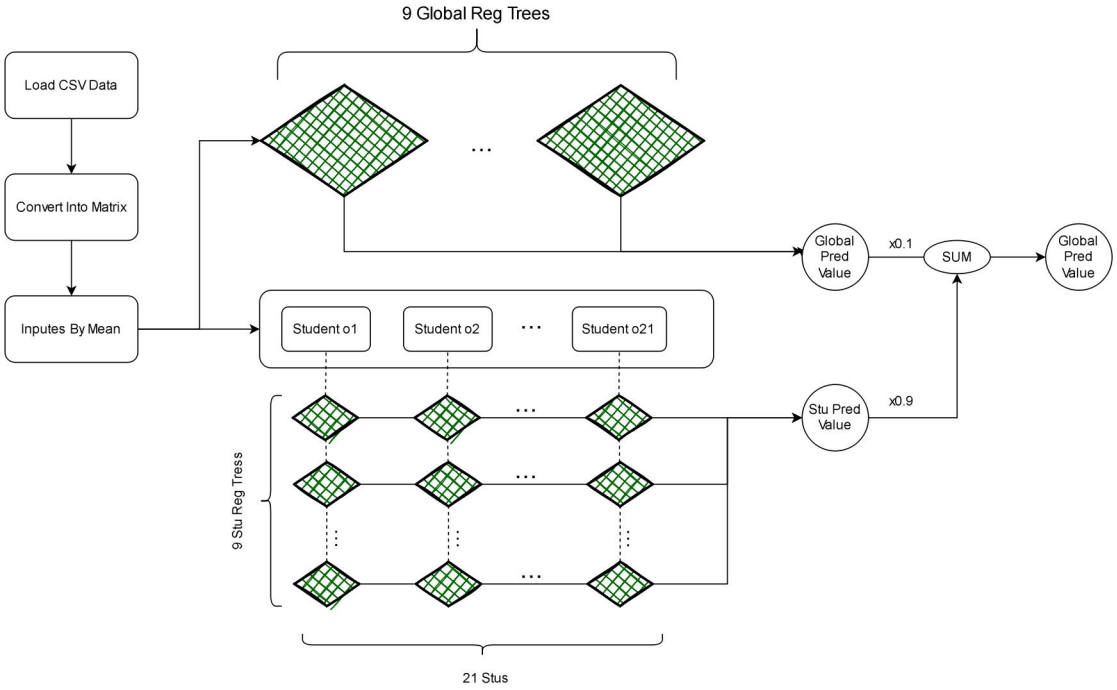
步骤 4：递归构建子树。对每个子集递归地应用上述步骤，直到满足停止条件。

步骤 5：生成决策树。构建完整的决策树结构。

2.3 决策回归树的优缺点

优点	易于理解和解释：决策树可以直观地呈现，易于理解和解释，可以帮助分析人员做出决策。 能够处理非线性关系：决策树可以处理非线性关系，不需要对数据进行线性假设。 对数据的缺失值不敏感：决策树在构建过程中可以处理数据的缺失值。
缺点	容易过拟合：决策树容易过拟合训练数据，特别是在数据量较小或树的深度较大时。 不稳定性：数据的小变化可能导致树结构的显著改变，使得决策树不够稳定。 难以处理连续性特征：决策树在处理连续性特征时，需要对其进行离散化处理，可能会损失一部分信息。

3. 算法设计



如上图所示，我们认为学生的成绩与班级是有关系的，例如高中的普通班、精英班之类的，所以我们训练两种决策树 1. 全局回归树（班级树）、2. 个人回归树（学生树）。并在最后，我们将这两种树的预测结果进行加权求和。

回归树的训练策略是：1. 使用学生个人各科平均值插补学生的缺失数据。2. 使用插补后的各科数据训练 9 棵全局回归树，即使用其余 8 个科目的成绩预测缺失的科目成绩（共 9 个科目）。3. 生成单个学生的缺失科目索引和缺失考试索引。4.根据该学生缺失科目索引来训练该学生的学生树（若学生在所有次月考

中，只有 3 个科目缺失过成绩，那么该学生的学生树就有 3 棵，即分别用于预测缺失科目的成绩）。5. 分别用班级树和学生树进行预测出缺失成绩。6. 将班级树与学生树的预测结果加权求和得最终预测结果。

