/home/adam/workspace/Tagger3D/utils/src

/home/adam/workspace/Tagger3D/utils/src/Cluster

```
/*
 * Cluster.cpp
 *
 *  Created on:         20-06-2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "Cluster.h"

#include <stdexcept>

namespace Tagger3D {

Cluster::Cluster(const std::map<std::string, std::string>& _configMap)
        : ProcessObject(_configMap),
          loaded(false) {

        logger = lgr::Logger::getLogger(loggerName);
        DEBUG(logger, "Creating Cluster");

        clusterCount = getParam<int>(clusterCountKey);
        dimCount = getParam<int>(dimCountKey);
}

Cluster::~Cluster() {

        DEBUG(logger, "Destroying Cluster");
}

cv::Mat Cluster::cluster(const std::vector<cv::Mat> &descriptors) {

        const cv::Mat *dPtr = &descriptors[0];
        long size = descriptors.size();

        cv::Mat clustered = cluster(dPtr[0]);
        clustered.reserve(size);
        for(int i = 1; i < size; ++i) {
                clustered.push_back(cluster(dPtr[i]));
        }

        return clustered.clone();
}


} /* namespace Tagger3D */


/*
 * Cluster.h
 *
 *  Created on:                 20-06-2013
 *      Author:                 Adam Kosiorek
 *      Description:
 */

#ifndef CLUSTER_H_
#define CLUSTER_H_

#include "../Common/ProcessObject.h"

#include <opencv2/core/core.hpp>
```

```
#include <vector>

namespace Tagger3D {
/**
 *      The class is a pure virtual class - an interface for clasterization.
 */
class Cluster : public ProcessObject {
public:


        /**
         *      Parametric constructor.
         *      @param  _configMap      A map of configuration parameters
         *      @return void
         *      @throws std::invalid_argument
         */
        Cluster(const std::map<std::string, std::string> &_configMap);

        /**
         * Default destructor.
         */
        virtual ~Cluster();

        /**
         *      The method computes a bag of words description of an input image.
         *      @param  descriptors     A set of descriptors for which a bag of words descr
iption is to be found.
         *      @return cv::Mat A vector of visual words.
         */
        virtual cv::Mat cluster(const cv::Mat &descriptors) = 0;
        virtual cv::Mat cluster(const std::vector<cv::Mat> &descriptors);


        /**
         *      The method trains the clasterizator.
         *      @iparam nputData        a cv::Matrix of [number of descriptor] x [number of
 points].
         *                              If points from multiple images are passed than:
         *                              number of points = number of points in a single ima
ge x number of images.
         *      @return bool    true if the training succeded, false otherwise.
         */
        virtual void train(const std::vector<cv::Mat> &descriptors) = 0;

        /**
         *      The method saves the KMeans (trained model and configuration) to a folder
         *      specified in a config file.
         *      In case the folder does not exist it will be created.
         *      Note that there should be only one KMeans per folder stored.
         *      @ return true if save successful, false otherwise
         */
        virtual void save() = 0;


        /**
         *      The method loads the KMeans(trained model and configuration) from a folder
         *      specified in a config file.
         *      Note that there should be only one KMeans per folder.
         *      @return true if load successful, false otherwise
         */
        virtual void load() = 0;

        virtual bool isLoaded() = 0;

protected:

        int clusterCount;
```

```
        int dimCount;
        const std::string clusterCountKey = "dictionarySize";
        const std::string dimCountKey = "dimCount";

        const std::string moduleName = "cluster" + separator;
        bool loaded;

private:
        const std::string loggerName = "Main.Cluster";

};

} /* namespace Tagger3D */
#endif /* CLUSTER_H_ */


/*
 * FisherCluster.cpp
 *
 *  Created on: 21 gru 2013
 *      Author: adam
 */

#include <FisherCluster.h>

#include <cstdio>

namespace Tagger3D {

FisherCluster::FisherCluster(const std::map<std::string, std::string> &_configMap) : Cluste
r(_configMap) {

        encodingSize = 2 * dimCount + clusterCount;

}

FisherCluster::~FisherCluster() {

        vl_gmm_delete(gmm);
}

cv::Mat FisherCluster::cluster(const cv::Mat &descriptors) {

        float *enc;

        // allocate space for the encoding
        enc = (float*)vl_malloc(sizeof(float) * encodingSize);

        // run fisher encoding
        vl_fisher_encode
         (enc, VL_TYPE_FLOAT,
         vl_gmm_get_means(gmm),
         dimCount, clusterCount,
         vl_gmm_get_covariances(gmm),
         vl_gmm_get_priors(gmm),
         descriptors.data, 1,
         VL_FISHER_FLAG_IMPROVED
         ) ;

        cv::Mat clustered(1, encodingSize, CV_32FC1, enc);

        vl_free(enc);

        return clustered;
}

void FisherCluster::train(const std::vector<cv::Mat> &descriptors) {
```

```
        vl_size numData = 0;
        for(const auto &mat : descriptors)
                numData += mat.rows;

        float *data = new float[numData * dimCount];

        long counter = 0;
        for(const auto &mat : descriptors) {
                memcpy((void*)(data + counter), (void*)mat.data, dimCount * mat.rows);
                counter += mat.rows;
        }

        gmm = vl_gmm_new(VL_TYPE_FLOAT, dimCount, clusterCount) ;

        vl_gmm_cluster (gmm, data, numData);

        delete [] data;
}

void FisherCluster::save() {
}

void FisherCluster::load() {
}

} /* namespace Tagger3D */


//
//void Controller::trainGMMRun() {
//
//    std::vector<cv::Mat> descriptors = io->loadVector<cv::Mat, float>(getParam<std::strin
g>(trainDescriptorsDest));
//
//    int all_rows = 0;
//    cv::Mat sum = cv::Mat::zeros(1, descriptors[0].cols, CV_32F);
//    // compute number ofrows
//    // summing in each dim (for computing EX after)
//    for(auto &img: descriptors){
//      all_rows += img.rows;
//      for(int row = 0; row<img.rows; row++){
//          const float* row_ptr = img.ptr<float>(row);
//          for(int col = 0; col < img.cols; ++col ){
//              sum.at<float>(col) += row_ptr[col];
//          }
//      }
//    }
//    cv::Mat ex = sum/all_rows;
//
//    // compute std
//    sum = cv::Mat::zeros(1, descriptors[0].cols, CV_32F);
//    for(auto &img: descriptors){
//      for(int row = 0; row<img.rows; row++){
//          const float* row_ptr = img.ptr<float>(row);
//          for(int col = 0; col < img.cols; ++col ){
//              auto el = row_ptr[col] - ex.ptr<float>(0)[col];
//              el = el*el;
//              sum.at<float>(col) += el;
//          }
//      }
//    }
//    cv::Mat std = sum/all_rows;
//    sqrt(std, std);
//
//    /* I can not do:
//     * io->saveMatBinary<float>(ex, EX_file);
//     * io->saveMatBinary<float>(vx, VX_file);
//     *
```

```
//     * so I do io operations on vectors:
//     */
//
//    std::vector<float> vec_ex;
//    ex.row(0).copyTo(vec_ex);
//    std::vector<float> vec_std;
//    std.row(0).copyTo(vec_std);
//    io->saveVector<float>(vec_ex, getParam<std::string>(gmmEXPath));
//    io->saveVector<float>(vec_std,  getParam<std::string>(gmmSTDPath));
//
//
//
//    // descriptors normalization
//    for(auto &img: descriptors){
//        img = (img - repeat(ex,  img.rows, 1));
//        img = (img / repeat(std,  img.rows, 1));
//    }
//
//    // initialize gmm parameters and fit data
//    vl_size const numData = all_rows;
//    vl_size const dimension = descriptors[0].cols;
//    vl_size const numClusters = getParam<int>(gmmNumClusters);
//    float *data = new float[dimension * numData];
//    int counter =0;
//    for(auto &img: descriptors){
//        for(int row = 0; row<img.rows; row++){
//            const float* row_ptr = img.ptr<float>(row);
//            for(int col = 0; col < img.cols; ++col ){
//                data[counter] = row_ptr[col];
//                counter++;
//            }
//        }
//    }
//
//    // clustering&saving&cleaning
//    VlGMM* gmm = vl_gmm_new (VL_TYPE_FLOAT, dimension, numClusters) ;
//    vl_gmm_cluster (gmm, data, numData);
//    io->saveGMMText(*gmm,  getParam<std::string>(gmmPath), dimension, numClusters);
//    vl_gmm_delete(gmm);
//    delete [] data;
//}
//
//void Controller::fisherEncodeRun(){
//    //std::vector<cv::Mat> descriptors = io->loadVector<cv::Mat, float>(getParam<std::str
ing>(testDescriptorsDest));
//    int dimension;
//    int numClusters;
//    VlGMM* gmm = io->loadGMMText(getParam<std::string>(gmmPath), dimension, numClusters);
//    std::cout<<"gmm ready"<<std::endl;
//    float* covariances = (float*)vl_gmm_get_covariances (gmm);
//    std::cout<<"covariance[0]: "<<covariances[0] <<std::endl;
//    std::vector<float> vec_ex = io->loadVector<float>(getParam<std::string>(gmmEXPath));
//    std::vector<float> vec_vx = io->loadVector<float>(getParam<std::string>(gmmSTDPath));
//    //cv::Mat ex = io->loadMatBinary<float>(EX_file);
//    //cv::Mat vx = io->loadMatBinary<float>(VX_file);
//    cv::Mat ex = cv::Mat(vec_ex);
//    cv::Mat vx = cv::Mat(vec_vx);
//    cv::transpose(ex,ex);
//    cv::transpose(vx,vx);
//    std::cout<<"vmax ready"<<std::endl;
//    std::cout<<"ex: "<< ex.rows << " " << ex.cols<<std::endl;
//
//    vector<std::string> variants;
//    variants.push_back (getParam<std::string>(testDescriptorsDest));
//    variants.push_back (getParam<std::string>(trainDescriptorsDest));
//
//    bool train = true;
//    for(auto &variant: variants){
```

```
//        std::vector<cv::Mat> descriptors = io->loadVector<cv::Mat, float>(variant);
//        std::string encoded_file;
//        if(train) encoded_file = getParam<std::string>(gmmFisherVectorsTrain);
//        else encoded_file = getParam<std::string>(gmmFisherVectorsTest);
//
//        //dataToEncode
//        //std::ofstream myfile;
//        //myfile.open (encoded_file);
//        io->initTextFile(encoded_file);
//        for(auto &test_img: descriptors){
//            test_img = (test_img - repeat(ex,  test_img.rows, 1));
//            test_img = (test_img / repeat(vx,  test_img.rows, 1));
//            float *dataToEncode = new float[dimension * test_img.rows];
//            int counter = 0;
//            for(int row = 0; row<test_img.rows; row++){
//                const float* row_ptr = test_img.ptr<float>(row);
//                for(int col = 0; col < test_img.cols; ++col ){
//                    dataToEncode[counter] = row_ptr[col];
//                    counter++;
//                }
//            }
//            vl_size const numDataToEncode = test_img.rows ;
//            // allocate space for the encoding
//            float* enc;
//            enc = (float *)vl_malloc(sizeof(float) * 2 * dimension * numClusters);
//            // run fisher encoding
//            vl_fisher_encode
//                (enc, VL_TYPE_FLOAT,
//                vl_gmm_get_means(gmm), dimension, numClusters,
//                vl_gmm_get_covariances(gmm),
//                vl_gmm_get_priors(gmm),
//                dataToEncode, numDataToEncode,
//                VL_FISHER_FLAG_IMPROVED
//                ) ;
//
//            //for(int i =0; i<2 * dimension * numClusters; ++i){
//                //myfile<<enc[i]<<" ";
//            //}
//            //myfile<<"\n";
//            std::vector<float> v(enc, enc + 2 * dimension * numClusters);
//            io->appendToTextFile(v);
//
//            vl_free(enc);
//            delete[] dataToEncode;
//
//        }
//        //myfile.close();
//        io->finalizeTextFile();
//        train = false;
//    }
//    vl_gmm_delete(gmm);
//}


/*
 * FisherCluster.h
 *
 *  Created on: 21 gru 2013
 *      Author: adam
 */

#ifndef FISHERCLUSTER_H_
#define FISHERCLUSTER_H_

#include <Cluster.h>

extern "C" {
        #include <vl/generic.h>
```

```
        #include <vl/fisher.h>
        #include <vl/gmm.h>

}

namespace Tagger3D {

class FisherCluster: public Cluster {
public:
        FisherCluster() = delete;
        FisherCluster(const std::map<std::string, std::string> &_configMap);
        virtual ~FisherCluster();

        virtual cv::Mat cluster(const cv::Mat &descriptors) override;
        virtual void train(const std::vector<cv::Mat> &descriptors) override;
        void save() override;
        void load() override;
        bool isLoaded() override { return loaded; } ;

private:
        VlGMM* gmm = nullptr;
        long encodingSize;
};

} /* namespace Tagger3D */

#endif /* FISHERCLUSTER_H_ */


/*
 * KMeansCluster.cpp
 *
 *  Created on:         20-06-2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "KMeansCluster.h"
#include "IoUtils.h"

#include <opencv2/opencv.hpp>
//#define NDEBUG
#include <assert.h>

namespace Tagger3D {

KMeansCluster::KMeansCluster(const std::map<std::string, std::string>& _configMap)
                : Cluster(_configMap){

        TRACE(logger,"configuring" );
        kMeans = nullptr;
        descriptorMatcher = nullptr;

        criteriaEps = getParam<float>(criteriaEpsKey);
        criteriaItr = getParam<int>(criteriaItrKey);
        attempts = getParam<int>(attemptsKey);
        flags = getParam<int>(flagsKey);
        matcherType = getParam<std::string>(matcherTypeKey);
        centroidIoName = getParam<std::string>(centroidIoNameKey);

        if( attempts == 0) {
                std::invalid_argument e("Attemps should be > 0");
                ERROR(logger, "Constructor: " << e.what() );
                throw e;
        }


        if( criteriaEps == 0 && criteriaItr == 0) {
```

```
                std::invalid_argument e("Invalid end criteria");
                ERROR(logger, "Constructor: " << e.what() );
                throw e;
        }

        createKMeans();
        createDescriptorMatcher();
}

void KMeansCluster::createKMeans() {

        TRACE(logger,"createKMeans: Starting" );
        cv::TermCriteria termCriteria;
        if(criteriaEps == 0) {

                termCriteria = cv::TermCriteria(cv::TermCriteria::COUNT, criteriaItr, crite
riaEps);
        } else if(criteriaItr == 0) {

                termCriteria = cv::TermCriteria(cv::TermCriteria::EPS, criteriaItr, criteri
aEps);
        } else {

                termCriteria = cv::TermCriteria(cv::TermCriteria::COUNT + cv::TermCriteria:
:EPS, criteriaItr, criteriaEps);
        }

        kMeans = std::unique_ptr<cv::BOWKMeansTrainer>(new cv::BOWKMeansTrainer(clusterCoun
t, termCriteria, attempts, flags));
        TRACE(logger,"createKMeans: Finished" );
}

void KMeansCluster::createDescriptorMatcher() {

        TRACE(logger,"createDescriptorMatcher: Starting" );
        descriptorMatcher = cv::DescriptorMatcher::create(matcherType);
        TRACE(logger,"createDescriptorMatcher: Finished" );
}

cv::Mat KMeansCluster::cluster(const cv::Mat &descriptors) {

        TRACE(logger,"cluster: Starting" );

        if( loaded == false) {

                std::logic_error e("KMeans has not been trained");
                ERROR(logger, "cluster: " << e.what());
                throw e;
        }

        std::vector<cv::DMatch> matches;
        descriptorMatcher->match(descriptors, centroids, matches);
        TRACE(logger,"cluster: Finished");
        return makeHistogram(matches);
}

void KMeansCluster::train(const std::vector<cv::Mat> &descriptors) {

        TRACE(logger,"train: Starting" );

        for(int i = 0; i < descriptors.size(); i++)
                kMeans->add(descriptors[i]);

        TRACE(logger,"train: Clustering" );
        centroids = kMeans->cluster();
        loaded = true;
```

```
        TRACE(logger,"train: Finished - centroids size = " << centroids.size() );
}

cv::Mat KMeansCluster::makeHistogram(const std::vector<int> &vec) const {

        cv::Mat hist = cv::Mat::zeros(1, clusterCount, CV_32FC1);
        auto *hPtr = hist.ptr<float>();
        const int *vPtr = &vec[0];
        int size = vec.size();
        for(int i = 0; i < size ;++i)
                hPtr[vPtr[i]] = hPtr[vPtr[i]] + 1;

        return hist;
}

cv::Mat KMeansCluster::makeHistogram(const std::vector<cv::DMatch> &matches) const {

        cv::Mat hist = cv::Mat::zeros(1, clusterCount, CV_32FC1);
        auto *hPtr = hist.ptr<float>();
        const cv::DMatch *mPtr = &matches[0];

        long matchesSize = matches.size();

        for(int i = 0; i < matchesSize; i++)
                        hPtr[mPtr[i].trainIdx] +=  1;

        return hist;
}

void KMeansCluster::save() {

        TRACE(logger, "save: Saving kMeans");
        IoUtils::getInstance()->saveMatBinary<float>(centroids, centroidIoName);
}

void KMeansCluster::load() {

        TRACE(logger, "load: Loading kMeans");
        centroids = IoUtils::getInstance()->loadMatBinary<float>(centroidIoName, clusterCou
nt, dimCount);
        loaded = true;
}

} /* namespace Tagger3D */



/*
 * KMeansCluster.h
 *
 *  Created on:         20-06-2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef KMEANSCLUSTER_H_
#define KMEANSCLUSTER_H_

#include "Cluster.h"

#include <opencv2/features2d/features2d.hpp>
#include <memory>

namespace Tagger3D {

/**
 * Class implements Cluster interface. It clasterizes data with KMeans algorithm.
