P1:

应该是NEARBY8。 比NEARBY6就是加上L1距离为2的12个点

```
template <>
void GridNN<3>::GenerateNearbyGrids() {
    if (nearby_type_ == NearbyType::CENTER) {
        nearby_grids_.emplace_back(KeyType::Zero());
    } else if (nearby_type_ == NearbyType::NEARBY6) {
        nearby_grids_ = {KeyType(0, 0, 0),  KeyType(-1, 0, 0), KeyType(1, 0, 0),
KeyType(0, 1, 0),
                         KeyType(0, -1, 0), KeyType(0, 0, -1), KeyType(0, 0, 1)};
    } else if (nearby_type_ == NearbyType::NEARBY18) {
        nearby_grids_ = {KeyType(0, 0, 0),
                         KeyType(-1, 0, 0), KeyType(1, 0, 0), KeyType(0, 1, 0),
                         KeyType(0, -1, 0), KeyType(0, 0, -1), KeyType(0, 0, 1),
                         KeyType(-1, -1, 0), KeyType(-1, 1, 0), KeyType(1, -1,
0), KeyType(1, 1, 0),
                         KeyType(0, -1, -1), KeyType(0, -1, 1), KeyType(0, 1,
-1), KeyType(0, 1, 1),
                         KeyType(-1, 0, -1), KeyType(-1, 0, 1), KeyType(1, 0,
-1), KeyType(1, 0, 1)};
    }
}
```

```
0230907 01:02:52.285648 17573 sys_utils.h:32] 方法 Grid 3D 单线程 平均调用时间/次数:
7.50573/10 毫秒.
I20230907 01:02:52.285670 17573 test_nn.cc:108] truth: 18779, esti: 8572
I20230907 01:02:52.351713 17573 test_nn.cc:134] precision: 0.911339, recall:
0.415997, fp: 760, fn: 10967
I20230907 01:02:52.351732 17573 test_nn.cc:213] ===================
I20230907 01:02:52.368994 17573 sys_utils.h:32] 方法 Grid 3D 多线程 平均调用时间/次
数: 1.7251/10 毫秒.
I20230907 01:02:52.369032 17573 test_nn.cc:108] truth: 18779, esti: 18779
I20230907 01:02:52.484673 17573 test_nn.cc:134] precision: 0.911339, recall:
0.415997, fp: 760, fn: 10967
I20230907 01:02:52.484694 17573 test_nn.cc:219] ===================
I20230907 01:02:52.631883 17573 sys_utils.h:32] 方法 Grid 3D 18 体素 单线程 平均调用
时间/次数: 14.7177/10 毫秒.
I20230907 01:02:52.631903 17573 test_nn.cc:108] truth: 18779, esti: 10070
I20230907 01:02:52.699419 17573 test_nn.cc:134] precision: 0.964846, recall:
0.517386, fp: 354, fn: 9063
I20230907 01:02:52.699427 17573 test_nn.cc:225] ===================
I20230907 01:02:52.730659 17573 sys_utils.h:32] 方法 Grid 3D 18 体素 多线程 平均调用
时间/次数: 3.12215/10 毫秒.
I20230907 01:02:52.730824 17573 test_nn.cc:108] truth: 18779, esti: 18779
I20230907 01:02:52.831476 17573 test_nn.cc:134] precision: 0.964846, recall:
0.517386, fp: 354, fn: 9063
```

可以看到18体素的速度比6体素慢很多，但是准确率和recall都高了不少。

P2:

$$2.\ d^* = \arg\max \|Ad\|_2^2$$

$$\|Ad\|_2^2 = d^T A^T A d$$

$$A = U\Sigma V^T$$

$$\|Ad\|_2^2 = d^T V \Sigma V^T d$$

由于 V 的各向量为 Ortho-normal

$$d^* = \arg\max_{d}\ d^T V \Sigma V^T d$$

d 可取最大值为 Σ 最大值对应的特征向最

$\therefore d^* =$ A 的最大特征向量

P3:
将nanoflann.h放到文件夹里，再把nanoflann里的utils.h中的PointCloud放到test_nn.cc中（因为例子中是用这个）。
尝试了一下用gridnn的形式来搭nanoflann的实现。一开始把KDTreeSingleIndexAdaptor放在getClosestPoint函数里面，但是速度太慢了，
放在构建函数中后编译错误，应该是KDTreeSingleIndexAdaptor没有默认（无输入的）构建函数所以遇到了一些问题，最后直接在test_nn.cc中实现了nanoflann

