ORBSLAM2学习（四）：DBoW2源码分析（OrbVocabulary部分）

2018年06月01日 14:12:08 Mega_Li 阅读数 1885 更多

0 前言

开始DBoW2源代码的学习，以工程中已有的demo.cpp为例分析DBoW2的使用与运行原理。demo.cpp中用到了OrbVocabulary和OrbDatabas过ORBSLAM2中只用到了前者，因此本文只对OrbVocabulary的使用做代码分析。OrbDatabase类，应该会在后面学习DLoopDetector（DBoW2作源项目）中用到吧，到时候再分析。。。

1 工程编译与运行

DBoW2的github地址为https://github.com/dorian3d/DBoW2，下载后进入工作目录，依次执行

```
mkdir build
cd build
cmake ../
make
```

编译过程中会下载DLib软件包（github地址https://github.com/dorian3d/DLib），包含一些作者整理的开发中常用的功能性代码。编译结束后录下生成可执行文件demo，可直接运行，效果如下图所示（只截了OrbVocabulary功能部分）：

```
leon@leon-VirtualBox:~/DBoW2-master/build$ ./demo
Extracting ORB features...
Creating a small 9^3 vocabulary...
... done!
Vocabulary information:
Vocabulary: k = 9, L = 3, Weighting = tf-idf, Scoring = L1-norm, Number of words = 704

Matching images against themselves (0 low, 1 high):
Image 0 vs Image 0: 1
Image 0 vs Image 1: 0.0672136
Image 0 vs Image 2: 0.0732975
Image 0 vs Image 3: 0.0602973
Image 1 vs Image 0: 0.0672136
Image 1 vs Image 1: 1
Image 1 vs Image 2: 0.139872
Image 1 vs Image 3: 0.217703
Image 2 vs Image 0: 0.0732975
Image 2 vs Image 1: 0.139872
Image 2 vs Image 2: 1
Image 2 vs Image 3: 0.138988
Image 3 vs Image 0: 0.0602973
Image 3 vs Image 1: 0.217703
Image 3 vs Image 2: 0.138988
Image 3 vs Image 3: 1

Saving vocabulary...
Done
```


可以看到demo中程序作了下面几件事：

1）提取图像集的特征

2）建立了一个vocabulary

3）利用vocabulary计算图像之间的相似度

4）保存vocabulary

现在看一下demo.cpp中的main()函数

```
1  int main()
2  {
3    vector<vector<cv::Mat > > features;
4    loadFeatures(features);
5
6    testVocCreation(features);
7
8    wait();
9
10   testDatabase(features);
11
```

```
12    return 0;
13  }
```

我们后面将主要分析loadFeatures()和testVocCreation()函数。

2 代码分析

2.1 提取图像特征

```
void loadFeatures(vector<vector<cv::Mat > > &features)
{
  features.clear();
  features.reserve(NIMAGES);

  cv::Ptr<cv::ORB> orb = cv::ORB::create();// 调用opencv中的ORB算子

  cout << "Extracting ORB features..." << endl;
  for(int i = 0; i < NIMAGES; ++i)
  {
    stringstream ss;
    ss << "images/image" << i << ".png";

    cv::Mat image = cv::imread(ss.str(), 0);
    cv::Mat mask;
    vector<cv::KeyPoint> keypoints;
    cv::Mat descriptors;

    orb->detectAndCompute(image, mask, keypoints, descriptors);

    features.push_back(vector<cv::Mat >());
    changeStructure(descriptors, features.back());
  }
}
void changeStructure(const cv::Mat &plain, vector<cv::Mat> &out)
{
  out.resize(plain.rows);

  for(int i = 0; i < plain.rows; ++i)
  {
    out[i] = plain.row(i);// 每一个256bit的描述子单独占用一个cv::Mat结构
  }
}
```

demo.cpp中调用loadFeatures()，使用opencv中的ORB提取特征点和描述子，每幅图像得到一个cv::Mat类型的保存了多个描述子数据；chang
之后会把它转换为vector<cv::Mat>，即一个Mat代表一个描述子，一个vector<cv::Mat>代表一幅图像的描述子序列