 * Based on OpenCV.
```

```
 */
class KMeansCluster: public Cluster {
public:

        KMeansCluster() = delete;

        /**
         * Parametric constructor.
         * @controller          controller
         * @configFilePath      filepath to a configuration file
         */
        KMeansCluster(const std::map<std::string, std::string>& _configMap);

        /**
         * Default destructor.
         */
        ~KMeansCluster() = default;


        cv::Mat cluster(const cv::Mat &descriptors) override;
        void train(const std::vector<cv::Mat> &descriptors) override;
        void save() override;
        void load() override;
        bool isLoaded() override { return loaded; };

private:

        std::unique_ptr<cv::BOWKMeansTrainer> kMeans;
        cv::Ptr<cv::DescriptorMatcher> descriptorMatcher;
        cv::Mat centroids;
        cv::Mat makeHistogram(const std::vector<int> &vec) const;
        cv::Mat makeHistogram(const std::vector<cv::DMatch> &vec) const;

        /**
         * KMeans parameters.
         */

        int criteriaEps;
        int criteriaItr;
        int attempts;
        int flags;

        /**
         *  Allowed matcher types:
         *  BruteForce                  Uses L2 metrics
         *  BruteForce-L1
         *  BruteForce-Hamming
         *  BruteForce-Hamming(2)
         */
        std::string matcherType;
        std::string centroidIoName;

        /**
         * KMeans key values. Configuration parameters' values should be stored
         * in a configuration file as values assigned to the keys below.
         * The supported format is one key:value pair per line.
         */
        const std::string criteriaEpsKey = moduleName + "criteriaEps";
        const std::string criteriaItrKey = moduleName + "criteriaItr";
        const std::string attemptsKey = moduleName + "attempts";
        const std::string flagsKey = moduleName + "flags";
        const std::string matcherTypeKey = moduleName + "matcherType";
        const std::string centroidIoNameKey = moduleName + "centroidIoName";

        void createKMeans();
        void createDescriptorMatcher();
};
```

```
} /* namespace Tagger3D */
#endif /* KMEANSCLUSTER_H_ */


cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project( Tagger3D )

set(CMAKE_BUILD_TYPE Debug)
set(CMAKE_CXX_COMPILER_ARG1 -std=c++11)
set(CMAKE_ECLIPSE_VERSION 4.3)


# Project details--
set( Project ${CMAKE_PROJECT_NAME} )

set( Tagger3D_Version_Major 1 )
set( Tagger3D_Version_Minor 0 )
set( Tagger3D_Version_Subminor 0 )

# Configure a header file
configure_file( ${CMAKE_SOURCE_DIR}/CMakeScripts/CMakeSettings.h.cmake
                ${CMAKE_SOURCE_DIR}/CMakeSettings.h )

# Compiler Flags
message( STATUS "Setting GCC flags" )
set( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -lm ")#-O3 -march=native -mfpmath=sse -funroll-loo
ps -fopenmp" )
message( STATUS "CMAKE_CXX_FLAGS: ${CMAKE_CXX_FLAGS}" )

# Use some of our own Find* scripts
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/CMakeScripts" )
message( STATUS "CMAKE_MODULE_PATH: ${CMAKE_MODULE_PATH}" )

# PCL
find_package( PCL 1.7 COMPONENTS common features keypoints REQUIRED )
include_directories( ${PCL_INCLUDE_DIRS} )
link_directories( ${PCL_LIBRARY_DIRS} )
add_definitions( ${PCL_DEFINITIONS} )
set( LIBS ${LIBS} ${PCL_LIBRARIES} )
message( STATUS "PCL Include: ${PCL_INCLUDE_DIRS}" )
message( STATUS "PCL Libraries: ${PCL_LIBRARY_DIRS}" )
message( STATUS "PCL Libraries: ${PCL_LIBRARIES}" )

#       Boost
set( Boost_USE_MULTITHREADED ON )
find_package( Boost 1.55 COMPONENTS program_options filesystem system REQUIRED )
include_directories( ${Boost_INCLUDE_DIR} )
link_directories( ${Boost_LIBRARY_DIRS} )
set( LIBS ${LIBS} ${Boost_LIBRARIES} )
message( STATUS "Boost Include: ${Boost_INCLUDE_DIR}" )
message( STATUS "Boost Libraries: ${Boost_LIBRARY_DIRS}" )
message( STATUS "Boost Libraries: ${Boost_LIBRARIES}" )

# OpenCV
find_package( OpenCV 2.4.6 COMPONENTS core highgui features2d imgproc ml REQUIRED)
set( LIBS ${LIBS} ${OpenCV_LIBRARIES} )
message( STATUS "OpenCV Include: ${OpenCV_INCLUDE_DIRS}" )
message( STATUS "OpenCV Libraries: ${OpenCV_LIBRARY_DIRS}" )
message( STATUS "OpenCV Libraries: ${OpenCV_LIBRARIES}" )

#       LIBSVM
find_package( LIBSVM REQUIRED )
include_directories( ${LIBSVM_INCLUDE_DIRS})
set( LIBS ${LIBS} ${LIBSVM_LIBRARIES} )

# vlfeat
message(STATUS "vlfeat ${CMAKE_SOURCE_DIR}/../lib/include")
include_directories("${CMAKE_SOURCE_DIR}/../lib/include")
```

```
set(vlfeat_LIBS "${CMAKE_SOURCE_DIR}/../lib/vlfeat/libvl.so")
set(LIBS ${LIBS} ${vlfeat_LIBS})

#       log4cxx
find_package( log4cxx REQUIRED )
include_directories( ${LOG4CXX_INCLUDE_DIRS} )
set( LIBS ${LIBS} ${LOG4CXX_LIBRARIES} )
message( STATUS "LOG4CXX_INCLUDE_DIRS: ${LOG4CXX_INCLUDE_DIRS}" )
message( STATUS "LOG4CXX_LIBRARIES: ${LOG4CXX_LIBRARIES}" )

FILE(GLOB_RECURSE sources *.cpp)
FILE(GLOB_RECURSE headers *.h)
SET(dir_list "")
FOREACH(file_path ${headers})
        GET_FILENAME_COMPONENT(dir_path ${file_path} PATH)
        SET(dir_list ${dir_list} ${dir_path})
ENDFOREACH()
LIST(REMOVE_DUPLICATES dir_list)
include_directories(${dir_list})
message( STATUS "Dirs: ${dir_list}" )

 # Exec list
add_executable ( ${Project} ${sources})

message( STATUS "Libs: ${LIBS}" )
target_link_libraries ( ${Project} ${LIBS} )




/*
 * CMakeSettings.h
 *
 *  Created on: 26 Jul 2013
 *      Author: Adam Kosiorek
 */

#ifndef CMAKESETTINGS_H_
#define CMAKESETTINGS_H_

#define PROJECT_NAME "Tagger3D"
#define VERSION_MAJOR 1
#define VERSION_MINOR 0
#define VERSION_PATCH 0
#define VERSION "Tagger3D Version: 1.00"


#endif /* CMAKESETTINGS_H_ */


/home/adam/workspace/Tagger3D/utils/src/Common


/*
 * Tagger3D : clouds.h
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef CLOUDS_H_
#define CLOUDS_H_

#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <vector>
```

```
/**
 *      Cloud types
 */
typedef pcl::PointCloud<pcl::PointWithScale> ScaleCloud;
typedef pcl::PointCloud<pcl::PointXYZRGB> ColorCloud;
typedef pcl::PointCloud<pcl::PFHSignature125> PfhCloud;
typedef pcl::PointCloud<pcl::FPFHSignature33> FpfhCloud;
typedef pcl::PointCloud<pcl::PFHRGBSignature250> PfhRgbCloud;
typedef pcl::PointCloud<pcl::Normal> NormalCloud;

/**
 * Cloud vectors
 */
typedef std::vector<ScaleCloud::Ptr> ScaleVec;
typedef std::vector<ColorCloud::Ptr> ColorVec;
typedef std::vector<PfhCloud::Ptr> PfhVec;
typedef std::vector<FpfhCloud::Ptr> FpfhVec;
typedef std::vector<PfhRgbCloud::Ptr> PfhRgbVec;
typedef std::vector<NormalCloud::Ptr> NormalVec;

#endif /* CLOUDS_H_ */


/*
 * Factory.cpp
 *
 *  Created on: 14 gru 2013
 *      Author: adam
 */

#include "Factory.h"

#include "RangeImgReader.h"
#include "PcdReader.h"
#include "NormalEstimator.h"
#include "Detector.h"
#include "SIFTDetector.h"
#include "Iss3dDetector.h"
#include "DenseDetector.h"

#include "PFHDescriptor.h"
#include "FPFHDescriptor.h"
#include "PFHRGBDescriptor.h"
#include "ShotDescriptor.h"
#include "ShotColorDescriptor.h"

#include "KMeansCluster.h"
#include "LibSVMPredictor.h"
#include "CvSVMPredictor.h"

#include <stdexcept>

namespace Tagger3D {

Factory::Factory(const std::map<std::string, std::string> &_configMap)
: ProcessObject(_configMap),
  configMap(_configMap) {

        logger = lgr::Logger::getLogger(loggerName);
        DEBUG(logger, "Creating Tagger3D");
}

std::unique_ptr<ImgReader> Factory::getReader() const {

        std::unique_ptr<ImgReader> imgReader;
        switch(getParam<int>(readerType)) {
        case Dataset::TOKYO: imgReader = std::unique_ptr<ImgReader>(new RangeImgReader(conf
igMap)); break;
```

```
        case Dataset::B3DO: imgReader = std::unique_ptr<ImgReader>(new PcdReader(configMap)
); break;
        default:
                std::runtime_error e("Invalid reader type");
                ERROR(logger, e.what());
                throw e;
        }
        return imgReader;
}

std::unique_ptr<PointNormal> Factory::getPointNormal() const {

        return std::unique_ptr<PointNormal> (new NormalEstimator(configMap));
}

std::unique_ptr<Detector> Factory::getDetector() const {

        std::unique_ptr<Detector> detector;
        switch( getParam<int>(detectorType)) {
        case DetectorType::SIFT: detector = std::unique_ptr<Detector> (new SIFTDetector(con
figMap)); break;
        case DetectorType::ISS3D: detector = std::unique_ptr<Detector> (new Iss3dDetector(c
onfigMap)); break;
        case DetectorType::DENSE: detector = std::unique_ptr<Detector> (new DenseDetector(c
onfigMap)); break;
        default:
                        std::runtime_error e("Invalid detector type");
                        ERROR(logger, e.what());
                        throw e;
                }
        return detector;
}

std::unique_ptr<Descriptor> Factory::getDescriptor() const {
        std::unique_ptr<Descriptor> descriptor;
        switch( getParam<int>( descType )) {
        case DescriptorType::PFH: descriptor = std::unique_ptr<Descriptor> (new PFHDescript
or(configMap)); break;
        case DescriptorType::FPFH: descriptor = std::unique_ptr<Descriptor> (new FPFHDescri
ptor(configMap)); break;
        case DescriptorType::PFHRGB: descriptor = std::unique_ptr<Descriptor> (new PFHRGBDe
scriptor(configMap)); break;
//      case DescriptorType::PFHRGB: descriptor = std::unique_ptr<Descriptor> (new PFHRGBgp
u(configMap)); break;
        case DescriptorType::SHOT: descriptor = std::unique_ptr<Descriptor> (new ShotDescri
ptor(configMap)); break;
        case DescriptorType::SHOTCOLOR: descriptor = std::unique_ptr<Descriptor> (new ShotC
olorDescriptor(configMap)); break;
        default:
                std::runtime_error e("Invalid descriptor type");
                ERROR(logger, e.what());
                throw e;
        }
        return descriptor;
}

std::unique_ptr<Cluster> Factory::getCluster() const {

        return std::unique_ptr<Cluster> (new KMeansCluster(configMap));
}

std::unique_ptr<Predictor> Factory::getPredictor() const {

        std::unique_ptr<Predictor> predictor;
        switch( getParam<int>( predictorType )) {
        case PredictorType::SVM_LIB: predictor = std::unique_ptr<Predictor> (new LibSVMPred
ictor(configMap)); break;
        case PredictorType::SVM_CV: predictor = std::unique_ptr<Predictor> (new CvSVMPredic
```

```
tor(configMap)); break;
        default:
                        std::runtime_error e("Invalid predictor type");
                        ERROR(logger, e.what());
                        throw e;
        }
        return predictor;
}

} //namespace Tagger3D



/*
 * Factory.h
 *
 *  Created on: 14 gru 2013
 *      Author: adam
 */

#ifndef FACTORY_H_
#define FACTORY_H_

#include "ProcessObject.h"

#include "ImgReader.h"
#include "PointNormal.h"
#include "Detector.h"
#include "Descriptor.h"
#include "Cluster.h"
#include "Predictor.h"

#include <memory>

namespace Tagger3D {

class Factory: public ProcessObject {
public:
        Factory() = delete;
        Factory(const std::map<std::string, std::string> &_configMap);
        virtual ~Factory() = default;

        std::unique_ptr<ImgReader> getReader() const;
        std::unique_ptr<PointNormal> getPointNormal() const;
        std::unique_ptr<Detector> getDetector() const;
        std::unique_ptr<Descriptor> getDescriptor() const;
        std::unique_ptr<Cluster> getCluster() const;
        std::unique_ptr<Predictor> getPredictor() const;

private:

        const std::map<std::string, std::string> &configMap;

        const std::string readerType = "readerType";
        const std::string detectorType = "detectorType";
        const std::string descType = "descType";
        const std::string predictorType = "predictorType";

        const std::string loggerName = "Main.Factory";

        enum Dataset { B3DO, TOKYO };
        enum DetectorType { SIFT, ISS3D, DENSE };
        enum DescriptorType { PFH, FPFH, PFHRGB, SHOT, SHOTCOLOR };
        enum PredictorType { SVM_LIB, SVM_CV };
};

} // namespace Tagger3D
```

```
#endif /* FACTORY_H_ */


/*
 * logger.h
 *
 *              Created on: 24 Jul 2013
 *              Author: Adam Kosiorek
 *              Description:    The header contains macros simplyfing the usage of LOG4CXX
 */

#ifndef LOGGER_H_
#define LOGGER_H_

#include <log4cxx/logger.h>

#define TRACE(a, b) LOG4CXX_TRACE(a, b)
#define DEBUG(a, b) LOG4CXX_DEBUG(a, b)
#define INFO(a, b) LOG4CXX_INFO(a, b)
#define WARN(a, b) LOG4CXX_WARN(a, b)
#define ERROR(a, b) LOG4CXX_ERROR(a, b)
#define FATAL(a, b) LOG4CXX_FATAL(a, b)

namespace lgr = log4cxx;

#endif /* LOGGER_H_ */


/*
 * Tagger3D : ProcessObject.cpp
 *
 *  Created on:         27 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "ProcessObject.h"
#include <fstream>

namespace Tagger3D {

ProcessObject::ProcessObject(const std::map<std::string, std::string> &_configMap) : config
Map(_configMap) {

        if( _configMap.empty() ) {

                throw std::invalid_argument("Empty configuration map");
        }

        directory = getParam<std::string>( directoryKey);
}

ProcessObject::~ProcessObject() {}

bool ProcessObject::checkConfig(const std::string &key) const{

        if( !configMap.count(key) || configMap.find(key)->second.length() == 0) {

                return false;
        }

        return true;
}

std::string ProcessObject::getParam(const std::string &key ) {

        if( !checkConfig( key )) {
```

```
                throw std::runtime_error("No parameter " + key + " in the configMap");
        }

        return configMap.find(key)->second;
}

} /* namespace Tagger3D */


/*
 * Tagger3D : ProcessObject.h
 *
 *  Created on:         27 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef PROCESSOBJECT_H_
#define PROCESSOBJECT_H_

#include "logger.h"

#include <string>
#include <map>
#include <vector>

#include <sstream>
#include <typeinfo>
#include <stdio.h>
#include <stdexcept>
#include <iostream>

#include "Utils.h"

namespace Tagger3D {

class ProcessObject {
public:
        ProcessObject(const std::map<std::string, std::string> &_configMap);
        virtual ~ProcessObject();

protected:
        template<typename T>
        T getParam(const std::string &key) const;
        std::string getParam(const std::string &key);
