20220426-C++

1.过程描述

knn.h C++ 复制代码

```cpp
#ifndef __KNN_H
#define __KNN_H

#include "Common.h"

// O(k*n) where k is the number of neighbors and N is the size of training data
// O(n) + O(k*n) + k

class KNN : public CommonData
{
  int k;
  std::vector<Data *> * neighbors;

  public:
  KNN(int);
  KNN();
  ~KNN();

  void findKnearest(Data *queryPoint);
  void setK(int val);
  int findMostFrequentClass();
  double calculateDistance(Data* queryPoint, Data* input);
  double validatePerformance();
  double testPerformance();
};

#endif
```

knn.cpp C++ 复制代码

```cpp
#include "knn.h"
#include <cmath>
#include <limits>
#include <map>
#include "stdint.h"
#include "DataHandler.h"


KNN::KNN(int val)
{
    k = val;
}

KNN::KNN()
{

}

KNN::~KNN()
{
    // NOTHING TO DO
}

void KNN::findKnearest(Data* queryPoint)
{
    neighbors = new std::vector<Data*>;
    double min = std::numeric_limits<double>::max();
    double previousMin = min;
    int index;
    for (int i = 0; i < k; i++)
    {
        if (i == 0)
        {
            for (int j = 0; j < trainingData->size(); j++)
            {
                double dist = calculateDistance(queryPoint,
trainingData->at(j));
                trainingData->at(j)->setDistance(dist);
                if (dist < min)
                {
                    min = dist;
                    index = j;
                }
            }
            neighbors->push_back(trainingData->at(index));
```

```
            previousMin = min;
            min = std::numeric_limits<double>::max();
        }
        else
        {
            for (int j = 0; j < trainingData->size(); j++)
            {
                double dist = trainingData->at(j)->getDistance();
                if (dist > previousMin && dist < min)
                {
                    min = dist;
                    index = j;
                }
            }
            neighbors->push_back(trainingData->at(index));
            previousMin = min;
            min = std::numeric_limits<double>::max();
        }
    }
}
void KNN::setK(int val)
{
    k = val;
}

int KNN::findMostFrequentClass()
{
    std::map<uint8_t, int> frequencyMap;
    for (int i = 0; i < neighbors->size(); i++)
    {
        if (frequencyMap.find(neighbors->at(i)->getLabel()) ==
frequencyMap.end())
        {
            frequencyMap[neighbors->at(i)->getLabel()] = 1;
        }
        else
        {
            frequencyMap[neighbors->at(i)->getLabel()]++;
        }
    }

    int best = 0;
    int max = 0;

    for (auto kv : frequencyMap)
    {
        if (kv.second > max)
        {
```

```
92              max = kv.second;
93              best = kv.first;
94          }
95      }
96      delete neighbors;
97      return best;
98
99  }
100
101 double KNN::calculateDistance(Data* queryPoint, Data* input)
102 {
103     double value = 0;
104     if (queryPoint->getNormalizedFeatureVector()->size() != input-
    >getNormalizedFeatureVector()->size())
105     {
106         printf("Vector size mismatch.\n");
107         exit(1);
108     }
109 #ifdef EUCLID
110     for (unsigned i = 0; i < queryPoint->getNormalizedFeatureVector()-
    >size(); i++)
111     {
112         value += pow(queryPoint->getNormalizedFeatureVector()->at(i) -
    input->getNormalizedFeatureVector()->at(i), 2);
113     }
114     return sqrt(value);
115 #elif defined MANHATTAN
116     //do some stuff
117 #endif
118 }
119
120 double KNN::validatePerformance()
121 {
122     double current_performance = 0;
123     int count = 0;
124     int data_index = 0;
125     for (Data* queryPoint : *validationData)
126     {
127         findKnearest(queryPoint);
128         int prediction = findMostFrequentClass();
129         data_index++;
130         if (prediction == queryPoint->getLabel())
131         {
132             count++;
133         }
134         printf("Current Performance: %.3f %%\n", ((double)count) * 100.0
    / ((double)data_index));
135     }
```

```
136         current_performance = ((double)count) * 100.0 /
    ((double)validationData->size());
137         printf("Validation Performance for K = %d: %.3f\n", k,
    current_performance);
138         return current_performance;
139     }
140     double KNN::testPerformance()
141     {
142         double current_performance = 0;
143         int count = 0;
144         for (Data* queryPoint : *testData)
145         {
146             findKnearest(queryPoint);
147             int prediction = findMostFrequentClass();
148             if (prediction == queryPoint->getLabel())
149             {
150                 count++;
151             }
152         }
153         current_performance = ((double)count) * 100.0 / ((double)testData-
    >size());
154         printf("Validation Performance for K = %d: %.3f\n", k,
    current_performance);
155         return current_performance;
156     }
157
158     int
159     main()
160     {
161         DataHandler* dh = new DataHandler();
162         //dh->read_csv("/home/gerardta/iris.data",",");
163         dh->readInputData("train-images.idx3-ubyte");
164         dh->readLabelData("train-labels.idx1-ubyte");
165         dh->countClasses();
166         dh->splitData();
167         KNN* nearest = new KNN();
168         nearest->setK(3);
169         nearest->setTrainingData(dh->getTrainingData());
170         nearest->setTestData(dh->getTestData());
171         nearest->setValidationData(dh->getValidationData());
172         double performance = 0;
173         double best_performance = 0;
174         int best_k = 1;
175         for (int k = 1; k <= 3; k++)
176         {
177             if (k == 1)
178             {
179                 performance = nearest->validatePerformance();
```

```
            best_performance = performance;
        }
        else
        {
            nearest->setK(k);
            performance = nearest->validatePerformance();
            if (performance > best_performance)
            {
                best_performance = performance;
                best_k = k;
            }
        }
    }
    nearest->setK(best_k);
    nearest->testPerformance();
}
```