2.2 创建vocabulary

在这之前需要结合上一篇文章（链接）介绍DBoW2中的一些基础类，方便后面的分析讲解。

2.2.1 基础知识准备

1）FORB

类FORB派生自FClass，简单查看下FClass，它是一个虚类，定义了一些对描述子操作的函数，重要的包括（1）meanValue()：用于计算描述子集
（2）distance()：计算两个描述子之间的距离。

FORB类是针对ORB检测到的描述子做操作函数的具体实现。我们提取得到的ORB描述子是一个指定长度的256bit的二进制序列，FORB::m
是多个256bit的二进制序列的中值。基本操作是对256bit中的每一位统计1的数量，然后根据1的数量是否超过描述子集合数量一半确定中值结
FORB::distance()计算两个ORB描述子之间的汉明距离。

2）BowVector类

还记得上一篇文章中我们说到，DBoW2中将图像最终转换为{(w1,weight1),(w2,weight2),...,(wn,weightn)}的形式，就对应着这里的BowV
派生自public std::map<WordId, WordValue>，实际上就是public std::map<int, double>。有关std::map这里做简单介绍，map是STL中
供一对一（第一个称为关键字，在map中唯一，第二个可称为键值）的映射关系。map自动建立key-value的对应，能够快速插入/删除key-va

速获取对应的value。BowVector封装了增加元素的操作addWeight()，注意其中的插入操作确保了元素是按照key升序排列的，这对后面调用L1Scori方法计算相似度时提供了便利。BowVector还封装了normalize()函数用于对元素的value也就是单词的权值做归一化操作。

3）GeneralScoring类

作者定义了GeneralScoring虚类，定义了计算两个BowVector对象之间相似度的虚函数score()，并根据计算方法的不同派生了L1Scoring、L2Sc demo中用到了L1Scoring类。

看一下L1Scoring的score()具体实现，如下。

```
double L1Scoring::score(const BowVector &v1, const BowVector &v2) const
{
  BowVector::const_iterator v1_it, v2_it;
  const BowVector::const_iterator v1_end = v1.end();
  const BowVector::const_iterator v2_end = v2.end();

  v1_it = v1.begin();
  v2_it = v2.begin();

  double score = 0;

  while(v1_it != v1_end && v2_it != v2_end)
  {
    const WordValue& vi = v1_it->second;
    const WordValue& wi = v2_it->second;

    if(v1_it->first == v2_it->first)
    {
      score += fabs(vi - wi) - fabs(vi) - fabs(wi);

      // move v1 and v2 forward
      ++v1_it;
      ++v2_it;
    }
    else if(v1_it->first < v2_it->first)
    {
      // move v1 forward
      v1_it = v1.lower_bound(v2_it->first);
      // v1_it = (first element >= v2_it.id)
    }
    else
    {
      // move v2 forward
      v2_it = v2.lower_bound(v1_it->first);
      // v2_it = (first element >= v1_it.id)
    }
  }
```

对于两个BowVector对象v1和v2，对象中各个元素已经按照关键字（wordId）升序排列，但对象间需要保证元素的关键字相同（对应同一个单词算其差异。因此需要通过一些手段做元素间关键字的对齐，代码中通过lower_bound()实现，对齐后权值间差异度计算通过下面的公式来计算，这样就v2的相似度。由于v1和v2会在计算前调用normalize()做归一化处理，score()的返回值在[0,1]之间，值越大代表相似度越高。

$$s(\boldsymbol{v}_A - \boldsymbol{v}_B) = 2\sum_{i=1}^{N} |\boldsymbol{v}_{Ai}| + |\boldsymbol{v}_{Bi}| - |\boldsymbol{v}_{Ai} - \boldsymbol{v}_{Bi}|$$