        bool fileExists(const std::string &path);

        const std::string separator = ".";
        lgr::LoggerPtr logger;
        std::string directory;
        const std::string mode = "mode";


private:
        ProcessObject();
        bool checkConfig(const std::string &key) const;
        template<typename T>
        T stringToNumber(const std::string &s, T def = T() ) const;

        std::map<std::string, std::string> configMap;


        const std::string directoryKey = "directory";
};

template<typename T>
T ProcessObject::getParam(const std::string &key) const {
```

```
        if( !checkConfig( key )) {

                throw std::runtime_error("No parameter " + key + " in the configMap");
        }

        std::string value = configMap.find(key)->second;
        return stringToNumber<T>(value);
}

template<typename T>
T ProcessObject::stringToNumber(const std::string &s, T def) const {

        std::stringstream ss(s);
        T result;
        return ss >> result ? result : def;
}

} /* namespace Tagger3D */
#endif /* PROCESSOBJECT_H_ */




/*
 * Tagger3D.cpp
 *
 *   Created on: 23 sie 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#include "Tagger3D.h"
#include "Factory.h"
#include "IoUtils.h"

namespace Tagger3D {

Tagger3D::Tagger3D(const std::map<std::string, std::string> &configMap) : ProcessObject(con
figMap) {

        logger = lgr::Logger::getLogger(loggerName);
        DEBUG(logger, "Creating Tagger3D");

        io = IoUtils::getInstance();
        io->setPath(directory);

        Factory f(configMap);
        imgReader = f.getReader();
        pointNormal = f.getPointNormal();
        detector = f.getDetector();
        descriptor = f.getDescriptor();
        cluster = f.getCluster();
        predictor = f.getPredictor();
}

Tagger3D::~Tagger3D() {
        DEBUG(logger, "Destroying Tagger3D");
}

int Tagger3D::predict(const std::string& rgbPath,
                const std::string& depthPath) {

        cluster->load();
        predictor->load();

        ColorCloud::Ptr colorCloud;// = imgReader->readImg(rgbPath, depthPath);
        NormalCloud::Ptr normalCloud = pointNormal->computeNormals( colorCloud );
        ScaleCloud::Ptr keypointCloud = detector->detect( colorCloud );
```

```
        cv::Mat descriptors = descriptor->describe( colorCloud, keypointCloud, normalCloud)
;

        cv::Mat wordDescriptors = cluster->cluster(descriptors);
        return getCatNum(predictor->predict(wordDescriptors));
}

std::vector<cv::Mat> Tagger3D::computeDescriptors() {
        INFO(logger, "Computing descriptors");

        std::vector<cv::Mat> descriptors;

        ColorCloud::Ptr colorCloud;
        NormalCloud::Ptr normalCloud;
        ScaleCloud::Ptr keypointCloud;
        cv::Mat descMat;
        int counter = 0;
        colorCloud = imgReader->readImg();

         while(!colorCloud->empty()) {

                TRACE(logger, "points: " << colorCloud->size())

                normalCloud = pointNormal->computeNormals(colorCloud);
                pointNormal->cleanupInputCloud(colorCloud);
                TRACE(logger, "after cleanup: " << colorCloud->size());

                keypointCloud = detector->detect(colorCloud);
                TRACE(logger, "keypoints: " << keypointCloud->size());

                if(keypointCloud->size() == 0) {

                        std::runtime_error e("Couldn't find any keypoints in a pointcloud #
" + std::to_string(counter));
                        ERROR(logger, "computeDescriptors: " << e.what());
                        throw e;
                }

                descMat = descriptor->describe(colorCloud, keypointCloud, normalCloud);
                descriptors.push_back(descMat.clone());
                colorCloud = imgReader->readImg();
                counter++;
        }

        return descriptors;
}


void Tagger3D::descRun() {
        INFO(logger, "Compute Descriptors Run");

        imgReader->setMode(ImgReader::TRAIN);
        std::vector<cv::Mat> descriptors = computeDescriptors();
        io->saveVector<cv::Mat, float>(descriptors, trainDescName);

        imgReader->setMode(ImgReader::TEST);
        descriptors = computeDescriptors();
        io->saveVector<cv::Mat, float>(descriptors, testDescName);
}

void Tagger3D::clustRun() {

        std::vector<cv::Mat> descriptors = io->loadVector<cv::Mat, float>(trainDescName);
        cluster->train(descriptors);
        cluster->save();
}

void Tagger3D::trainRun() {
```

```
	INFO(logger, "Train Predictor Run");

	bool storeHistogram = getParam<bool>(storeHistogramKey);

	std::vector<cv::Mat> descriptors = io->loadVector<cv::Mat, float>(trainDescName);
	imgReader->setMode(ImgReader::TRAIN);
	std::vector<int> labels = imgReader->readLabels();
	cluster->load();
	cv::Mat wordDescriptors = cluster->cluster(descriptors);

	wordDescriptors.convertTo(wordDescriptors, CV_32FC1);
	if(storeHistogram) {

		io->initTextFile(trainHistogram);
		for(int i = 0; i < wordDescriptors.rows; ++i) {
			io->appendToTextFile<int>(labels[i]);
			io->appendToTextFile<float>(wordDescriptors.row(i));
			TRACE(logger, "trainHistogram #" << i);
		}
		io->finalizeTextFile();
	}

	predictor->train(wordDescriptors, labels);
	predictor->save();
}

void Tagger3D::predRun() {
	INFO(logger, "Test Run");

	bool storeHistogram = getParam<bool>(storeHistogramKey);

	std::vector<cv::Mat> descriptors = io->loadVector<cv::Mat, float>(testDescName);
	imgReader->setMode(ImgReader::TEST);
	std::vector<int> labels = imgReader->readLabels();

	cluster->load();
	predictor->load();

	cv::Mat wordDescriptors = cluster->cluster(descriptors);

	if(storeHistogram) {

		io->initTextFile(testHistogram);
		for(int i = 0; i < wordDescriptors.rows; ++i) {
			io->appendToTextFile<int>(labels[i]);
			io->appendToTextFile<float>(wordDescriptors.row(i));
		}
		io->finalizeTextFile();
	}

	long correct = 0;
	for(int i = 0; i < wordDescriptors.rows; ++i) {

		std::vector<float> prediction = predictor->predict(wordDescriptors.row(i));
		int cat = getCatNum(prediction);
//			std::cout << cat << "\t";
//		for(auto p : prediction)
//			std::cout << p << " ";
//		std::cout << std::endl;
		if(cat == labels[i]) correct++;
	}

	std::cout << "Average: " << float(correct) / labels.size() * 100 << "%" << std::end
l;
}

void Tagger3D::run() {
	INFO(logger, "Tagger3D running");
```

```
        switch(getRunMode()) {

        case Mode::DESC: descRun(); break;
        case Mode::CLUST: clustRun(); break;
        case Mode::TRAIN: trainRun(); break;
        case Mode::PRED: predRun(); break;
        }
}

int Tagger3D::getRunMode() {

        std::string m = getParam<std::string>( mode );
        auto it = std::find(std::begin(modeStrings), std::end(modeStrings), m);
        if(it == std::end(modeStrings)) {
                std::runtime_error e("Invalid mode: " + mode);
                ERROR(logger, e.what());
                throw e;
        }
        return std::distance(std::begin(modeStrings), it);
}

int Tagger3D::getCatNum(const std::vector<float> &vec) const {

        auto it = std::max_element(std::begin(vec), std::end(vec));
        return std::distance(std::begin(vec), it);
}

} /* namespace Tagger3D */


/*
 * Tagger3D.h
 *
 *   Created on: 23 sie 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#ifndef TAGGER3D_H_
#define TAGGER3D_H_

#include "ProcessObject.h"
#include "ImgReader.h"
#include "PointNormal.h"
#include "Detector.h"
#include "Descriptor.h"
#include "Cluster.h"
#include "Predictor.h"
#include "IoUtils.h"
#include <memory>

namespace Tagger3D {

/**
 * Tagger3D is an object categorization.
 * Provided an RGB-D input it predicts a category of an object.
 */
class Tagger3D: public ProcessObject {

        //      ------------------------------------------------------------------------
        //      Methods -----------------------------------------------------------------
public:
        Tagger3D() = delete;
        Tagger3D(const std::map<std::string, std::string> &configMap);
        virtual ~Tagger3D();

        /**
```

```
         * Performs model training
         * @param       void
         * @return      void
         */
        void train();

        /**
         * Performs batch prediction in order to evaluate an average accuracy
         * @param void
         * @return void
         */
        void test();

        /**
         * Predicts a category membership of a provided tuple of rgb and depth images
         * @param       rgbPath path to an rgb image
         * @param       depthPath       path to a depth image
         * @return      an int value being a category number
         */
        int predict(const std::string &rgbPath, const std::string &depthPath);

        /**
         * Launched one of the train, test or predict mode based on the config settings
         * @param none
         * @return an int number of a category in case the prediction mode was launched
         */
        void run();

        //      Separate run options ------------------------------------------------
        void descRun();
        void clustRun();
        void trainRun();
        void predRun();

private:
        std::vector<cv::Mat> computeDescriptors();
        int getRunMode();
        int getCatNum(const std::vector<float> &vec) const;

        //      ---------------------------------------------------------------------
        // Fields       ---------------------------------------------------------------
public:
private:
        // Pointers     ---------------------------------------------------------------
        std::unique_ptr<ImgReader> imgReader;
        std::unique_ptr<PointNormal> pointNormal;
        std::unique_ptr<Detector> detector;
        std::unique_ptr<Descriptor> descriptor;
        std::unique_ptr<Cluster> cluster;
        std::unique_ptr<Predictor> predictor;
        std::shared_ptr<IoUtils> io;

        // Constants    ---------------------------------------------------------------
        const std::string loggerName = "Tagger3D";
        const std::string moduleName = "Tagger3D" + separator;

        // Config keys  ---------------------------------------------------------------
        const std::string trainDescName = "trainDescriptors";
        const std::string testDescName = "testDescriptors";
        const std::string trainHistogram = "trainHistogram";
        const std::string testHistogram = "testHistogram";

        const std::string storeHistogramKey = "storeHistogram";

        //      Modes   ------------------------------------------------------------------
        const std::vector<std::string> modeStrings = {"desc", "clust", "train", "pred"};

        // Enums ----------------------------------------------------------------
```

```
        enum Mode { DESC, CLUST, TRAIN, PRED };
};

} /* namespace Tagger3D */
#endif /* TAGGER3D_H_ */
```

/home/adam/workspace/Tagger3D/utils/src/Config

```
/*
 * Tagger3D : Config.cpp
 *
 *  Created on:         22 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:    Implementation of a config file parser.
 */

#include "Config.h"

#include <boost/program_options/options_description.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/variables_map.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/detail/config_file.hpp>

#include <boost/algorithm/string.hpp>

#include <set>
#include <exception>
#include <fstream>
#include <iostream>

namespace Tagger3D {
namespace pod = boost::program_options::detail;
namespace po = boost::program_options;

Config::Config(const int _argc, const char **_argv) : argc(_argc), argv(_argv) {

        logger = lgr::Logger::getLogger( loggerName );
        DEBUG(logger, "Creating Config");
}

Config::~Config() {

        DEBUG(logger, "Destroying Config");
}

std::map<std::string, std::string> Config::parseConfigFile(std::string filePath) {

        TRACE(logger, "parseConfigFile: Starting");
        std::map<std::string,std::string> confMap;
        std::ifstream config(filePath);

        if(!config)
        {
                std::runtime_error e("Could not open the config file: " + filePath);
                ERROR(logger, "parseConfigFile: " << e.what() );
                throw e;
        }

        //parameters
        std::set<std::string> options;
        std::map<std::string, std::string> parameters;
        options.insert("*");

        try
        {
```

```
                for (pod::config_file_iterator i(config, options), e ; i != e; ++i)
                {
                        confMap[i->string_key] = i->value[0];
                        parameters[i->string_key] = i->value[0];
                }
        }
        catch(std::exception& e)
        {
                std::runtime_error er( e.what() );
                ERROR(logger, "parseConfigFile: " << er.what() );
                throw er;
        }

        TRACE(logger, "parseConfigFile: Finished");
        return confMap;
}

std::map<std::string, std::string> Config::getConfigMap() {

        TRACE(logger, "getConfigMap: Starting");

        std::map<std::string,std::string> confMap;
        std::string helpTmp = help + "," + help[0];
        std::string versionTmp = version + "," + version[0];
        std::string pictureTmp = picture + "," + picture[0];
        std::string modeTmp = mode + "," + mode[0];

        /**
         *      Options allowed only on a command line.
         */
        po::options_description genericOps("Generic options");
        genericOps.add_options()
                        (versionTmp.c_str(), "Print version")
                        (helpTmp.c_str(), "Produce help message")
                        ("readerType", po::value<std::string>(), "0 - rangeImg 1 - PCD")
                        ("detectorType", po::value<std::string>(), "0 - SIFT 1 - ISS3D")
                        ("descType", po::value<std::string>(), "0 - PFH 1 -FPFH")
                        ("predictorType", po::value<std::string>(), "")
                        ("dimCount", po::value<std::string>(), "")
                        ("dictionarySize", po::value<std::string>(), "")
                        ("resize", po::value<std::string>(), "resize")
                        ;

        /**
         *      Options allowed on a command line and in a config file.