common.h C++ 复制代码

```
#ifndef __COMMON_HPP
#define __COMMON_HPP
#include "data.h"
#include <vector>
class CommonData
{
protected:
    std::vector<Data*>* trainingData;
    std::vector<Data*>* testData;
    std::vector<Data*>* validationData;
public:
    void setTrainingData(std::vector<Data*>* vect);
    void setTestData(std::vector<Data*>* vect);
    void setValidationData(std::vector<Data*>* vect);
};
#endif

```

common.cpp C++ 复制代码

```cpp
#include "common.h"

void CommonData::setTrainingData(std::vector<Data*>* vect)
{
    trainingData = vect;
}
void CommonData::setTestData(std::vector<Data*>* vect)
{
    testData = vect;
}
void CommonData::setValidationData(std::vector<Data*>* vect)
{
    validationData = vect;
}
```

data.h C++ 复制代码

```cpp
#ifndef __DATA_H
#define __DATA_H

#include <vector>
#include "stdint.h" // uint8_t
#include "stdio.h"
class Data
{
    std::vector<uint8_t>* featureVector;
    std::vector<double>* normalizedFeatureVector;
    std::vector<int>* classVector;
    uint8_t label;
    uint8_t enumeratedLabel; // A -> 1
    double distance;

public:
    void setDistance(double);
    void setFeatureVector(std::vector<uint8_t>*);
    void setNormalizedFeatureVector(std::vector<double>*);
    void setClassVector(int counts);
    void appendToFeatureVector(uint8_t);
    void appendToFeatureVector(double);
    void setLabel(uint8_t);
    void setEnumeratedLabel(uint8_t);
    void printVector();
    void printNormalizedVector();

    double getDistance();
    int getFeatureVectorSize();
    uint8_t getLabel();
    uint8_t getEnumeratedLabel();

    std::vector<uint8_t>* getFeatureVector();
    std::vector<double>* getNormalizedFeatureVector();
    std::vector<int> getClassVector();

};

#endif


```

data.cpp C++ 复制代码

```cpp
#include "data.h"

void Data::setDistance(double dist)
{
    distance = dist;
}
void Data::setFeatureVector(std::vector<uint8_t>* vect)
{
    featureVector = vect;
}


void Data::setNormalizedFeatureVector(std::vector<double>* vect)
{
    normalizedFeatureVector = vect;
}
void Data::appendToFeatureVector(uint8_t val)
{
    featureVector->push_back(val);
}
void Data::appendToFeatureVector(double val)
{
    normalizedFeatureVector->push_back(val);
}
void Data::setLabel(uint8_t val)
{
    label = val;
}
void Data::setEnumeratedLabel(uint8_t val)
{
    enumeratedLabel = val;
}

void Data::setClassVector(int classCounts)
{
    classVector = new std::vector<int>();
    for (int i = 0; i < classCounts; i++)
    {
        if (i == label)
            classVector->push_back(1);
        else
            classVector->push_back(0);
    }
}