4. 算法实现

4.1. 从 CSV 文件读取数据

```
std::vector<Student> load_data_from_csv(const std::string &path) {
  std::vector<Student> stus;
  std::ifstream file(path);
  std::string line;

  // Skip the header line
  // stu, tm, x1, x2, ..., x9
  std::getline(file, line);

  while (std::getline(file, line)) {
    std::map<std::string, std::vector<float>> scores;
    std::string stu_name;
    std::string stu_tm;
    std::string value;
    std::vector<float> tm_scores;
    std::istringstream iss(line);
    // stu
    std::getline(iss, stu_name, ',');
    // tm
    std::getline(iss, stu_tm, ',');
    // x1, x2, ..., x9
    for (size_t i = 0; i < 9; i++) {
      std::getline(iss, value, ',');
      tm_scores.push_back(std::stof(value));
    }
    scores.insert(std::make_pair(stu_tm, tm_scores));
    for (size_t i = 0; i < 5; i++) {
      tm_scores.clear();
      std::getline(file, line);
      std::istringstream ss(line);
      // stu
      std::getline(ss, stu_name, ',');
      // tm
      std::getline(ss, stu_tm, ',');
      // x1, x2, ..., x9
      for (size_t i = 0; i < 9; i++) {
```

```
      std::getline(ss, value, ',');
      tm_scores.push_back(std::stof(value));
    }
    scores.insert(std::make_pair(stu_tm, tm_scores));
  }

  stus.push_back(Student(stu_name, scores));
 }

 return stus;
}
```

4.2. 生成缺失科目索引

```
std::vector<size_t>
gen_missing_feat_idxs(const std::vector<std::vector<float>> &X) {
 std::vector<size_t> idxs;
 size_t n_sample = X.size();
 size_t n_feat = X[0].size();

 for (auto i = 0; i < n_feat; i++) {
  for (auto j = 0; j < n_sample; j++) {
   if (std::abs(X[j][i] - (-1.0)) < 0.00001) {
    idxs.push_back(i);
    break;
   }
  }
 }

 return idxs;
}
```

4.3. 生成缺失考试索引

```
std::vector<size_t>
gen_missing_sample_idxs(const std::vector<std::vector<float>> &X) {
 std::vector<size_t> idxs;
 size_t n_sample = X.size();
 size_t n_feat = X[0].size();

 for (auto i = 0; i < n_sample; i++) {
  for (auto j = 0; j < n_feat; j++) {
   if (std::abs(X[i][j] - (-1.0)) < 0.00001) {
    idxs.push_back(i);
    break;
   }
```

```
    }
  }

  return idxs;
}
```

4.3. 均值插补

```
std::vector<std::vector<float>>
impute_missing_by_mean(const std::vector<std::vector<float>> &X) {
  std::vector<std::vector<float>> X_new = X;
  std::vector<float> avarage_feat;
  size_t n_sample = X.size();
  size_t n_feat = X[0].size();

  for (int i = 0; i < n_feat; i++) {
    float sum = 0.0;
    int n_valid = 0;
    for (int j = 0; j < n_sample; j++) {
      if (!(std::abs(X[j][i] - (-1.0)) < 0.00001)) {
        n_valid += 1;
        sum += X[j][i];
      }
    }
    avarage_feat.push_back(sum / n_valid);
  }

  for (int i = 0; i < n_feat; i++) {
    for (int j = 0; j < n_sample; j++) {
      if (std::abs(X_new[j][i] - (-1.0)) < 0.00001) {
        X_new[j][i] = avarage_feat[i];
      }
    }
  }

  return X_new;
}
```

4.4. 决策回归树（森林）插补

```
void impute_missing_values(std::vector<Student> &stus) {
  std::vector<std::vector<float>> X_all = ::stus_to_mat(stus);
  std::vector<std::vector<float>> X_all_meaned =
      ::impute_missing_by_mean(X_all);
  std::vector<RegressionTree> trees_all;
  size_t n_all_samples = X_all.size();
```

```
  size_t n_feat = X_all[0].size();

  Logger::info("trees_all is training...");

  for (auto i = 0; i < n_feat; i++) {
    auto X = X_all_meaned;
    std::vector<float> y;
    for (auto &x : X) {
      y.push_back(x[i]);
      x.erase(x.begin() + i);
    }
    auto tree = RegressionTree(5, 3);
    tree.train(X, y);
    trees_all.push_back(tree);
    Logger::info("trees_all[%d] has trained.", i);
  }

  Logger::info("trees_all has trained.");

  for (auto &stu : stus) {
    std::vector<std::vector<float>> X = ::stu_to_mat(stu);
    std::vector<std::vector<float>> X_output = X;
    std::vector<std::vector<float>> X_meaned = ::impute_missing_by_mean(X);
    std::vector<size_t> missing_sample_idxs = ::gen_missing_sample_idxs(X);
    std::vector<size_t> missing_feat_idxs = ::gen_missing_feat_idxs(X);
    size_t n_sample = X.size();

    Logger::debug("stu.name = %s", stu.m_name.c_str());
    Logger::debug("missing_feat_idxs.size = %lld",
                  missing_feat_idxs.size());
    Logger::debug("missing_sample_idxs.size = %lld",
                  missing_sample_idxs.size());

    for (auto i : missing_feat_idxs) {
      auto X_erased = X_meaned;
      auto y = std::vector<float>();
      auto p = std::vector<float>();
      auto tree = RegressionTree(3, 1);

      for (auto &x : X_erased) {
        y.push_back(x[i]);
        x.erase(x.begin() + i);
      }
```

```
    tree.train(X_erased, y);

    for (auto j = 0; j < n_sample; j++) {
      if (std::abs(X_output[j][i] - (-1.0)) < 0.00001) {
        X_output[j][i] = 0.9 * tree.predict(X_erased[j]) +
                         0.1 * trees_all[i].predict(X_erased[j]);
      }
    }

    stu.from_mat(X_output);
   }
  }
}
```