         */
        po::options_description readerOps("Reader options");
                readerOps.add_options()
                        ("leafSize", po::value<std::string>(), "")
                        ;

        po::options_description detectorOps("Detector options");
                detectorOps.add_options()
                        ("gamma21", po::value<std::string>(), "")
                        ("gamma32", po::value<std::string>(), "")
                        ("minNeighbours", po::value<std::string>(), "")
                        ("modelResolution", po::value<std::string>(), "")
                        ;

        po::options_description normalOps("Normal options");
                normalOps.add_options()
                        ("normalRadius", po::value<std::string>(), "")
                        ;

        po::options_description descriptorOps("Descriptor options");
                descriptorOps.add_options()
                        ("radiusSearch", po::value<std::string>(), "")
                        ;
```

```
        po::options_description clusterOps("Clusterization options");
                clusterOps.add_options()
                        ("dictionarySize", po::value<std::string>(), "Number of clusters")
                        ("criteriaEps", po::value<std::string>(), "Required precision")
                        ("criteriaItr", po::value<std::string>(), "Maximum number of iterat
ions")
                        ("trainCluster", po::value<std::string>(), "train cluster if 1")
                        ;


        po::options_description predictorOps("Predictor options");
        predictorOps.add_options()
                ("alpha", po::value<std::string>(), "Alpha value")
                ("numTopics", po::value<std::string>(), "Number of topics")
                ("initMethod", po::value<std::string>(), "An initialization method")
                ("sldaSettings", po::value<std::string>(), "sLDA settings file")
                ("sldaOutputDir", po::value<std::string>(), "A sLDA output directory")
                ("modelPath", po::value<std::string>(), "A path to a *.model file for sLDA"
)
                ("epsilon", po::value<std::string>(), "")
                ("maxIter", po::value<std::string>(), "")
                ("degree", po::value<std::string>(), "")
                ("gamma", po::value<std::string>(), "")
                ("C", po::value<std::string>(), "")
                ;


        po::options_description configOps("Configuration");
        configOps.add_options()
                (modeTmp.c_str(), po::value<std::string>(), "Mode")
                (directory.c_str(), po::value<std::string>(), "Model directory")
                (pictureTmp.c_str(), po::value<std::string>(), "Path to a picture")
                ;
        configOps.add(readerOps).add(descriptorOps).add(normalOps).add(detectorOps).add(clu
sterOps).add(predictorOps);

        /**
         *      Hidden options. Allowed on a command line and in a config file.
         *      These options are concealed from the user.
         */
        po::options_description hiddenOps("Hidden options");
        hiddenOps.add_options()
                (config.c_str(), po::value<std::string>(), "Configuration file")
                ;


        po::options_description cmdOptions;
        cmdOptions.add(genericOps).add(configOps).add(hiddenOps);

        po::options_description visibleOptions("Allowed options:");
        visibleOptions.add(genericOps).add(configOps);

        po::positional_options_description positionalOptions;
        positionalOptions.add(config.c_str(), 1).add(mode.c_str(), 1).add(directory.c_str()
, 1);

         po::variables_map vm;

         try{
                 store(po::command_line_parser(argc, argv)
                         .options(cmdOptions).positional(positionalOptions).run(), vm);
         } catch(std::exception &e) {

                 std::runtime_error er( e.what() );
                 ERROR(logger, "getConfigFile: " << er.what() );
                 throw er;
         }
```

```
        if( vm.count(help)) {

                std::cout << visibleOptions;
                return confMap;
        }

        if( vm.count(version) ) {

                std::cout << "Version xxx";
                return confMap;
        }

        if( !vm.count( config ) || (vm[config].as<std::string>()).size() == 0 ) {

                std::runtime_error e( "No config file has been specified. Please specify a
 config file.");
                ERROR(logger, "getConfigFile: " << e.what() );
                throw e;
        }



       std::string filePath = vm[config].as<std::string>();
       confMap = parseConfigFile(filePath);
       for(std::map<std::string, std::string>::iterator it = confMap.begin(); it != confMa
p.end(); ++it ) {

                std::string key = it->first;
                std::vector<std::string> splitted;
                boost::split(splitted, key, boost::is_any_of("."));
                key = splitted.back();
                if( vm.count( key ) ) {

                        it->second = vm[ key ].as< std::string >();
                }

                // Print confMap
                TRACE(logger, it->first << " = " << it->second );
       }

       TRACE(logger, "getConfigMap: Finished");
       return confMap;
}



} /* namespace Tagger3D */


/*
 * Tagger3D : Config.h
 *
 *  Created on:         22 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:    A config file parser.
 */

#ifndef CONFIG_H_
#define CONFIG_H_

#include "../Common/logger.h"

#include <boost/program_options/variables_map.hpp>

namespace Tagger3D {

class Config {
```

```
public:

        Config(const int _argc, const char **_argv);
        virtual ~Config();

        /**
         *      Prepares and returns a config map.
         *      @retun  std::map<std::string, std::string> containing pairs of key=valueget
ConfigMap
         */
        std::map<std::string,std::string> getConfigMap();

private:

        int argc;
        const char** argv;

        const std::string config = "config";
        const std::string help = "help";
        const std::string version = "version";
        const std::string picture = "picture";
        const std::string mode = "mode";
        const std::string directory = "directory";

        std::map<std::string,std::string> parseConfigFile(std::string filePath);

        const std::string loggerName = "Main.Config";
        lgr::LoggerPtr logger;
};

} /* namespace Tagger3D */
#endif /* CONFIG_H_ */


/home/adam/workspace/Tagger3D/utils/src/Descriptor


/home/adam/workspace/Tagger3D/utils/src/Descriptor/CloudParser


/*
 * Tagger3D : CloudParser.h
 *
 *  Created on:         31 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:    Class for parsing different types of point clouds to cv::Mat
 */

#ifndef CLOUDPARSER_H_
#define CLOUDPARSER_H_

#include "../../Common/logger.h"
#include "../../Common/clouds.h"
#include "PfhTraits.h"

#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <opencv2/core/core.hpp>

#include <vector>

namespace Tagger3D {

class CloudParser {

public:
        CloudParser();
        virtual ~CloudParser();
```

```
        template<typename T>
        static cv::Mat parse(const typename T::Ptr &cloud);

        template<typename T>
        static std::vector<cv::Mat> parse(const std::vector<typename T::Ptr> &cloud);

};

template<typename T>
inline cv::Mat CloudParser::parse(const typename T::Ptr& cloud) {

        int size = cloud->size();
        int matSize = PfhTraits<typename T::PointType>::size();
        cv::Mat mat( size, matSize, CV_32FC1 );
        for(int i = 0; i < size; i++) {
                float* dPtr = mat.ptr<float>(i);
                float* hPtr = cloud->points[i].histogram;
                for(int j = 0; j < matSize; j++)
                        dPtr[j] = hPtr[j];
        }
        return mat;
}

template<>
inline cv::Mat CloudParser::parse<pcl::PointCloud<pcl::SHOT352>>(const pcl::PointCloud<pcl:
:SHOT352>::Ptr& cloud) {

        int size = cloud->size();
        static int matSize = 352;
        cv::Mat mat( size, matSize, CV_32FC1 );
        for(int i = 0; i < size; i++) {
                float* dPtr = mat.ptr<float>(i);
                float* hPtr = cloud->points[i].descriptor;
                for(int j = 0; j < matSize; j++)
                        dPtr[j] = hPtr[j];
        }
        return mat;
}

template<>
inline cv::Mat CloudParser::parse<pcl::PointCloud<pcl::SHOT1344>>(const pcl::PointCloud<pcl
::SHOT1344>::Ptr& cloud) {

        int size = cloud->size();
        static int matSize = 1344;
        cv::Mat mat( size, matSize, CV_32FC1 );
        for(int i = 0; i < size; i++) {
                float* dPtr = mat.ptr<float>(i);
                float* hPtr = cloud->points[i].descriptor;
                for(int j = 0; j < matSize; j++)
                        dPtr[j] = hPtr[j];
        }
        return mat;
}

template<typename T>
inline std::vector<cv::Mat> CloudParser::parse(
                const std::vector<typename T::Ptr>& clouds) {

        std::vector<cv::Mat> vec;
        for(const auto &cloud : clouds) {

                vec.push_back( parse<T>( cloud ) );
        }
        return vec;
}
```

```
} /* namespace Tagger3D */
#endif /* CLOUDPARSER_H_ */


/*
 * PfhTraits.h
 *
 *   Created on: 11 paÅº 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#ifndef PFHTRAITS_H_
#define PFHTRAITS_H_

template<typename T>
struct PfhTraits {
        static int size() { return -1; };
        typedef int type;
};

template<> struct PfhTraits<pcl::FPFHSignature33>{ static int size() { return 33; }; };
template<> struct PfhTraits<pcl::PFHSignature125>{ static int size() { return 125; }; };
template<> struct PfhTraits<pcl::PFHRGBSignature250>{ static int size() { return 250; }; };
template<> struct PfhTraits<pcl::SHOT352>{ static int size() { return 352; }; };
template<> struct PfhTraits<pcl::SHOT1344>{ static int size() { return 1344; }; };

#endif /* PFHTRAITS_H_ */


/*
 * Tagger3D : Descriptor.cpp
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "Descriptor.h"

namespace Tagger3D {

Descriptor::Descriptor(const std::map<std::string, std::string> &configMap) : ProcessObject
(configMap) {

        logger = lgr::Logger::getLogger( loggerName );
        DEBUG(logger, "Creating Descriptor");
}

Descriptor::~Descriptor() {

        DEBUG(logger, "Destroying Descriptor");
}

std::vector<cv::Mat> Descriptor::describe(const ColorVec &clouds, const ScaleVec &keyClouds
, const NormalVec &normalClouds) {

        TRACE(logger, "describe: Starting batch processing");
        if( clouds.size() != keyClouds.size() || clouds.size() != normalClouds.size() ) {

                throw std::invalid_argument("Clouds have different size");
        }
        std::vector<cv::Mat> vec;
        vec.reserve(clouds.size());
        auto keyCloud = keyClouds.begin();
        auto normalCloud = normalClouds.begin();
        for(const auto &cloud : clouds) {
```

```
                vec.push_back( describe(cloud, *keyCloud, *normalCloud) );
                ++keyCloud;
                ++normalCloud;
        }
        TRACE(logger, "describe: Finished batch processing");
        return vec;
}

} /* namespace Tagger3D */


/*
 * Tagger3D : Descriptor.h
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef DESCRIPTOR_H_
#define DESCRIPTOR_H_

#include "../Common/clouds.h"
#include "../Common/ProcessObject.h"

#include <opencv2/core/core.hpp>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>


namespace Tagger3D {



class Descriptor : public ProcessObject {
public:
        Descriptor(const std::map<std::string, std::string> &configMap);
        virtual ~Descriptor();

        virtual cv::Mat describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCl
oud, const NormalCloud::Ptr &normalCloud ) = 0;
        virtual std::vector<cv::Mat> describe(const ColorVec &clouds, const ScaleVec &keyCl
ouds, const NormalVec &normalClouds);

protected:

        const std::string moduleName = "descriptor" + separator;
private:
        Descriptor();
        const std::string loggerName = "Main.Descriptor";

};

} /* namespace Tagger3D */
#endif /* DESCRIPTOR_H_ */


/*
 * FPFHDescriptor.cpp
 *
 *   Created on: 30 sie 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#include "FPFHDescriptor.h"
#include "CloudParser/CloudParser.h"
```

```
#include <assert.h>

namespace Tagger3D {

FPFHDescriptor::FPFHDescriptor(const std::map<std::string, std::string> &configMap) : Descr
iptor(configMap) {

        radiusSearch = getParam<float>( radiusSearchKey );
        createFpfhDescriptor();
        assert(descriptor != nullptr);
}

FPFHDescriptor::~FPFHDescriptor() {}

void FPFHDescriptor::createFpfhDescriptor() {

        TRACE(logger, "createFpfhDescriptor: Starting");
        decltype(descriptor) newDescriptor( new pcl::FPFHEstimationOMP<pcl::PointXYZRGB, pc
l::Normal, pcl::FPFHSignature33>() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
        newDescriptor->setSearchMethod( kdTree );
        newDescriptor->setRadiusSearch( radiusSearch );

        descriptor = std::move( newDescriptor );
        TRACE(logger, "createFpfhDescriptor: Finished");
}

cv::Mat FPFHDescriptor::describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCl
oud, const NormalCloud::Ptr &normalCloud) {

        TRACE(logger, "describe fpfh: Starting");
        FpfhCloud::Ptr descriptors( new FpfhCloud() );
        ColorCloud::Ptr keyCloudRgb( new ColorCloud() );
        pcl::copyPointCloud( *keyCloud, *keyCloudRgb);
        descriptor->setInputNormals( normalCloud );
        descriptor->setSearchSurface( cloud );
        descriptor->setInputCloud( keyCloudRgb );
        descriptor->compute( *descriptors );
        TRACE(logger, "describe fpfh: Finished; descriptors size = " << descriptors->size()
);
        return CloudParser::parse<FpfhCloud>(descriptors);