```

```
46  void Data::printVector()
47  {
48      printf("[ ");
49      for (uint8_t val : *featureVector)
50      {
51          printf("%u ", val);
52      }
53      printf("]\n");
54  }
55
56  void Data::printNormalizedVector()
57  {
58      printf("[ ");
59      for (auto val : *normalizedFeatureVector)
60      {
61          printf("%.2f ", val);
62      }
63      printf("]\n");
64
65  }
66
67  double Data::getDistance()
68  {
69      return distance;
70  }
71
72  int Data::getFeatureVectorSize()
73  {
74      return featureVector->size();
75  }
76  uint8_t Data::getLabel()
77  {
78      return label;
79  }
80  uint8_t Data::getEnumeratedLabel()
81  {
82      return enumeratedLabel;
83  }
84
85  std::vector<uint8_t>* Data::getFeatureVector()
86  {
87      return featureVector;
88  }
89  std::vector<double>* Data::getNormalizedFeatureVector()
90  {
91      return normalizedFeatureVector;
92  }
93
```

```
94   std::vector<int>  Data::getClassVector()
95   {
96       return *classVector;
97   }
```

DataHandler.h C++ 复制代码

```cpp
#ifndef __DATA_HANDLER_H
#define __DATA_HANDLER_H

#include "fstream"
#include "stdint.h"
#include "data.h"
#include <vector>
#include <string>
#include <map>
#include <unordered_set>
#include <math.h>

class DataHandler
{
    std::vector<Data*>* dataArray; // all of the data
    std::vector<Data*>* trainingData;
    std::vector<Data*>* testData;
    std::vector<Data*>* validationData;
    int class_counts;
    int featureVectorSize;
    std::map<uint8_t, int> classFromInt;
    std::map<std::string, int> classFromString; //string key

public:
    const double TRAIN_SET_PERCENT = .1;
    const double TEST_SET_PERCENT = .075;
    const double VALID_SET_PERCENT = 0.005;

    DataHandler();
    ~DataHandler();

    void readCsv(std::string, std::string);
    void readInputData(std::string path);
    void readLabelData(std::string path);
    void splitData();
    void countClasses();
    void normalize();
    void print();

    int getClassCounts();
    int getDataArraySize();
    int getTrainingDataSize();
    int getTestDataSize();
    int getValidationSize();

```

```
46        uint32_t format(const unsigned char* bytes);
47
48        std::vector<Data*>* getTrainingData();
49        std::vector<Data*>* getTestData();
50        std::vector<Data*>* getValidationData();
51        std::map<uint8_t, int> getClassMap();
52
53    };
54
55    #endif
```