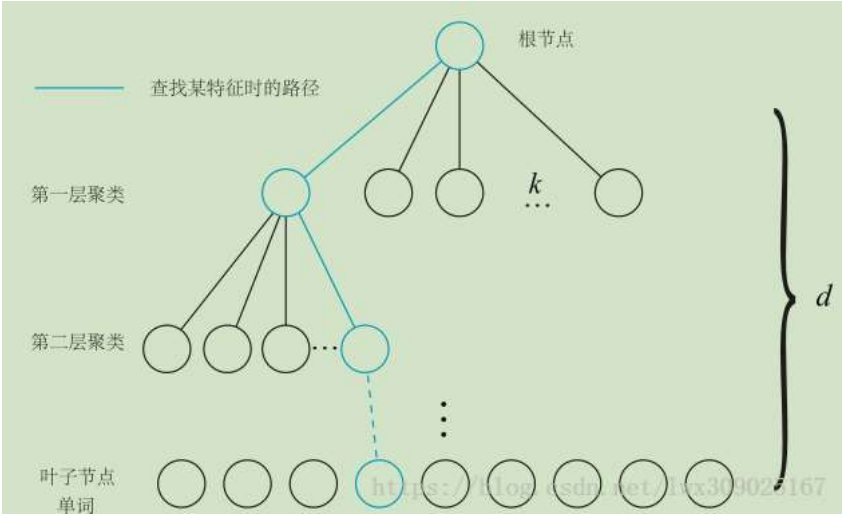
4）OrbVocabulary类

查看代码，发现OrbVocabulary是这样定义的

typedef DBoW2::TemplatedVocabulary<DBoW2::FORB::TDescriptor, DBoW2::FORB> OrbVocabulary;

TemplatedVocabulary是一个字典模板类，具体定义见TemplatedVocabulary.h，它定义了字典类的通用操作：字典的构造、保存、读取 为单词表示等。 OrbVocabulary中传入的类是DBoW2::FORB::TDescriptor和DBoW2::FORB，前者是描述子类型的定义，查看代码后发现就是 ORB得到的描述子使用cv::Mat保存）；后者前面已经介绍了。

回忆一下vocabulary tree的结构，如下图所示。有很多节点，同时只有叶子节点代表一个word。



TemplatedVocabulary中定义了Node结构体用于保存节点信息，树中有很多节点，但只有叶子节点代表word，节点之间存在“父子”关系；父表word，但也有自己的一个描述子，是为了加速特征转换为word的匹配过程。这样看Node中的各个元素定义就很清晰了。TemplatedVocabulary中vector<Node> m_nodes用来保存生成的vocabulary tree。

```
/// Tree node
 struct Node
 {
   /// Node id
   NodeId id;
   /// Weight if the node is a word
   WordValue weight;
   /// Children
   std::vector<NodeId> children;
   /// Parent node (undefined in case of root)
   NodeId parent;
   /// Node descriptor
   TDescriptor descriptor;

   /// Word id if the node is a word
   WordId word_id;

   /**
    * Empty constructor
    */
   Node(): id(0), weight(0), parent(0), word_id(0){}

   /**
    * Constructor
    * @param _id node id
    */
   Node(NodeId _id): id(_id), weight(0), parent(0), word_id(0){}

   /**
    * Returns whether the node is a leaf node
    * @return true iff the node is a leaf
    */
   inline bool isLeaf() const { return children.empty(); }// 没有子节点的节点就是最后一层的叶子，代表一个word
 };
```

另外由于vocabulary tree中node数量大于word，TemplatedVocabulary中还定义了成员变量vector<Node*> m_words，专门用于保存代表w且索引从0开始，满足m_words[wid]->word_id == wid

2.2.2 代码分析

基础知识分析完毕，进入代码分析阶段，testVocCreation()中主要包括五步：创建字典对象，构造字典，利用字典将图像转换为BowVect 的相似度，保存字典。

```
void testVocCreation(const vector<vector<cv::Mat > > &features)
{
```

```
  // branching factor and depth levels 
  const int k = 9;
  const int L = 3;
  const WeightingType weight = TF_IDF;
  const ScoringType score = L1_NORM;

  OrbVocabulary voc(k, L, weight, score);// 创建字典对象

  cout << "Creating a small " << k << "^" << L << " vocabulary..." << endl;
  voc.create(features);// 关键函数，构造字典
  cout << "... done!" << endl;

  cout << "Vocabulary information: " << endl
  << voc << endl << endl;

  // lets do something with this vocabulary
  cout << "Matching images against themselves (0 low, 1 high): " << endl;
  BowVector v1, v2;
  for(int i = 0; i < NIMAGES; i++)
  {
    voc.transform(features[i], v1);
    for(int j = 0; j < NIMAGES; j++)
    {
      voc.transform(features[j], v2);// 图像特征转换为BowVector对象

      double score = voc.score(v1, v2);// 计算图像相似度
      cout << "Image " << i << " vs Image " << j << ": " << score << endl;
    }
  }

  // save the vocabulary to disk
  cout << endl << "Saving vocabulary..." << endl;
  voc.save("small_voc.yml.gz");// 保存vocabulary
  cout << "Done" << endl;
}