}



} /* namespace Tagger3D */


/*
 * FPFHDescriptor.h
 *
 *   Created on: 30 sie 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#ifndef FPFHDESCRIPTOR_H_
#define FPFHDESCRIPTOR_H_

#include "Descriptor.h"

#include <pcl/features/fpfh_omp.h>

namespace Tagger3D {

/*
 *
```

```
 */
class FPFHDescriptor: public Descriptor {
        /**
         *
         */
public:
        FPFHDescriptor(const std::map<std::string, std::string> &configM);
        virtual ~FPFHDescriptor();

        cv::Mat describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCloud, con
st NormalCloud::Ptr &normalCloud );

private:
        FPFHDescriptor();
        void createFpfhDescriptor();

        std::unique_ptr<pcl::FPFHEstimationOMP<pcl::PointXYZRGB, pcl::Normal, pcl::FPFHSign
ature33>> descriptor;

        //      Config parameters
        float radiusSearch;

        //      Config keys
        const std::string radiusSearchKey = moduleName + "radiusSearch";
};

} /* namespace Tagger3D */
#endif /* FPFHDESCRIPTOR_H_ */


/*
 * Tagger3D : PFHDescriptor.cpp
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "PFHDescriptor.h"
#include "CloudParser/CloudParser.h"
#include <pcl/search/kdtree.h>
#include <assert.h>

namespace Tagger3D {

PFHDescriptor::PFHDescriptor(const std::map<std::string, std::string> &configMap) : Descrip
tor(configMap) {

        radiusSearch = getParam<float>( radiusSearchKey );
        createPfhDescriptor();
        assert(descriptor != nullptr);
}

PFHDescriptor::~PFHDescriptor() {}

void PFHDescriptor::createPfhDescriptor() {

        TRACE(logger, "createPfhDescriptor: Starting");
        decltype(descriptor) newDescriptor( new pcl::PFHEstimation<pcl::PointXYZRGB, pcl::N
ormal, pcl::PFHSignature125>() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
        newDescriptor->setSearchMethod( kdTree );
        newDescriptor->setRadiusSearch( radiusSearch );

        descriptor = std::move( newDescriptor );
        TRACE(logger, "createPfhDescriptor: Finished");
}
```

```
cv::Mat PFHDescriptor::describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyClo
ud, const NormalCloud::Ptr &normalCloud) {

        TRACE(logger, "describe: Starting");
        PfhCloud::Ptr descriptors( new PfhCloud() );
        ColorCloud::Ptr keyCloudRgb( new ColorCloud() );
        pcl::copyPointCloud( *keyCloud, *keyCloudRgb);
        descriptor->setInputNormals( normalCloud );
        descriptor->setSearchSurface( cloud );
        descriptor->setInputCloud( keyCloudRgb );
        descriptor->compute( *descriptors );
        TRACE(logger, "describe: Finished");
        return CloudParser::parse<PfhCloud>(descriptors);
}



} /* namespace Tagger3D */


/*
 * Tagger3D : PFHDescriptor.h
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef PFHDESCRIPTOR_H_
#define PFHDESCRIPTOR_H_

#include "Descriptor.h"

#include <pcl/features/pfh.h>


namespace Tagger3D {

class PFHDescriptor : public Descriptor {
public:
        PFHDescriptor(const std::map<std::string, std::string> &configMap);
        virtual ~PFHDescriptor();

        cv::Mat describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCloud, con
st NormalCloud::Ptr &normalCloud );

private:
        PFHDescriptor();
        void createPfhDescriptor();

        std::unique_ptr<pcl::PFHEstimation<pcl::PointXYZRGB, pcl::Normal, pcl::PFHSignature
125>> descriptor;

        //      Config parameters
        float radiusSearch;

        //      Config keys
        const std::string radiusSearchKey = moduleName + "radiusSearch";
};

} /* namespace Tagger3D */
#endif /* PFHDESCRIPTOR_H_ */


/*
 * PFHRGBDescriptor.cpp
 *
```

```
 *   Created on: 11 paÅº 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#include "PFHRGBDescriptor.h"
#include "CloudParser/CloudParser.h"

namespace Tagger3D {

PFHRGBDescriptor::PFHRGBDescriptor(const std::map<std::string, std::string> &configMap) : D
escriptor(configMap) {
        createDescriptor();
        assert(descriptor != nullptr);
}

void PFHRGBDescriptor::createDescriptor() {

        TRACE(logger, "createDescriptor: Starting");
        decltype(descriptor) newDescriptor( new descriptorType() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
        newDescriptor->setSearchMethod( kdTree );
        newDescriptor->setRadiusSearch(getParam<float>(radiusSearch));

        descriptor = std::move( newDescriptor );
        TRACE(logger, "createDescriptor: Finished");
}

cv::Mat PFHRGBDescriptor::describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &key
Cloud, const NormalCloud::Ptr &normalCloud) {

        TRACE(logger, "describe PFHRGB: Starting");
        PfhRgbCloud::Ptr descriptors( new PfhRgbCloud() );
        ColorCloud::Ptr keyCloudRgb( new ColorCloud() );
        pcl::copyPointCloud( *keyCloud, *keyCloudRgb);
        descriptor->setInputNormals( normalCloud );
        descriptor->setSearchSurface( cloud );
        descriptor->setInputCloud( keyCloudRgb );
        descriptor->compute( *descriptors );
        TRACE(logger, "describe PFHRGB: Finished; descriptors size = " << descriptors->size
());
        return CloudParser::parse<PfhRgbCloud>(descriptors);
}


} /* namespace Tagger3D */


/*
 * PFHRGBDescriptor.h
 *
 *   Created on: 11 paÅº 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#ifndef PFHRGBDESCRIPTOR_H_
#define PFHRGBDESCRIPTOR_H_

#include "Descriptor.h"

#include <pcl/features/pfhrgb.h>

namespace Tagger3D {

/*
 *
```

```
 */
class PFHRGBDescriptor: public Descriptor {
        /**
         *
         */
public:
        PFHRGBDescriptor() = delete;
        PFHRGBDescriptor(const std::map<std::string, std::string> &configMap);
        virtual ~PFHRGBDescriptor() = default;

        cv::Mat describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCloud, con
st NormalCloud::Ptr &normalCloud );

private:

        void createDescriptor();
        typedef pcl::PFHRGBEstimation<pcl::PointXYZRGB, pcl::Normal, pcl::PFHRGBSignature25
0> descriptorType;
        std::unique_ptr<descriptorType> descriptor;

        //      Config keys
        const std::string radiusSearch = moduleName + "radiusSearch";
};

} /* namespace Tagger3D */
#endif /* PFHRGBDESCRIPTOR_H_ */


/*
 * ShotColorDescriptor.cpp
 *
 *  Created on: 6 sty 2014
 *      Author: adam
 */

#include <ShotColorDescriptor.h>
#include "CloudParser.h"

namespace Tagger3D {

ShotColorDescriptor::ShotColorDescriptor(const std::map<std::string, std::string> &configMa
p) : Descriptor(configMap) {
        createDescriptor();
}

cv::Mat ShotColorDescriptor::describe(const ColorCloud::Ptr& cloud,
                const ScaleCloud::Ptr& keyCloud, const NormalCloud::Ptr& normalCloud) {

        TRACE(logger, "describe: Starting");
        ColorCloud::Ptr keyCloudRgb( new ColorCloud() );
        pcl::copyPointCloud( *keyCloud, *keyCloudRgb);

        pcl::PointCloud<PointType>::Ptr descriptors (new pcl::PointCloud<PointType>());
        descriptor->setInputCloud(keyCloudRgb);
        descriptor->setInputNormals(normalCloud);
        descriptor->setSearchSurface(cloud);
        descriptor->compute(*descriptors);

        TRACE(logger, "describe: Finished");
        return CloudParser::parse<pcl::PointCloud<PointType>>(descriptors);
}

void ShotColorDescriptor::createDescriptor() {

        TRACE(logger, "createDescriptor: Starting");
        decltype(descriptor) newDescriptor( new descriptorType() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
```

```
        newDescriptor->setSearchMethod( kdTree );
        newDescriptor->setRadiusSearch(getParam<float>(supportRadius));

        descriptor = std::move( newDescriptor );
        TRACE(logger, "createDescriptor: Finished");
}

} /* namespace Tagger3D */


/*
 * ShotColorDescriptor.h
 *
 *  Created on: 6 sty 2014
 *      Author: adam
 */

#ifndef SHOTCOLORDESCRIPTOR_H_
#define SHOTCOLORDESCRIPTOR_H_

#include <Descriptor.h>

#include <pcl/features/shot_omp.h>

namespace Tagger3D {

class ShotColorDescriptor: public Descriptor {
public:
        ShotColorDescriptor() = delete;
        ShotColorDescriptor(const std::map<std::string, std::string> &configMap);
        virtual ~ShotColorDescriptor() = default;

        cv::Mat describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCloud, con
st NormalCloud::Ptr &normalCloud );

private:

        void createDescriptor();
        typedef pcl::SHOT1344 PointType;
        typedef pcl::SHOTColorEstimationOMP<pcl::PointXYZRGB, pcl::Normal, PointType> descr
iptorType;
        std::unique_ptr<descriptorType> descriptor;

        //      Config keys
        const std::string supportRadius = moduleName + "supportRadius";
};

} /* namespace Tagger3D */

#endif /* SHOTCOLORDESCRIPTOR_H_ */


/*
 * ShotDescriptor.cpp
 *
 *  Created on: 6 sty 2014
 *      Author: adam
 */

#include <ShotDescriptor.h>
#include "CloudParser.h"

namespace Tagger3D {

ShotDescriptor::ShotDescriptor(const std::map<std::string, std::string> &configMap) : Descr
iptor(configMap) {

        createDescriptor();
```

```
        assert(descriptor != nullptr);
}

cv::Mat ShotDescriptor::describe(const ColorCloud::Ptr& cloud,
                const ScaleCloud::Ptr& keyCloud, const NormalCloud::Ptr& normalCloud) {

        TRACE(logger, "describe: Starting");
        ColorCloud::Ptr keyCloudRgb( new ColorCloud() );
        pcl::copyPointCloud( *keyCloud, *keyCloudRgb);

        pcl::PointCloud<PointType>::Ptr descriptors (new pcl::PointCloud<PointType>());
        descriptor->setInputCloud(keyCloudRgb);
        descriptor->setInputNormals(normalCloud);
        descriptor->setSearchSurface(cloud);
        descriptor->compute(*descriptors);

        TRACE(logger, "describe: Finished");
        return CloudParser::parse<pcl::PointCloud<PointType>>(descriptors);
}

void ShotDescriptor::createDescriptor() {

        TRACE(logger, "createDescriptor: Starting");
        decltype(descriptor) newDescriptor( new descriptorType() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
        newDescriptor->setSearchMethod( kdTree );
        newDescriptor->setRadiusSearch(getParam<float>(supportRadius));

        descriptor = std::move( newDescriptor );
        TRACE(logger, "createDescriptor: Finished");
}

} /* namespace Tagger3D */


/*
 * ShotDescriptor.h
 *
 *  Created on: 6 sty 2014
 *      Author: adam
 */

#ifndef SHOTDESCRIPTOR_H_
#define SHOTDESCRIPTOR_H_

#include <Descriptor.h>

#include <pcl/features/shot_omp.h>

namespace Tagger3D {

class ShotDescriptor: public Descriptor {
public:
        ShotDescriptor() = delete;
        ShotDescriptor(const std::map<std::string, std::string> &configMap);
        virtual ~ShotDescriptor() = default;

        cv::Mat describe(const ColorCloud::Ptr &cloud, const ScaleCloud::Ptr &keyCloud, con
st NormalCloud::Ptr &normalCloud );

private:

        void createDescriptor();
        typedef pcl::SHOT352 PointType;
        typedef pcl::SHOTEstimationOMP<pcl::PointXYZRGB, pcl::Normal, PointType> descriptor
Type;
        std::unique_ptr<descriptorType> descriptor;
```

```
        //      Config keys
        const std::string supportRadius = moduleName + "supportRadius";
};

} /* namespace Tagger3D */

#endif /* SHOTDESCRIPTOR_H_ */
```

/home/adam/workspace/Tagger3D/utils/src/Detector

```
/*
 * DenseDetector.cpp
 *
 *  Created on: 16 gru 2013
 *      Author: adam
 */

#include <DenseDetector.h>

namespace Tagger3D {

DenseDetector::DenseDetector(const std::map<std::string, std::string> &_configMap)
        : Detector(_configMap) {

        detector = std::unique_ptr<detectorType>(new detectorType());
        detector->setRadiusSearch(getParam<float>(radiusSearch));
}

ScaleCloud::Ptr DenseDetector::detect(const ColorCloud::Ptr &cloud) {
        TRACE(logger, "detect: Starting")

        pcl::PointCloud<int> sampled_indices;
        ScaleCloud::Ptr destCloud = ScaleCloud::Ptr(new ScaleCloud());

        detector->setInputCloud(cloud);
        detector->compute (sampled_indices);
        pcl::copyPointCloud (*cloud, sampled_indices.points, *destCloud);

        TRACE(logger, "detect: Finished")
        return destCloud;
}

} /* namespace Tagger3D */


/*
 * DenseDetector.h
 *
 *  Created on: 16 gru 2013
 *      Author: adam
 */

#ifndef DENSEDETECTOR_H_
#define DENSEDETECTOR_H_

#include <Detector.h>

#include <pcl/keypoints/uniform_sampling.h>

namespace Tagger3D {

class DenseDetector: public Detector {
public:
        DenseDetector() = delete;
        DenseDetector(const std::map<std::string, std::string> &_configMap);
```

```
        virtual ~DenseDetector() = default;

        ScaleCloud::Ptr detect(const ColorCloud::Ptr &cloud);

        typedef pcl::UniformSampling<pcl::PointXYZRGB> detectorType;
        typedef std::unique_ptr<detectorType> detectorPtrType;

private:

        detectorPtrType detector;
        const std::string radiusSearch = moduleName + "radiusSearch";
};

} /* namespace Tagger3D */

#endif /* DENSEDETECTOR_H_ */


/*
 * Tagger3D : Detector.cpp
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "Detector.h"

namespace Tagger3D {

Detector::Detector(const std::map<std::string, std::string> &configMap) : ProcessObject(con
figMap) {

        logger = lgr::Logger::getLogger( loggerName );
        DEBUG(logger, "Creating Detector");
}

Detector::~Detector() {

        DEBUG(logger, "Destroying Detector");
}

ScaleVec Detector::detect(const ColorVec &clouds) {
        TRACE(logger, "detect: Starting batch processing");

        int size = clouds.size();
        ScaleVec vec;
        vec.reserve(size);
        auto cloudPtr = &clouds[0];

        for(int i = 0; i < size; ++i) {
                vec.push_back(std::move(detect(cloudPtr[i])));
        }

        TRACE(logger, "detect: Finished batch processing");
        return vec;
}

} /* namespace Tagger3D */


/*
 * Tagger3D : Detector.h
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */
```

```
#ifndef DETECTOR_H_