DataHandler.cpp C++ 复制代码

```cpp
#include "DataHandler.h"
#include <algorithm>
#include <random>
#pragma warning(disable:4996)

DataHandler::DataHandler()
{
    dataArray = new std::vector<Data*>;
    trainingData = new std::vector<Data*>;
    testData = new std::vector<Data*>;
    validationData = new std::vector<Data*>;
}

DataHandler::~DataHandler()
{
    // FIX ME
}

void DataHandler::readCsv(std::string path, std::string delimiter)
{
    class_counts = 0;
    std::ifstream data_file;
    data_file.open(path.c_str());
    std::string line;

    while (std::getline(data_file, line))
    {
        if (line.length() == 0) continue;
        Data* d = new Data();
        d->setNormalizedFeatureVector(new std::vector<double>());
        size_t position = 0;
        std::string token;
        while ((position = line.find(delimiter)) != std::string::npos)
        {
            token = line.substr(0, position);
            d->appendToFeatureVector(std::stod(token));
            line.erase(0, position + delimiter.length());
        }

        if (classFromString.find(line) != classFromString.end())
        {
            d->setLabel(classFromString[line]);
        }
        else
        {
```

```
46              classFromString[line] = class_counts;
47              d->setLabel(classFromString[token]);
48              class_counts++;
49          }
50          dataArray->push_back(d);
51      }
52      for (Data* data : *dataArray)
53          data->setClassVector(class_counts);;
54      //normalize();
55      featureVectorSize = dataArray->at(0)->getNormalizedFeatureVector()-
    >size();
56  }
57
58  void DataHandler::readInputData(std::string path)
59  {
60      uint32_t magic = 0;
61      uint32_t num_images = 0;
62      uint32_t num_rows = 0;
63      uint32_t num_cols = 0;
64
65      unsigned char bytes[4];
66      FILE* f = fopen(path.c_str(), "r");
67      if (f)
68      {
69          int i = 0;
70          while (i < 4)
71          {
72              if (fread(bytes, sizeof(bytes), 1, f))
73              {
74                  switch (i)
75                  {
76                  case 0:
77                      magic = format(bytes);
78                      i++;
79                      break;
80                  case 1:
81                      num_images = format(bytes);
82                      i++;
83                      break;
84                  case 2:
85                      num_rows = format(bytes);
86                      i++;
87                      break;
88                  case 3:
89                      num_cols = format(bytes);
90                      i++;
91                      break;
92                  }
```

```
            }
        }
        printf("Done getting file header.\n");
        uint32_t image_size = num_rows * num_cols;
        for (i = 0; i < num_images; i++)
        {
            Data* d = new Data();
            d->setFeatureVector(new std::vector<uint8_t>());
            uint8_t element[1];
            for (int j = 0; j < image_size; j++)
            {
                if (fread(element, sizeof(element), 1, f))
                {
                    d->appendToFeatureVector(element[0]);
                }
            }
            dataArray->push_back(d);
            dataArray->back()->setClassVector(class_counts);
        }
        normalize();
        featureVectorSize = dataArray->at(0)->getFeatureVector()->size();
        printf("Successfully read %lu data entries.\n", dataArray->size());
        printf("The Feature Vector Size is: %d\n", featureVectorSize);
    }
    else
    {
        printf("Invalid Input File Path\n");
        exit(1);
    }
}
void DataHandler::readLabelData(std::string path)
{
    uint32_t magic = 0;
    uint32_t num_images = 0;
    unsigned char bytes[4];
    FILE* f = fopen(path.c_str(), "r");
    if (f)
    {
        int i = 0;
        while (i < 2)
        {
            if (fread(bytes, sizeof(bytes), 1, f))
            {
                switch (i)
                {
                case 0:
```

```
                    magic = format(bytes);
                    i++;
                    break;
                case 1:
                    num_images = format(bytes);
                    i++;
                    break;
                }
            }
        }

        for (unsigned j = 0; j < num_images; j++)
        {
            uint8_t element[1];
            if (fread(element, sizeof(element), 1, f))
            {
                dataArray->at(j)->setLabel(element[0]);
            }
        }

        printf("Done getting Label header.\n");
    }
    else
    {
        printf("Invalid Label File Path\n");
        exit(1);
    }
}
void DataHandler::splitData()
{
    std::unordered_set<int> used_indexes;
    int train_size = dataArray->size() * TRAIN_SET_PERCENT;
    int test_size = dataArray->size() * TEST_SET_PERCENT;
    int valid_size = dataArray->size() * VALID_SET_PERCENT;

    std::random_shuffle(dataArray->begin(), dataArray->end());

    // Training Data

    int count = 0;
    int index = 0;
    while (count < train_size)
    {
        trainingData->push_back(dataArray->at(index++));