一开始使用了findNeighbors函数来找，但是在实现的时候没有找到5NN的实现方法，最后用了knnSearch 来做5NN

```
TEST(CH5_TEST, NANOFLANN_KNN) {
    sad::CloudPtr first(new sad::PointCloudType), second(new
sad::PointCloudType);
    pcl::io::loadPCDFile(FLAGS_first_scan_path, *first);
    pcl::io::loadPCDFile(FLAGS_second_scan_path, *second);

    if (first->empty() || second->empty()) {
        LOG(ERROR) << "cannot load cloud";
        FAIL();
    }

    // voxel grid 至 0.05
    sad::VoxelGrid(first);
    sad::VoxelGrid(second);

    PointCloud_flann<float> first_cloud;
    first_cloud.pts.resize(first->size());
    std::vector<size_t> num_index(first->size());
    std::for_each(num_index.begin(), num_index.end(), [idx = 0](size_t& i)
mutable { i = idx++; });

    std::for_each(num_index.begin(), num_index.end(), [&first_cloud, &first,
this](const size_t& idx) {
        auto pt = first->points[idx];
        first_cloud.pts[idx].x = pt.x;
        first_cloud.pts[idx].y = pt.y;
        first_cloud.pts[idx].z = pt.z;
    });

    using my_kd_tree_t = nanoflann::KDTreeSingleIndexAdaptor<
        nanoflann::L2_Simple_Adaptor<float, PointCloud_flann<float>>,
        PointCloud_flann<float>, 3>;
    my_kd_tree_t flann_knn(3, first_cloud, {10});
    flann_knn.buildIndex();

    // 比较 bfnn
    std::vector<std::pair<size_t, size_t>> true_matches;
    sad::bfnn_cloud_mt_k(first, second, true_matches);

    // 对第2个点云执行knn
    std::vector<std::pair<size_t, size_t>> matches;

    matches.clear();

    std::vector<size_t> index(second->size());

    matches.resize(index.size() * 5);
    auto t1 = std::chrono::high_resolution_clock::now();
    std::for_each(index.begin(), index.end(), [idx = 0](size_t& i) mutable { i =
idx++; });
    std::for_each(index.begin(), index.end(), [this, &matches, &second,
&flann_knn](const size_t& idx) {
        size_t cp_idx;
        auto pt = second->points[idx];
        float query_pt[3] = {pt.x, pt.y, pt.z};
```

```
        size_t                num_results = 5;              // 最近的5个点
        std::vector<uint32_t> ret_index(num_results);       // 返回的点的索引
        std::vector<float>    out_dist_sqr(num_results);    // 返回的点的距离

        num_results = flann_knn.knnSearch(&query_pt[0], num_results,
&ret_index[0], &out_dist_sqr[0]);
        for (int i = 0; i < ret_index.size(); ++i) {
            matches[idx * 5 + i].first = ret_index[i];
            matches[idx * 5 + i].second = idx;
        }


    });
    auto t2 = std::chrono::high_resolution_clock::now();
    auto total_time = std::chrono::duration_cast<std::chrono::duration<double>>
(t2 - t1).count() * 1000;

    LOG(INFO) << "方法 " << "NANOFLANN Kd Tree 5NN" << " 平均调用时间/次数: " <<
total_time << "/1 毫秒.";
    EvaluateMatches(true_matches, matches);

    LOG(INFO) << "done.";

    SUCCEED();
}
```

跑出来的结果：

```
I20230907 14:56:45.584944  7577 test_nn.cc:473] 方法 NANOFLANN Kd Tree 5NN 平均调用
时间/次数: 13.4338/1 毫秒.
I20230907 14:56:45.584970  7577 test_nn.cc:108] truth: 93895, esti: 93895
I20230907 14:56:48.908072  7577 test_nn.cc:134] precision: 1, recall: 1, fp: 0,
fn: 0
I20230907 14:56:48.908098  7577 test_nn.cc:476] done.
```

在我的电脑上其他算法的速度：

```
I20230907 14:56:25.992892  7577 sys_utils.h:32] 方法 Kd Tree build 平均调用时间/次
数: 6.93181/1 毫秒.
I20230907 14:56:25.992902  7577 test_nn.cc:284] Kd tree leaves: 18869, points:
18869
I20230907 14:56:28.940630  7577 sys_utils.h:32] 方法 Kd Tree 5NN 多线程 平均调用时间/
次数: 5.66564/1 毫秒.
I20230907 14:56:28.940672  7577 test_nn.cc:108] truth: 93895, esti: 93895
I20230907 14:56:32.387584  7577 test_nn.cc:134] precision: 1, recall: 1, fp: 0,
fn: 0
I20230907 14:56:32.387611  7577 test_nn.cc:296] building kdtree pcl
I20230907 14:56:32.402770  7577 sys_utils.h:32] 方法 Kd Tree build 平均调用时间/次
数: 15.1221/1 毫秒.
I20230907 14:56:32.402798  7577 test_nn.cc:301] searching pcl
I20230907 14:56:32.457382  7577 sys_utils.h:32] 方法 Kd Tree 5NN in PCL 平均调用时
间/次数: 54.5544/1 毫秒.
I20230907 14:56:32.457669  7577 test_nn.cc:108] truth: 93895, esti: 93895
I20230907 14:56:36.010355  7577 test_nn.cc:134] precision: 1, recall: 1, fp: 0,
fn: 0
```

```
I20230907 14:56:36.010380  7577 test_nn.cc:322] done.
[       OK ] CH5_TEST.KDTREE_KNN (10032 ms)
[ RUN      ] CH5_TEST.OCTREE_BASICS
I20230907 14:56:36.013893  7577 test_nn.cc:354] Octo tree leaves: 4, points: 4
[       OK ] CH5_TEST.OCTREE_BASICS (0 ms)
[ RUN      ] CH5_TEST.OCTREE_KNN
Failed to find match for field 'intensity'.
Failed to find match for field 'intensity'.
I20230907 14:56:36.052919  7577 sys_utils.h:32] 方法 Octo Tree build 平均调用时间/次数: 33.6464/1 毫秒.
I20230907 14:56:36.052945  7577 test_nn.cc:377] Octo tree leaves: 18869, points: 18869
I20230907 14:56:36.052949  7577 test_nn.cc:380] testing knn
I20230907 14:56:36.090013  7577 sys_utils.h:32] 方法 Octo Tree 5NN 多线程 平均调用时间/次数: 37.0516/1 毫秒.
```

可以看出，除了kd tree的速度比nanoflann快之外，其他的速度都比nanoflann慢一些。这几个算法的准确度和召回度也都在100%。