#define DETECTOR_H_

#include "../Common/clouds.h"
#include "../Common/ProcessObject.h"

namespace Tagger3D {

class Detector : public ProcessObject {
public:
        Detector() = delete;
        Detector(const std::map<std::string, std::string> &configMap);
        virtual ~Detector();

        virtual ScaleCloud::Ptr detect(const ColorCloud::Ptr &cloud) = 0;
        virtual ScaleVec detect(const ColorVec &clouds);

protected:
        const std::string moduleName = "detector" + separator;

private:
        const std::string loggerName = "Main.Detector";
};

} /* namespace Tagger3D */
#endif /* DETECTOR_H_ */


/*
 * Iss3dDetector.cpp
 *
 *   Created on: 8 wrz 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#include "Iss3dDetector.h"

#include <pcl/search/kdtree.h>
#include <assert.h>

namespace Tagger3D {

Iss3dDetector::Iss3dDetector(const std::map<std::string, std::string> &configMap)
        : Detector(configMap) {

        createDetector();
        assert(detector != nullptr);
}

void Iss3dDetector::createDetector() {

        detectorPtrType temp(new pcl::ISSKeypoint3D<pcl::PointXYZRGB, pcl::PointXYZRGB>);
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree( new pcl::search::KdTree<pcl::Point
XYZRGB>() );
        float resolution = getParam<float>(modelResolution);
        salientRadius = 6 * resolution;
        nonMaxRadius = 4 * resolution;
        normalRadius = 4 * resolution;
        borderRadius = 1 * resolution;

        temp->setSearchMethod(tree);
        temp->setSalientRadius(salientRadius);
        temp->setNonMaxRadius(nonMaxRadius);
        temp->setBorderRadius(borderRadius);
        temp->setNormalRadius(normalRadius);
        temp->setThreshold21(getParam<float>(gamma21));
```

```
        temp->setThreshold32(getParam<float>(gamma32));
        temp->setMinNeighbors(getParam<int>(minNeighbours));
        temp->setNumberOfThreads(getParam<int>(threads));
        detector = std::move(temp);
}

ScaleCloud::Ptr Iss3dDetector::detect(const ColorCloud::Ptr &cloud) {

        TRACE(logger, "detect: Starting");
        ColorCloud::Ptr keyPoints(new ColorCloud() );
        //detector->setSearchSurface( cloud );
        detector->setInputCloud( cloud );

        detector->compute( *keyPoints);
        DEBUG(logger, "Detected " << keyPoints->size() << " keypoints");
        TRACE(logger, "detect: Finished");
        if(keyPoints->size() == 0) {

                std::runtime_error e("Could not find any keypoints");
                ERROR(logger, "detect: " << e.what() );
                throw e;
        }

        ScaleCloud::Ptr keyPointsScale(new ScaleCloud());
        pcl::copyPointCloud(*keyPoints, *keyPointsScale);
        return keyPointsScale;
}

} /* namespace Tagger3D */


/*
 * Iss3dDetector.h
 *
 *   Created on: 8 wrz 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#ifndef ISS3DDETECTOR_H_
#define ISS3DDETECTOR_H_

#include "Detector.h"

#include <pcl/keypoints/iss_3d.h>

namespace Tagger3D {

/*
 *
 */
class Iss3dDetector: public Detector {
        /**
         *
         */
public:
        Iss3dDetector() = delete;
        Iss3dDetector(const std::map<std::string, std::string> &configMap);
        virtual ~Iss3dDetector() = default;

        ScaleCloud::Ptr detect(const ColorCloud::Ptr &cloud);

        typedef pcl::ISSKeypoint3D<pcl::PointXYZRGB, pcl::PointXYZRGB> detectorType;
        typedef std::unique_ptr<detectorType> detectorPtrType;

private:
        void createDetector();
```

```
         detectorPtrType detector;

        const std::string
                gamma21 = moduleName + "gamma21",
                gamma32 = moduleName + "gamma32",
                minNeighbours = moduleName + "minNeighbours",
                threads = moduleName + "threads",
                modelResolution = moduleName + "modelResolution";

        float
                salientRadius,
                nonMaxRadius,
                normalRadius,
                borderRadius;
};

} /* namespace Tagger3D */
#endif /* ISS3DDETECTOR_H_ */


/*
 * Tagger3D : SIFTDetector.cpp
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "SIFTDetector.h"

#include <pcl/search/kdtree.h>
#include <assert.h>

namespace Tagger3D {

SIFTDetector::SIFTDetector(const std::map<std::string, std::string> &configMap) : Detector(
configMap) {

        minScale = getParam<float>(minScaleKey);
        octaves = getParam<int>(octavesKey);
        scalesPerOctave = getParam<int>(scalesPerOctaveKey);
        minContrast = getParam<float>(minContrastKey);
        createSiftDetector();
        assert( siftDetector != nullptr );
}

SIFTDetector::~SIFTDetector() {}

bool SIFTDetector::createSiftDetector() {

        TRACE(logger, "createSiftDetector: Starting");
        decltype(siftDetector) detector( new pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::Point
WithScale>() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
        detector->setSearchMethod(kdTree);
        detector->setScales(minScale, octaves, scalesPerOctave);
        detector->setMinimumContrast(minContrast);

        siftDetector = std::move( detector );

        TRACE(logger, "createSiftDetector: Finished");
        return true;
}

ScaleCloud::Ptr SIFTDetector::detect(const ColorCloud::Ptr &cloud) {

        TRACE(logger, "detect: Starting");
```

```
        ScaleCloud::Ptr keyPoints(new ScaleCloud() );
        siftDetector->setSearchSurface( cloud );
        siftDetector->setInputCloud( cloud );

        siftDetector->compute( *keyPoints);
        DEBUG(logger, "Detected " << keyPoints->size() << " keypoints");
        TRACE(logger, "detect: Finished");
        if(keyPoints->size() == 0) {

                std::runtime_error e("Could not find any keypoints");
                ERROR(logger, "detect: " << e.what() );
                throw e;
        }
        return keyPoints;
}

} /* namespace Tagger3D */


/*
 * Tagger3D : SIFTDetector.h
 *
 *  Created on:         24 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef SIFTDETECTOR_H_
#define SIFTDETECTOR_H_

#include "Detector.h"
#include <pcl/keypoints/sift_keypoint.h>

namespace Tagger3D {

class SIFTDetector : public Detector {
public:
        SIFTDetector(const std::map<std::string, std::string> &configMap);
        virtual ~SIFTDetector();

        ScaleCloud::Ptr detect(const ColorCloud::Ptr &cloud);

private:
        SIFTDetector();
        bool createSiftDetector();

        std::unique_ptr<pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::PointWithScale>> siftDetec
tor;

        float minScale;
        int octaves;
        int scalesPerOctave;
        float minContrast;

        const std::string minScaleKey = moduleName + "minScale";
        const std::string octavesKey = moduleName + "octaves";
        const std::string scalesPerOctaveKey = moduleName + "scalesPerOctave";
        const std::string minContrastKey = moduleName + "minContrast";


};

} /* namespace Tagger3D */
#endif /* SIFTDETECTOR_H_ */


/home/adam/workspace/Tagger3D/utils/src/ImgReader
```

```
%PDF-1.4
%Çì\217¢
5 0 obj
<</Length 6 0 R/Filter /FlateDecode>>
stream
x\234\235UkOÛ0\024ý\236_qU¤Ñ2h^<\003CêZÄ\030\024±*\2326-\0232\216\223Z$qp\234NÛÄ˜ûì<Ú¤M§j
\216\224\207Ï¹ç>|\035¿\200Ñ7ÁPWùÄ\221¦ON H5\003\002íE3s\020Ê\007\216à½+ §jÂõµÂÆ\004Ë\206
\223³Ã¾mÛàFÚ•®>c\021Ñ\221\207"ý\007ãÏi\2020Ñ]\024\004\204Û#=åX¿\211\202 A\036áú\004Å\001Y|ö
g½\203Ã#Ãèº³
```

F\004\203y\006æ©c\036;GÒ\215aÚ½ïîG\031ÁY\025Á±òj\201{§uõ½Þ®f\232ù;ìAå\017\034Xó±«\030Å\035‘
Ĕ\202\220\022\005\002\226ÈÄR214ñ%¥>\0062U¸e)e\234<W\006j\214H\2129M\004\225âÅ¼.\037V\021ð\016õc
\217ø0\031Ü__Ý\214¯’\203ÑÕäñÃã2\215\035 Ó\230´1\026\0321\0163\217@§\226_G¹Z \027,!1\236[:
\226Ñå•þ,I.\025’F\021ÉWgY-ßj\036\207(MWÊæ@\222=\205\024Ãb&ç\226\221\026X\221c9\232æÓ.fq*
\025\236ãD(¹È_RÁi\034ìCíã\022ÞH¦O\2031J¦½óºâ\234r\221¡\020þ¬J\027¼²$\025{ÈBÆ\207!Ë¤ô\203à
\227Ù\2236\025{Á£±È±;ôDÂ5´ò9gÔ\203\224\2101óÈ´«l"õV\220\023Î\004Á²\211\234Z\013V\002Êpá»V
\204"]\225-\214ó\001\211Ù>´ \036IÄ¬Dç\2163FBZ(\237ÊJJÖ¦sªòÒÌ‘c\035þ?\226iO9¨Õ<át.7QKúªR\230
e±h©yJ\177\221\226\205[÷\PoÉOx§Ê\236\205ä^v.¼\205N\201tÎ×\035¯«216h<,ë¶¦Ô@ïh*:\215\2107È
\215Ê\232•ËUèÖry\017¶kåPº\225\022IÅæ4kà¶b\233\223¬\201Û\212mH±B\226\031®tE®3\227»\214ñ\213Æ
ï¢jûÏ\004•´ÁF3\\026¡4Sg×?˜\036\021\022.Ë¿«}³Ü\215Ë\235Ø\004\232û±µÉý\220I¦\217Tx_

```
Y\025°-£oøòü°ûG\206I\016l\177co×\215¿*cë°Í¶Ììµ\026Ä+è{ÐòóÏO§êì!±G}E\=}
Ú\225«}\222×_¸Ñg?endstream
endobj
6 0 obj
742
endobj
4 0 obj
<</Type/Page/MediaBox [0 0 595 842]
/Rotate 0/Parent 3 0 R
/Resources<</ProcSet[/PDF /Text]
/ExtGState 10 0 R
/Font 11 0 R
>>
/Contents 5 0 R
>>
endobj
3 0 obj
<< /Type /Pages /Kids [
4 0 R
] /Count 1
>>
endobj
1 0 obj
<</Type /Catalog /Pages 3 0 R
/Metadata 13 0 R
>>
endobj
7 0 obj
<</Type/ExtGState
/OPM 1>>endobj
10 0 obj
<</R7
7 0 R>>
endobj
11 0 obj
<</R9
9 0 R/R8
8 0 R>>
endobj
9 0 obj
<</BaseFont/Courier/Type/Font
/Encoding 12 0 R/Subtype/Type1>>
endobj
12 0 obj
<</Type/Encoding/Differences[
126/tilde]>>
endobj
8 0 obj
<</BaseFont/Courier-Bold/Type/Font
/Subtype/Type1>>
endobj
13 0 obj
<</Type/Metadata
/Subtype/XML/Length 1354>>stream
<?xpacket begin='ï»¿' id='W5M0MpCehiHzreSzNTczkc9d'?>
<?adobe-xap-filters esc="CRLF"?>
<x:xmpmeta xmlns:x='adobe:ns:meta/' x:xmptk='XMP toolkit 2.9.1-13, framework 1.6'>
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#' xmlns:iX='http://ns.adobe.
com/iX/1.0/'>
<rdf:Description rdf:about='uuid:04cfb447-c41c-11ee-0000-ee1f08c3c913' xmlns:pdf='http://ns
.adobe.com/pdf/1.3/' pdf:Producer='GPL Ghostscript 9.07'/>
<rdf:Description rdf:about='uuid:04cfb447-c41c-11ee-0000-ee1f08c3c913' xmlns:xmp='http://ns
.adobe.com/xap/1.0/'><xmp:ModifyDate>2014-02-02T12:45:10+01:00</xmp:ModifyDate>
<xmp:CreateDate>2014-02-02T12:45:10+01:00</xmp:CreateDate>
<xmp:CreatorTool>GNU Enscript 1.6.5.90</xmp:CreatorTool></rdf:Description>
<rdf:Description rdf:about='uuid:04cfb447-c41c-11ee-0000-ee1f08c3c913' xmlns:xapMM='http://
ns.adobe.com/xap/1.0/mm/' xapMM:DocumentID='uuid:04cfb447-c41c-11ee-0000-ee1f08c3c913'/>
<rdf:Description rdf:about='uuid:04cfb447-c41c-11ee-0000-ee1f08c3c913' xmlns:dc='http://pur
```

```
l.org/dc/elements/1.1/' dc:format='application/pdf'><dc:title><rdf:Alt><rdf:li xml:lang='x-
default'>Enscript Output</rdf:li></rdf:Alt></dc:title></rdf:Description>
</rdf:RDF>
</x:xmpmeta>


<?xpacket end='w'?>
endstream
endobj
2 0 obj
<</Producer(GPL Ghostscript 9.07)
/CreationDate(D:20140202124510+01'00')
/ModDate(D:20140202124510+01'00')
/Title(Enscript Output)
/Creator(GNU Enscript 1.6.5.90)>>endobj
xref
0 14
0000000000 65535 f
0000001065 00000 n
0000002875 00000 n
0000001006 00000 n
0000000846 00000 n
0000000015 00000 n
0000000827 00000 n
0000001130 00000 n
0000001377 00000 n
0000001240 00000 n
0000001171 00000 n
0000001201 00000 n
0000001318 00000 n
0000001444 00000 n
trailer
<< /Size 14 /Root 1 0 R /Info 2 0 R
/ID [<0B4D48F33EEB4A584E657BDB080BF4BD><0B4D48F33EEB4A584E657BDB080BF4BD>]
>>
startxref
3054
%%EOF


/*
 * Tagger3D : ImgReader.cpp
 *
 *  Created on:         22 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "ImgReader.h"
#include <opencv2/core/types_c.h>
#include <fstream>
namespace Tagger3D {

ImgReader::ImgReader(const std::map<std::string, std::string> &configMap) : ProcessObject(c
onfigMap) {

        logger = lgr::Logger::getLogger( loggerName );
        DEBUG(logger, "Creating ImgReader");
}

ImgReader::~ImgReader() {

        DEBUG(logger, "Destroying ImgReader");
}

void ImgReader::init() {

        currentMode = -1;
```

```
        std::string stringMode = getParam<std::string>(mode);
        int tmpMode;
        bool att = false;
        if(stringMode == "train") {tmpMode = TRAIN; att = true;}
        else if(stringMode == "test") {tmpMode = TEST; att = true;}
        if(att)
                setMode(tmpMode);
        else
                setMode(getParam<int>( mode ));
}

std::vector<std::string> ImgReader::getLineList(const std::string &path) {

        TRACE(logger, "getImgList: Starting");
        std::ifstream listFile( path );
        if( !listFile.good()) {

                std::runtime_error e("Cannot open the following file " + path);
                ERROR(logger, e.what());
                throw e;
        }

        std::string line;
        std::vector<std::string> imgList;
        while( !listFile.eof() ) {

                std::getline(listFile, line);
                if( !line.empty() ) {
                        imgList.push_back( line );
                }

        }
        TRACE(logger, "getImgList: Finished");
        return imgList;
}

std::string ImgReader::typeToStr(const int &type) {

  std:: string r;

  uchar depth = type & CV_MAT_DEPTH_MASK;
  uchar chans = 1 + (type >> CV_CN_SHIFT);

  switch ( depth ) {
    case CV_8U:  r = "8U"; break;
    case CV_8S:  r = "8S"; break;
    case CV_16U: r = "16U"; break;
    case CV_16S: r = "16S"; break;
    case CV_32S: r = "32S"; break;
    case CV_32F: r = "32F"; break;
    case CV_64F: r = "64F"; break;
    default:     r = "User"; break;
  }

  r += "C";
  r += (chans+'0');

  return r;
}

std::vector<int> ImgReader::readLabels() {
        INFO(logger, "Reading labels");

        std::vector<int> labels;
        for(const auto &str : labelVec) {

                labels.push_back(atoi(str.c_str()));
        }
```

```
        DEBUG(logger, "Read " << labels.size() << " labels");
        return labels;
}



} /* namespace Tagger3D */


/*
 * Tagger3D : ImgReader.h
 *
 *  Created on:         22 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */
#ifndef IMGREADER_H_
#define IMGREADER_H_

#include "../Common/clouds.h"
#include "../Common/ProcessObject.h"

#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
namespace Tagger3D {

class ImgReader : public ProcessObject{
public:
        ImgReader(const std::map<std::string, std::string> &configMap);
        virtual ~ImgReader();

        /**
         *      Reads a single image.
         *      @return a range image.