```

1）创建字典对象

OrbVocabulary voc(k, L, weight, score);声明了层数为3，每一个父节点聚类数目为9的字典，权值计算使用TF_IDF，计算图像间相似度时使用L1构造函数内部会调用createScoringObject()构造用于计算图像间相似度的L1Scoring类对象。

2）构造字典

现在看TemplatedVocabulary构造vocabulary tree的create()函数，demo中对描述子序列做了重排操作，使得std::vector<std::vector<TDesc &training_features中的每个元素代表的都是一个256bit的二进制特征描述子。

```
template<class TDescriptor, class F>
void TemplatedVocabulary<TDescriptor,F>::create(
  const std::vector<std::vector<TDescriptor> > &training_features)
{
  m_nodes.clear();
  m_words.clear();

  // expected_nodes = Sum_{i=0..L} ( k^i )
        int expected_nodes =
                (int)((pow((double)m_k, (double)m_L + 1) - 1)/(m_k - 1));

  m_nodes.reserve(expected_nodes); // avoid allocations when creating the tree


  std::vector<pDescriptor> features;
  getFeatures(training_features, features);


  // create root
  m_nodes.push_back(Node(0)); // root

  // create the tree
  HKmeansStep(0, features, 1);

```

```
25    // create the words
26    createWords();
27
28    // and set the weight of each node of the tree
29    setNodeWeights(training_features);
30
31  }
```

首先，计算建立k分支，L层的tree会产生多少个节点并为m_nodes分配空间。之后调用getFeatures()把类似于二维数组的cv::Mat集合保存到了一构中，行数就是描述子的数量，每一行都是一个256bit的二进制描述子。

之后，m_nodes中先加入根节点，然后HKmeansStep()把这些nodes组织为vocabulary tree的形式。基本原理是迭代利用Kmeans++算法聚类，上一篇文章中已经叙述过，代码中Kmeans++迭代终止条件是分类状态不再发生改变。需要注意的是迭代中不同情况的处理：有些分支不一定扩展到第征数量不够多，或者分支上的特征数量不够多）；有时候分支中特征数量小于等于k，则不需要使用Kmeans++聚类。注意在聚类计算中，也为每个非点计算了一个特征描述子，计算方法是调用FORB::meanValue()求该父节点拥有的所有描述子的中值。

得到了vocabulary tree，调用createWords()函数向成员变量vector<Node*> m_words填充代表word的叶子node数据。

demo中使用TF_IDF权值，因此现在需要使用IDF计算word的权值，通过setNodeWeights()实现。由于输入变量是每幅图像的特征描述子，我们转换为word，调用void TemplatedVocabulary<TDescriptor,F>::transform

(const TDescriptor &feature, WordId &id) const实现，这个函数后面再详细讲。然后累计每个word拥有的特征数量Ni，IDF部分的权值就是log(N是vocabulary tree中特征数量总和。

由此，vocabulary tree的建立完毕，后面使用它将图像转换为BowVector的形式。

3）图像转换为BowVector对象

首先需要使用特征提取算法得到图像的特征描述子，之后调用TemplatedVocabulary的transform()函数做转换。转换基本流程就是从根节点开始特征描述子和节点自身的特征描述子使用FORB::distance()计算距离，取距离最小的节点作为迭代计算新的起点，直至找到一个叶子节点，也就是word中会计算TF部分权值，得到各个word的权值。

4）计算图像相似度

得到两幅图像对应的BowVector对象后，调用score()（代码中调用L1Scoring::score()）计算两幅图像之间的相似度。

5）保存vocabulary tree

调用TemplatedVocabulary类的save()函数保存，其中会调用opencv的FileStorage类做操作，具体不再分析。

3 ORBSLAM2中的DBoW2

ORBSLAM2中 对DBoW2做了一定的裁剪，最主要的是没有用到TemplatedDatabase和它的派生类，由于版本原因代码在一些地方有一点差异，分相同。具体如何使用的，还没看到这一阶段，后面再总结分析吧。