4.5. 决策回归树类

```
class RegressionTree {
public:
  struct Node {
    bool is_leaf;
    float value;
    int feat_idx;
    float threshold;
    Node *left;
    Node *right;

    Node()
        : is_leaf(false), value(0), feat_idx(-1), threshold(0), left(nullptr),
          right(nullptr) {}
  };

private:
  Node *m_root;
  int m_max_depth;
  int m_min_samples_split;

  Node *build_tree(const std::vector<std::vector<float>> &X,
                   const std::vector<float> &y, int depth) {
    if (X.size() <= m_min_samples_split || depth >= m_max_depth) {
      // 创建叶节点
      Node *leaf = new Node();
      leaf->is_leaf = true;
      leaf->value = calc_mean(y); // 均值
      return leaf;
```

```
  }

  // 寻找最佳分割
  int best_feat;
  float best_threshold;
  float best_mse = std::numeric_limits<float>::max();
  std::vector<std::vector<float>> best_left_x, best_right_x;
  std::vector<float> best_left_y, best_right_y;

  for (int feat = 0; feat < X[0].size(); ++feat) {
    // 按特征排序
    std::vector<std::pair<float, float>> feat_values;
    for (size_t i = 0; i < X.size(); ++i) {
      feat_values.push_back({X[i][feat], y[i]});
    }

    std::sort(feat_values.begin(), feat_values.end());

    // 尝试每个分割阈值
    for (size_t i = 1; i < feat_values.size(); ++i) {
      float threshold = (feat_values[i - 1].first + feat_values[i].first) / 2;
      std::vector<std::vector<float>> left_x, right_x;
      std::vector<float> left_y, right_y;

      for (size_t j = 0; j < X.size(); ++j) {
        if (X[j][feat] <= threshold) {
          left_x.push_back(X[j]);
          left_y.push_back(y[j]);
        } else {
          right_x.push_back(X[j]);
          right_y.push_back(y[j]);
        }
      }

      // 计算 MSE
      float mse = calc_mse(left_y, right_y);
      if (mse < best_mse) {
        best_mse = mse;
        best_feat = feat;
        best_threshold = threshold;
        best_left_x = left_x;
        best_left_y = left_y;
        best_right_x = right_x;
        best_right_y = right_y;
```

```
      }
    }
  }

  // 创建内部节点
  Node *node = new Node();
  node->is_leaf = false;
  node->feat_idx = best_feat;
  node->threshold = best_threshold;
  node->left = build_tree(best_left_x, best_left_y, depth + 1);
  node->right = build_tree(best_right_x, best_right_y, depth + 1);

  return node;
}

float predict_node(Node *node, const std::vector<float> &x) const {
  if (node->is_leaf) {
    return node->value;
  }

  if (x[node->feat_idx] <= node->threshold) {
    return predict_node(node->left, x);
  } else {
    return predict_node(node->right, x);
  }
}

float calc_mean(const std::vector<float> &y) {
  float sum = 0;
  for (float value : y) {
    sum += value;
  }
  return sum / y.size();
}

float calc_mse(const std::vector<float> &leftY,
               const std::vector<float> &rightY) {
  float mse = 0;

  // 左子树的 MSE
  float meanLeft = calc_mean(leftY);
  for (float value : leftY) {
    mse += (value - meanLeft) * (value - meanLeft);
  }
```

```
    // 右子树的 MSE
    float meanRight = calc_mean(rightY);
    for (float value : rightY) {
      mse += (value - meanRight) * (value - meanRight);
    }

    return mse;
  }

public:
  RegressionTree(size_t max_depth = 10, size_t min_samples_split = 2)
      : m_max_depth(max_depth), m_min_samples_split(min_samples_split) {};

  void train(const std::vector<std::vector<float>> &X,
             const std::vector<float> &y);
  float predict(const std::vector<float> &x) const;
};
```

5. 附录

github 地址: https://github.com/SWUST-XKCV/math-hw-proj