         */
        virtual ColorCloud::Ptr readImg() = 0;

        /**
         * Returns a label for a previously read image
         */
        virtual int readLabel() = 0;

        /**
         * Reads labels for a batch of images
         * @return a vector of labels
         */
        virtual std::vector<int> readLabels();

        virtual void setMode(int mode) = 0;

        enum { TRAIN, TEST };
protected:
        std::string typeToStr(const int &type);
        std::vector<std::string> getLineList(const std::string &path);
        std::string moduleName = "ImgReader" + separator;
        std::vector<std::string> labelVec;

        int currentMode;

        void init();


private:
        const std::string loggerName = "Main.ImgReader";
};

} /* namespace Tagger3D */
#endif /* IMGREADER_H_ */
```

```
/*
 * PcdReader.cpp
 *
 *   Created on: 29 sie 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#include "PcdReader.h"

#include <pcl/io/pcd_io.h>
#include <pcl/visualization/cloud_viewer.h>

#include <assert.h>

namespace Tagger3D {

PcdReader::PcdReader(const std::map<std::string, std::string> &configMap) : ImgReader(confi
gMap) {

        init();
        voxelGrid = std::unique_ptr<pcl::VoxelGrid<pcl::PointXYZRGB>>(new pcl::VoxelGrid<pc
l::PointXYZRGB>());
        assert(voxelGrid != nullptr);
        leaf = getParam<float>( leafSize );
        voxelGrid->setLeafSize(leaf, leaf, leaf);

        count = -1;
}

PcdReader::~PcdReader() {}

ColorCloud::Ptr PcdReader::readImg(const std::string& pcdPath) {

        ColorCloud::Ptr cloud(new ColorCloud());
        pcl::io::loadPCDFile(pcdPath, *cloud);

//      for(const auto& point : cloud->points)
//              std::cout << "x: " << point.x << " y: " << point.y << " z: " << point.z <<
std::endl;

//      pcl::visualization::CloudViewer viewer ("Simple Cloud Viewer");
//      viewer.showCloud (cloud);
//      while (!viewer.wasStopped ())
//      {
//      }

        if(leaf != 0) {

                voxelGrid->setInputCloud(cloud);
                voxelGrid->filter(*cloud);
        }

//      for(const auto& point : cloud->points)
//              std::cout << "x: " << point.x << " y: " << point.y << " z: " << point.z <<
std::endl;




        //std::terminate();
        DEBUG(logger, "Cloud size = " << cloud->size());
        return cloud;
}

ColorCloud::Ptr PcdReader::readImg() {
```

```
        count++;
        ColorCloud::Ptr ptr;
        if(count < pcdVec.size())
                ptr = readImg(pcdVec[count]);
        else
                ptr = ColorCloud::Ptr(new ColorCloud());

        if(ptr->empty() && count < pcdVec.size()) {

                std::runtime_error e("Corrupted point cloud #" + count);
                ERROR(logger, "readImg: " << e.what());
                throw e;
        }

        return ptr;
}

int PcdReader::readLabel() {

        if(count < labelVec.size())
                return atoi(labelVec[count-1].c_str());
        return -1;
}

void PcdReader::setMode(int mode) {

        if(currentMode != mode) {
                currentMode = mode;
                count = -1;

                switch(currentMode) {
                        case TRAIN:
                                pcdVec = getLineList( getParam<std::string>(trainPcd));
                                labelVec = getLineList( getParam<std::string>(trainPcdLabel
s));
                                break;
                        case TEST:
                                pcdVec = getLineList( getParam<std::string>(testPcd) );
                                labelVec = getLineList( getParam<std::string>(testPcdLabels
));
                                break;
                        default:
                                std::runtime_error e("Invalid mode");
                                ERROR(logger, "ImgReader: " << e.what());
                                throw e;
                        }
        }

}

} /* namespace Tagger3D */


/*
 * PcdReader.h
 *
 *   Created on: 29 sie 2013
 *       Author: Adam Kosiorek
 *      Description:
 */

#ifndef PCDREADER_H_
#define PCDREADER_H_

#include "ImgReader.h"
#include <pcl/filters/voxel_grid.h>
namespace Tagger3D {
```

```
/*
 *
 */
class PcdReader: public ImgReader {
        /**
         *
         */
public:
        PcdReader(const std::map<std::string, std::string> &configMap);
        virtual ~PcdReader();

        ColorCloud::Ptr readImg();
        ColorCloud::Ptr readImg(const std::string &pcdPath);

        int readLabel();

        virtual void setMode(int mode);

private:
        PcdReader();
        std::unique_ptr<pcl::VoxelGrid<pcl::PointXYZRGB>> voxelGrid;
        float leaf;
        int count;

        const std::string leafSize = moduleName + "leafSize";
        const std::string trainPcd = moduleName + "trainPcdList";
        const std::string testPcd = moduleName + "testPcdList";
        const std::string trainPcdLabels = moduleName + "trainPcdLabels";
        const std::string testPcdLabels = moduleName + "testPcdLabels";

        std::vector<std::string> pcdVec;

};

} /* namespace Tagger3D */
#endif /* PCDREADER_H_ */


/*
 * Tagger3D : RangeImgReader.cpp
 *
 *  Created on:         22 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "RangeImgReader.h"

#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <pcl/filters/filter.h>
#include <pcl/visualization/cloud_viewer.h>
#include <fstream>

namespace Tagger3D {

RangeImgReader::RangeImgReader(const std::map<std::string, std::string> &configMap) : ImgRe
ader(configMap) {

        init();
        count = -1;
        resize = getParam<int>( resizeKey );
}

RangeImgReader::~RangeImgReader() {}

ColorCloud::Ptr RangeImgReader::matToCloud(const cv::Mat &colorImg, const cv::Mat &depthImg
```

```
) {

        TRACE(logger, "MatToCloud: Starting");
        int height = depthImg.rows;
        int width = depthImg.cols;

        int height2 = height/2;
        int width2 = width/2;
        float factorX = factorX0 / width;
        float factorY = factorY0 / height;

        if( height != colorImg.rows || width != colorImg.cols ) {

                std::runtime_error e("Images have different dimensions");
                ERROR(logger, "MatToCloud: " << e.what() );
                throw e;
        }
        ColorCloud::Ptr cloud( new ColorCloud() );
        cloud->height = height;
        cloud->width = width;
        cloud->reserve(height * width);
        cloud->is_dense = false;

        pcl::PointXYZRGB newPoint;
        for (int y = 0; y < height; y++) {

                // For the sake of simplicity
                typedef ushort MatType;
                const MatType *depthPtr = depthImg.ptr<MatType>(y);
                const cv::Vec3b *colorPtr = colorImg.ptr<cv::Vec3b>(y);

                for (int x = 0; x < width; x++) {
                        float depth = depthPtr[x] / 1000.0f;

                        if ( depth == depth) {            // if depthValue is not NaN

                                newPoint.z = depth;
                                newPoint.x = (x - width2) * factorX * depth;
                                newPoint.y = (y - height2) * factorY * depth;
//                              std::cout << "x0: " << x << " y0: " << y << " z: " << newPo
int.z << " x: " << newPoint.x << " y: " << newPoint.y << std::endl;
                                cv::Vec3b vec = colorPtr[x];
                                newPoint.r = vec[2];
                                newPoint.g = vec[1];
                                newPoint.b = vec[0];

                        } else {

                                newPoint.z = std::numeric_limits<MatType>::quiet_NaN();
                                newPoint.x = std::numeric_limits<MatType>::quiet_NaN();
                                newPoint.y = std::numeric_limits<MatType>::quiet_NaN();
                                newPoint.r = std::numeric_limits<unsigned char>::quiet_NaN(
);
                                newPoint.g = std::numeric_limits<unsigned char>::quiet_NaN(
);
                                newPoint.b = std::numeric_limits<unsigned char>::quiet_NaN(
);
                        }
                        cloud->push_back(newPoint);
                }
        }
        TRACE(logger, "MatToCloud: Finished");
        std::vector<int> vec;
        pcl::removeNaNFromPointCloud( *cloud, *cloud, vec );


//      pcl::visualization::CloudViewer viewer ("Simple Cloud Viewer");
//      viewer.showCloud (cloud);
```

```
//      while (!viewer.wasStopped ())
//      {
//      }

        return cloud;
}

void RangeImgReader::readImg(const std::string &colorPath, const std::string &depthPath, cv
::Mat &colorImg, cv::Mat &depthImg) {

        TRACE(logger, "readImg: Starting");
        DEBUG(logger, "ColorImg = " << colorPath << " DepthImg = " << depthPath);
        depthImg = cv::imread(depthPath, CV_LOAD_IMAGE_ANYCOLOR | CV_LOAD_IMAGE_ANYDEPTH);
        if( !depthImg.data ) {

                std::runtime_error e("Cloud not read image " + depthPath);
                ERROR(logger, "readImg: " << e.what() );
                throw e;
        }

        colorImg = cv::imread(colorPath);
        if( !colorImg.data ) {

                std::runtime_error e("Cloud not read image " + colorPath);
                ERROR(logger, "readImg: " << e.what() );
                throw e;
        }

        if( resize > 0) {

                int width = colorImg.cols;
                int height = colorImg.rows;
                float ratio = float(width)/height;

                if( height >= width && height > resize ) {

                        height = resize;
                        width = height * ratio;
                } else if ( width > resize ) {

                        width = resize;
                        height = width / ratio;
                }
                TRACE(logger, "width = " << width << " height = " << height);
                cv::resize(colorImg, colorImg, cv::Size(width, height), 0, 0, CV_INTER_CUBI
C);
                cv::resize(depthImg, depthImg, cv::Size(width, height), 0, 0, CV_INTER_CUBI

C);
        }
        TRACE(logger, "readImg: Finished");
}

ColorCloud::Ptr RangeImgReader::readImg(const std::string &colorPath, const std::string &de
pthPath) {

        cv::Mat colorImg, depthImg;
        readImg(colorPath, depthPath, colorImg, depthImg);
        return matToCloud(colorImg, depthImg);

}

ColorCloud::Ptr RangeImgReader::readImg() {

        DEBUG(logger, "Img #" << count);
        DEBUG(logger, "depthImgVec size = " << colorImgVec.size());
        ColorCloud::Ptr cloud;

        count++;
```

```
        if(count < colorImgVec.size())
                cloud = readImg(colorImgVec.at(count), depthImgVec.at(count));
        else
                cloud = ColorCloud::Ptr(new ColorCloud());

        return cloud;
}

int RangeImgReader::readLabel() {

        DEBUG(logger, "readLabel");
        if(count  < colorImgVec.size())
                return atoi(labelVec[count].c_str());
        return -1;
}

void RangeImgReader::setMode(int mode) {

        if(currentMode != mode) {
                currentMode = mode;
                count = -1;

                switch(currentMode) {
                        case TRAIN:
                                colorImgVec = getLineList( getParam<std::string>(trainColor
Img) );
                                depthImgVec = getLineList( getParam<std::string>(trainDepth
Img) );
                                labelVec = getLineList( getParam<std::string>(trainLabel));
                                break;
                        case TEST:
                                colorImgVec = getLineList( getParam<std::string>(testColorI
mg) );
                                depthImgVec = getLineList( getParam<std::string>(testDepthI
mg) );
                                labelVec = getLineList( getParam<std::string>(testLabel));
                                break;
                        default:
                                std::runtime_error e("Invalid mode");
                                ERROR(logger, "ImgReader: " << e.what());
                                throw e;
                        }
        }

}

} /* namespace Tagger3D */


/*
 * Tagger3D : RangeImgReader.h
 *
 *  Created on:         22 lip 2013
 *  Author:                 Adam Kosiorek
 *      Description:
 */

#ifndef RANGEIMGRADER_H_
#define RANGEIMGRADER_H_

#include "ImgReader.h"

#include <opencv2/core/core.hpp>

namespace Tagger3D {

class RangeImgReader: public ImgReader {
public:
```

```
        RangeImgReader(const std::map<std::string, std::string> &configMap);
        virtual ~RangeImgReader();

        ColorCloud::Ptr readImg();

        int readLabel();

        virtual void setMode(int mode);

protected:
        void readImg(const std::string &colorPath, const std::string &depthPath, cv::Mat &c
olorImg, cv::Mat &depthImg);
        ColorCloud::Ptr readImg(const std::string &colorPath, const std::string &depthPath)
;

private:
        int count;
        int resize;

        const std::string resizeKey = moduleName + "resize";
        const std::string trainColorImg = moduleName + "trainColorImgList";
        const std::string trainDepthImg = moduleName + "trainDepthImgList";
        const std::string trainLabel = moduleName + "trainLabelsList";
        const std::string testColorImg = moduleName + "testColorImgList";
        const std::string testDepthImg = moduleName + "testDepthImgList";
        const std::string testLabel = moduleName + "testLabelsList";

        std::vector<std::string> depthImgVec;
        std::vector<std::string> colorImgVec;


        ColorCloud::Ptr matToCloud(const cv::Mat &colorImg, const cv::Mat &depthImg);

        const float factorX0 =  320.0f * 3.501e-3f;
        const float factorY0 = 240.0f * 3.501e-3f;


};

} /* namespace Tagger3D */
#endif /* RANGEIMGRADER_H_ */


/*
 * Tagger3D : main.cpp
 *
 *  Created on:         22 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "Common/clouds.h"
#include "Common/logger.h"
#include "Config/Config.h"
#include "Common/Tagger3D.h"

#include "log4cxx/consoleappender.h"
#include "log4cxx/propertyconfigurator.h"
#include "log4cxx/patternlayout.h"
#include <boost/algorithm/string.hpp>

#include <map>
#include <string>
#include <memory>

namespace t3d = Tagger3D;

void configureLogger(int &argc, char** argv, const lgr::LoggerPtr &logger);
```

```
int main(int argc, char** argv) {

        static lgr::LoggerPtr logger(lgr::Logger::getLogger("Main"));
        configureLogger(argc, argv, logger);

        TRACE(logger, "Entering app");

        t3d::Config config(argc, const_cast<const char**>(argv));
        const std::map<std::string, std::string> configMap = config.getConfigMap();
        if( configMap.empty() )
                exit(2);

        t3d::Tagger3D tagger(configMap);
        tagger.run();
}

void configureLogger(int &argc, char** argv, const lgr::LoggerPtr &logger) {

        bool foundConfig = false;
        if(argc > 1) {

                std::string filePath;
                std::vector<std::string> splitted;
                for(int i = 1; i < argc; i++) {
                        filePath = argv[i];
                        boost::split(splitted, filePath, boost::is_any_of("."));
                        if(splitted.back() == "ini") {
                                foundConfig = true;
                                lgr::PropertyConfigurator::configure(filePath);
                                break;
                        }
                }
        }

        if(!foundConfig) {

                std::string pattern = " %d{HH:mm:ss:SSS} (%c{1}:%L) - %m%n";
                std::string target = "System.out";
                lgr::LayoutPtr layout(new lgr::PatternLayout(pattern));
                lgr::AppenderPtr consoleAppender(new lgr::ConsoleAppender(layout, target));
                consoleAppender->setName("Console");
                logger->addAppender(consoleAppender);
        }
}


/home/adam/workspace/Tagger3D/utils/src/PointNormal


/*
 * Tagger3D : NormalEstimator.cpp
 *
 *  Created on:         28 lip 2013
 *  Author:                 Adam Kosiorek
 *      Description:
 */

#include "NormalEstimator.h"

#include <pcl/search/kdtree.h>
#include <pcl/filters/filter.h>
#include <assert.h>

namespace Tagger3D {

NormalEstimator::NormalEstimator(const std::map<std::string, std::string> &configMap) : Poi
ntNormal(configMap) {
```

```
        createNormalEstimator();
        assert( normalEstimator != nullptr );
}

void NormalEstimator::createNormalEstimator() {

        TRACE(logger, "createNormalEstimator: Starting");
        std::unique_ptr<pcl::NormalEstimationOMP<pcl::PointXYZRGB, pcl::Normal>> estimator(
new pcl::NormalEstimationOMP<pcl::PointXYZRGB, pcl::Normal>() );
        pcl::search::KdTree<pcl::PointXYZRGB>::Ptr kdTree( new pcl::search::KdTree<pcl::Poi
ntXYZRGB>() );
        estimator->setSearchMethod( kdTree );

        float radius = getParam<float>( normalRadius );
        if(radius > 0)
                estimator->setRadiusSearch( radius );
        else {
                int k = getParam<int>( kNN );
                assert(k > 0);
                estimator->setKSearch(k);
        }

        normalEstimator = std::move( estimator );
        TRACE(logger, "createNormalEstimator: Finished");
}

NormalCloud::Ptr NormalEstimator::computeNormals(const ColorCloud::Ptr &cloud) {

        TRACE(logger, "computeNormals: Starting");

        NormalCloud::Ptr normalCloud( new NormalCloud() );
        normalEstimator->setInputCloud( cloud );
        normalEstimator->compute( *normalCloud );
        pcl::removeNaNNormalsFromPointCloud( *normalCloud, *normalCloud, index);

        TRACE(logger, "computeNormals: Finished");
        return normalCloud;
}

NormalVec NormalEstimator::computeNormals(const ColorVec &clouds) {

        TRACE(logger, "computeNormals: Starting batch processing");
        NormalVec vec;
        for(auto &cloud : clouds) {

                vec.emplace_back( computeNormals( cloud ) );
        }

        TRACE(logger, "computeNormals: Finished batch processing");
        return vec;
}

} /* namespace Tagger3D */


/*
 * Tagger3D : NormalEstimator.h
 *
 *  Created on:         28 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef NORMALESTIMATOR_H_
#define NORMALESTIMATOR_H_

#include "PointNormal.h"
```

```
#include <pcl/features/normal_3d_omp.h>

namespace Tagger3D {

class NormalEstimator: public PointNormal {
public:
        NormalEstimator() = delete;
        NormalEstimator(const std::map<std::string, std::string> &configMap);
        virtual ~NormalEstimator() = default;

        NormalCloud::Ptr computeNormals(const ColorCloud::Ptr &cloud);
        NormalVec computeNormals(const ColorVec &clouds);


private:
        void createNormalEstimator();

        std::unique_ptr<pcl::NormalEstimationOMP<pcl::PointXYZRGB, pcl::Normal>> normalEsti
mator;

        //      Config keys
        const std::string normalRadius = moduleName + "normalRadius";
        const std::string kNN = moduleName + "kNN";
};

} /* namespace Tagger3D */
#endif /* NORMALESTIMATOR_H_ */


/*
 * Tagger3D : PointNormal.cpp
 *
 *  Created on:         28 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#include "PointNormal.h"

namespace Tagger3D {

PointNormal::PointNormal(const std::map<std::string, std::string> &configMap) : ProcessObje
ct(configMap){

        logger = lgr::Logger::getLogger( loggerName );
        DEBUG(logger, "Creating PointNormal");
}

PointNormal::~PointNormal() {

        DEBUG(logger, "Destroying PointNormal");
}

void PointNormal::cleanupInputCloud( ColorCloud::Ptr &cloud) {

        TRACE(logger, "cleanupInputCloud: Starting");
        size_t j = 0;
        size_t indexSize = index.size();
        size_t cloudSize = cloud->points.size();
        if( cloudSize == indexSize ) {

                return;
        }

        for(size_t i = 0; i < indexSize; ++i) {

                cloud->points[i] = cloud->points[index[i]];
```

```
        }

        cloud->resize(indexSize);
        cloud->height = 1;
        cloud->width = indexSize;
        TRACE(logger, "cleanupInputCloud: Finished");
}

} /* namespace Tagger3D */


/*
 * Tagger3D : PointNormal.h
 *
 *  Created on:         28 lip 2013
 *  Author:                     Adam Kosiorek
 *      Description:
 */

#ifndef POINTNORMAL_H_
#define POINTNORMAL_H_

#include "../Common/clouds.h"
#include "../Common/ProcessObject.h"

#include <pcl/point_types.h>
#include <pcl/point_cloud.h>

namespace Tagger3D {

class PointNormal: public Tagger3D::ProcessObject {
public:
        PointNormal(const std::map<std::string, std::string> &configMap);
        virtual ~PointNormal();

        virtual NormalCloud::Ptr computeNormals(const ColorCloud::Ptr &cloud) = 0;
        virtual NormalVec computeNormals(const ColorVec &clouds) = 0;
        void cleanupInputCloud(ColorCloud::Ptr &cloud);

protected:
        const std::string moduleName = "PointNormal" + separator;
        std::vector<int> index;

private:
        PointNormal();
        const std::string loggerName = "Main.PointNormal";
};

} /* namespace Tagger3D */
#endif /* POINTNORMAL_H_ */


/home/adam/workspace/Tagger3D/utils/src/Predictor


/*
 * CvSVMPredictor.cpp
 *
 *  Created on: Aug 22, 2013
 *      Author: Adam Kosiorek
 * Description: svm predictor implementation
 */

#include "CvSVMPredictor.h"

#include <stdio.h>
#include <assert.h>
#include <algorithm>
```

```
#include <iostream>
#include <vector>
#include <cmath>
#include <iostream>
#include <fstream>
#include <iterator>

namespace Tagger3D {

CvSVMPredictor::CvSVMPredictor(const std::map<std::string, std::string> &_configMap,
                const std::string &predictorType)

        : Predictor(_configMap, predictorType) {

    svmPath = directory + "/" + getParam<std::string>( svmPathKey );
    createSVM();
}

CvSVMPredictor::~CvSVMPredictor() {

}

void CvSVMPredictor::createSVM() {

        params.svm_type    = getParam<int>( svmType );
        params.kernel_type = getParam<int>( kernelType );
        params.term_crit   = cvTermCriteria(getParam<int>(termCrit), getParam<int>(maxIter)
, getParam<double>(epsilon));

        params.gamma = getParam<double>( gamma );
        params.C = getParam<double>( C );
        params.degree = getParam<int>( degree );

        std::cout <<params.C << " " << params.gamma<< std::endl;
}

void CvSVMPredictor::train(cv::Mat& data, const std::vector<int>& labels) {


        TRACE(logger, "SVM train: Starting");
        if( data.rows == 0 ) {

                throw std::logic_error("Empty mat has been submitted");
        }

        computeMaxValues(data);
    normaliseData(data);

    INFO(logger, "Training SVM.")