        count++;
    }

    // Test Data
```

```
187         count = 0;
188         while (count < test_size)
189         {
190             testData->push_back(dataArray->at(index++));
191             count++;
192         }
193 
194         // Test Data
195 
196         count = 0;
197         while (count < valid_size)
198         {
199             validationData->push_back(dataArray->at(index++));
200             count++;
201         }
202 
203         printf("Training Data Size: %lu.\n", trainingData->size());
204         printf("Test Data Size: %lu.\n", testData->size());
205         printf("Validation Data Size: %lu.\n", validationData->size());
206     }
207 
208     void DataHandler::countClasses()
209     {
210         int count = 0;
211         for (unsigned i = 0; i < dataArray->size(); i++)
212         {
213             if (classFromInt.find(dataArray->at(i)->getLabel()) ==
    classFromInt.end())
214             {
215                 classFromInt[dataArray->at(i)->getLabel()] = count;
216                 dataArray->at(i)->setEnumeratedLabel(count);
217                 count++;
218             }
219             else
220             {
221                 dataArray->at(i)->setEnumeratedLabel(classFromInt[dataArray-
    >at(i)->getLabel()]);
222             }
223         }
224 
225         class_counts = count;
226         for (Data* data : *dataArray)
227             data->setClassVector(class_counts);
228         printf("Successfully Extraced %d Unique Classes.\n", class_counts);
229     }
230 
231     void DataHandler::normalize()
232     {
```

```
233         std::vector<double> mins, maxs;
234         // fill min and max lists
235 
236         Data* d = dataArray->at(0);
237         for (auto val : *d->getFeatureVector())
238         {
239             mins.push_back(val);
240             maxs.push_back(val);
241         }
242 
243         for (int i = 1; i < dataArray->size(); i++)
244         {
245             d = dataArray->at(i);
246             for (int j = 0; j < d->getFeatureVectorSize(); j++)
247             {
248                 double value = (double)d->getFeatureVector()->at(j);
249                 if (value < mins.at(j)) mins[j] = value;
250                 if (value > maxs.at(j)) maxs[j] = value;
251             }
252         }
253         // normalize data array
254 
255         for (int i = 0; i < dataArray->size(); i++)
256         {
257             dataArray->at(i)->setNormalizedFeatureVector(new
    std::vector<double>());
258             dataArray->at(i)->setClassVector(class_counts);
259             for (int j = 0; j < dataArray->at(i)->getFeatureVectorSize();
    j++)
260             {
261                 if (maxs[j] - mins[j] == 0) dataArray->at(i)-
    >appendToFeatureVector(0.0);
262                 else
263                     dataArray->at(i)->appendToFeatureVector(
264                         (double)(dataArray->at(i)->getFeatureVector()->at(j)
    - mins[j]) / (maxs[j] - mins[j]));
265             }
266         }
267     }
268 
269     int DataHandler::getClassCounts()
270     {
271         return class_counts;
272     }
273 
274     int DataHandler::getDataArraySize()
275     {
276         return dataArray->size();
```

```
277   }
278   int DataHandler::getTrainingDataSize()
279   {
280       return trainingData->size();
281   }
282   int DataHandler::getTestDataSize()
283   {
284       return testData->size();
285   }
286   int DataHandler::getValidationSize()
287   {
288       return validationData->size();
289   }
290
291   uint32_t DataHandler::format(const unsigned char* bytes)
292   {
293       return (uint32_t)((bytes[0] << 24) |
294           (bytes[1] << 16) |
295           (bytes[2] << 8) |
296           (bytes[3]));
297   }
298
299   std::vector<Data*>* DataHandler::getTrainingData()
300   {
301       return trainingData;
302   }
303   std::vector<Data*>* DataHandler::getTestData()
304   {
305       return testData;
306   }
307   std::vector<Data*>* DataHandler::getValidationData()
308   {
309       return validationData;
310   }
311
312   std::map<uint8_t, int> DataHandler::getClassMap()
313   {
314       return classFromInt;
315   }
316
317   void DataHandler::print()
318   {
319       printf("Training Data:\n");
320       for (auto data : *trainingData)
321       {
322           for (auto value : *data->getNormalizedFeatureVector())
323           {
324               printf("%.3f,", value);
```

```
            }
            printf(" ->   %d\n", data->getLabel());
        }
        return;

        printf("Test Data:\n");
        for (auto data : *testData)
        {
            for (auto value : *data->getNormalizedFeatureVector())
            {
                printf("%.3f,", value);
            }
            printf(" ->   %d\n", data->getLabel());
        }

        printf("Validation Data:\n");
        for (auto data : *validationData)
        {
            for (auto value : *data->getNormalizedFeatureVector())
            {
                printf("%.3f,", value);
            }
            printf(" ->   %d\n", data->getLabel());
        }

    }

```

2.结果输出

今天只看了一个C++ ML的视频，效率低下，赶紧找回状态！