    SVM.train(data, cv::Mat(1, labels.size(), CV_32SC1, const_cast<int*>(&labels[0])), cv::
Mat(), cv::Mat(), params);

    TRACE(logger, "SVM train: Finished");
}

std::vector<float> CvSVMPredictor::predict(const cv::Mat& histogram) {

        assert(histogram.rows == 1);
        INFO(logger, "Classifying")
        TRACE(logger, "predict: Starting");

        cv::Mat data = histogram.clone();
        normaliseData(data);
    std::vector<float> predictions(class_number);
    predictions[SVM.predict(histogram)] = 1;

        TRACE(logger, "predict: Finished");
    return predictions;
```

```
}

void CvSVMPredictor::load() {

        TRACE(logger, "load: Starting");
        SVM.load(svmPath.c_str());

        loadVMax();
        TRACE(logger, "load: Finished");
}

void CvSVMPredictor::save() {

        TRACE(logger, "load: Starting");
        SVM.save(svmPath.c_str());
        INFO(logger, "SVM model saved: " + svmPath);

        saveVMax();
        TRACE(logger, "load: Finished");
}

} /* namespace semantic_tagger */




/*
 * CvSVMPredictor.h
 *
 *  Created on: August 22, 2013
 *      Author: Adam Kosiorek
 * Description: SVM predictor class declaration
 */

#ifndef SVMPREDICTER_H_
#define SVMPREDICTER_H_

#include "Predictor.h"

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/ml/ml.hpp>


namespace Tagger3D {

/**
 * SVM predictor class. Predicts to which category an image belongs to
 */
class CvSVMPredictor: public Predictor {
public:

        CvSVMPredictor();
        /**
         * Parametric constructor
         * @param _configMap - a map of configuration parameters
         */
        CvSVMPredictor(const std::map<std::string, std::string> &_configMap,
                       const std::string &predictorType = "predictor");

        /**
         * Default descrutor
         */
        virtual ~CvSVMPredictor();

        virtual void train(cv::Mat &data, const std::vector<int> &labels) override;
        virtual std::vector<float> predict(const cv::Mat &visualWords) override;
        virtual void load() override;
```

```
        virtual void save() override;

private:
        virtual void createSVM();

    CvSVMParams params;
    CvSVM SVM;

    // Configuration parameters
    std::string svmPath;
    std::string histogramPath;
    bool storeHistogram;
    int dictionarySize;

    //  Configuration keys
    const std::string svmType = moduleName + "svmType";
    const std::string kernelType = moduleName + "kernelType";
    const std::string termCrit = moduleName + "termCrit";
    const std::string svmPathKey = moduleName + "svmPath";
    const std::string epsilon = moduleName + "eps";
    const std::string maxIter = moduleName + "maxIter";
    const std::string degree = moduleName + "degree";
    const std::string gamma = moduleName + "gamma";
    const std::string C = moduleName + "C";

    const int svmMatType = CV_32F;

    const std::string normValuesFile = "maxValues.xml";
    const std::string key = "key";


};

} /* namespace semantic_tagger */
#endif /* SVMPREDICTER_H_ */


/*
 * LibSVMPredictor.cpp
 *
 *  Created on: Aug 22, 2013
 *      Author: Adam Kosiorek
 * Description: svm predictor implementation
 */

#include "LibSVMPredictor.h"

#include <algorithm>
#include <vector>
#include <cmath>
#include <iostream>

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

namespace Tagger3D {

LibSVMPredictor::LibSVMPredictor(const std::map<std::string, std::string> &_configMap,
                const std::string &predictorType)

        : Predictor(_configMap, predictorType),
        svmModel(nullptr) {

        model = io->getPath() + "/" + getParam<std::string>(modelKey);
        createSVM();
}
```

```
LibSVMPredictor::~LibSVMPredictor() {

        svm_free_and_destroy_model(&svmModel);
        svm_destroy_param(&params);
}

void LibSVMPredictor::createSVM() {

    params.svm_type = getParam<int>( svmTypeKey );
    params.kernel_type  = getParam<int>( kernelTypeKey );
    params.degree = NULL;//getParam<int>( degreeKey );
    params.gamma = getParam<double>( gammaKey );
    params.coef0 = NULL;//getParam<int>( coef0Key );
    params.eps = getParam<double>( epsKey );
    params.cache_size = getParam<int>( cache_sizeKey );
//    params.p = NULL;//getParam<double>( pKey );
    params.shrinking = getParam<int>( shrinkingKey );
    params.probability = getParam<int> (probabilityKey);
    params.nr_weight = getParam<int> (nr_weightKey);
    params.C = getParam<double>( CKey );
    params.weight_label = NULL;
    params.weight = NULL;

}

void LibSVMPredictor::train( cv::Mat &data, const std::vector<int> &labels) {
        INFO(logger, "Training SVM");
    TRACE(logger, "SVM train: Starting");

        if( data.empty()) {
                std::logic_error e("Cannot train the SVM without any data");
                ERROR(logger, "train: " << e.what());
                throw e;
        }

        int rows = data.rows;
        int cols = data.cols;

        computeMaxValues(data);
    normaliseData(data);

    svm_problem prob;
    prob.l =  rows;
    prob.x  = (struct svm_node**)malloc(rows * sizeof(struct svm_node*));
        for(int r = 0; r < rows; ++r) {
                prob.x[r] = (struct svm_node*)malloc((cols + 1) * sizeof(struct svm_node));
        }

        prob.y = (double*)malloc(rows * sizeof(double));
        const int* labPtr = &labels[0];
        for (int r = 0; r < rows; ++r) {
                prob.y[r] = (double)labPtr[r];
        }

    for (int r = 0; r < rows; ++r) {
        float *dataPtr = data.ptr<float>(r); // Get data from OpenCV Mat
        long counter = 0;
        for (int c = 0; c < cols; ++c) {
                if(dataPtr[c] != -1) {
                                prob.x[r][counter].index = c + 1;  // Index starts from 1;
Pre-computed kernel starts from 0
                                prob.x[r][counter].value = (double)dataPtr[c];
                                counter++;
                }
        }
        prob.x[r][counter].index = -1;   // End of line
    }
```

```
    // Train
    svmModel = svm_train(&prob, &params);
//      svm_save_model(model.c_str(), svmModel);
    //  Clean up
//    for(size_t r = 0; r < rows; ++r) {
//        free(prob.x[r]);
//    }
//    free(prob.x);
//    free(prob.y);
    TRACE(logger, "SVM train: Finished");
}

std::vector<float> LibSVMPredictor::predict(const cv::Mat &histogram) {

        assert(histogram.rows == 1);
        cv::Mat visualWords = histogram.clone();
        normaliseData(visualWords);

        std::vector<double> predictions(class_number);
        svm_node *svmVec = (struct svm_node *)malloc((dims + 1) * sizeof(struct svm_node));
        const float* hPtr = visualWords.ptr<float>();
        int counter = 0;
        for(int i = 0; i < dims; ++i) {
                if(hPtr[i] != -1) {
                svmVec[counter].index = i + 1;
                svmVec[counter].value = (double)hPtr[i];
                counter++;
                }
        }
        svmVec[counter].index = -1;
        svm_predict_probability(svmModel, svmVec, &predictions[0]);

        return std::vector<float>(predictions.begin(), predictions.end());
}

void LibSVMPredictor::load() {
        INFO(logger, "Loading SVM");
        TRACE(logger, "load: Starting");
        std::cout << model << std::endl;
        if((svmModel = svm_load_model(model.c_str())) == 0) {
                std::runtime_error e("Could not load the SVM model from " + model);
                ERROR(logger, e.what());
                throw e;
        }

        loadVMax();
        TRACE(logger, "load: Finished");
}

void LibSVMPredictor::save() {
        TRACE(logger, "save: Starting")
        svm_save_model(model.c_str(), svmModel);

        saveVMax();
        INFO(logger, "SVM model saved: " + model);
        TRACE(logger, "save: Finished")
}

} /* namespace Tagger3D */


/*
 * LibSVMPredictor.h
 *
 *  Created on: August 22, 2013
 *      Author: Adam Kosiorek
 * Description: SVM predictor class declaration
 */
```

```
#ifndef SVMPREDICTOR_H_
#define SVMPREDICTOR_H_

#include "Predictor.h"

#include <svm.h>
#include <opencv2/core/core.hpp>

namespace Tagger3D {

/**
 * SVM predictor class. Predicts to which category an image belongs to
 */
class LibSVMPredictor: public Predictor {
public:
        LibSVMPredictor() = delete;

        /**
         * Parametric constructor
         * @param _configMap - a map of configuration parameters
         */
        LibSVMPredictor(const std::map<std::string, std::string> &_configMap,
                        const std::string &predictorType = "predictor");

        /**
         * Default descrutor
         */
        virtual ~LibSVMPredictor();

        virtual void train(cv::Mat &data, const std::vector<int> &labels) override;
        virtual std::vector<float> predict(const cv::Mat &visualWords) override;
        virtual void load() override;
    virtual void save() override;

private:
    virtual void createSVM();

    // Configuration parameters
    std::string model;
    std::string histogramPath;

    bool storeHistogram;

    //  Configuration keys
    const std::string svmTypeKey = moduleName + "svmType";
    const std::string kernelTypeKey = moduleName + "kernelType";

    const std::string degreeKey = moduleName + "degree";
    const std::string gammaKey = moduleName + "gamma";
    const std::string coef0Key = moduleName + "coef0";
    const std::string epsKey = moduleName + "eps";
    const std::string cache_sizeKey = moduleName + "cacheSize";
    const std::string pKey = moduleName + "p";
    const std::string shrinkingKey = moduleName + "shrinking";
    const std::string probabilityKey = moduleName + "probability";
    const std::string nr_weightKey = moduleName + "nrWeight";
    const std::string CKey = moduleName + "C";
    const std::string modelKey = moduleName + "svmPath";

    svm_parameter params;
    svm_model* svmModel;
};

} /* namespace Tagger3d */
#endif /* SVMPREDICTOR_H_ */
```

```
/*
 * Predictor.cpp
 *
 *  Created on: Jul 4, 2013
 *      Author: Adam Kosiorek
 * Description: Predictor base class implementation
 */

#include "Predictor.h"

#include <opencv2/core/core.hpp>

namespace Tagger3D {

Predictor::Predictor(const std::map<std::string, std::string> &_configMap,
                const std::string predictorType)

        : ProcessObject(_configMap),
          moduleName(predictorType + separator) {


        logger = lgr::Logger::getLogger(loggerName);
        DEBUG(logger, "Creating Predictor");

        if( _configMap.empty() ) {

                throw std::invalid_argument("Empty configuration map");
        }
        io = IoUtils::getInstance();
        class_number = getParam<int>(class_numberKey);
        normalizationPath = getParam<std::string>(normalizationPathKey);
}

Predictor::~Predictor() {

        TRACE(logger, "Destroying Predictor");
}

void Predictor::normaliseData(cv::Mat &mat) {

        assert(!v_max.empty());
        mat = (mat / repeat(v_max,  mat.rows, 1)) * 2 - 1;
}

const cv::Mat Predictor::computeMaxValues(const cv::Mat& mat) {
//      v_max = cv::Mat(1, mat.cols, CV_32SC1);
        cv::reduce(mat, v_max, REDUCE_TO_ROW, CV_REDUCE_MAX);
        TRACE(logger, "v_max size = " << v_max.size());
        return v_max;
}

cv::Mat Predictor::confusionMatrix(const std::vector<int> &labels, const std::vector<int> &
predictions) const {

        int size = labels.size();
        int classes = labels[size-1] + 1;
        const int *lPtr = &labels[0];
        const int *pPtr = &predictions[0];
        cv::Mat confusionMatrix = cv::Mat::zeros(classes, classes, CV_8UC1);
        for(int i = 0; i < size; i++) {

                confusionMatrix.at<uchar>(lPtr[i], pPtr[i]) += 1;
        }

        cv::Mat classCount;
        cv::reduce(confusionMatrix, classCount, 1, CV_REDUCE_SUM, CV_32SC1);
        cv::Mat average(1, classes, CV_32FC1);
        float avg = 0;
```

```
        for(int i = 0 ; i < classes; i++) {
                average.at<float>(0, i) = float(confusionMatrix.at<uchar>(i, i)) / classCou
nt.at<int>(0, i) * 100;
                avg += confusionMatrix.at<uchar>(i, i);
        }
        avg /= size;

        std::cout << "Entries per class: " << std::endl << classCount << std::endl;
        std::cout << "Confusion Matrix:" << std::endl << confusionMatrix << std::endl;
        std::cout << "Averages: " << std::endl << average << std::endl;
        std::cout << "Average: " << avg * 100 << std::endl;
}

void Predictor::saveVMax() {

        io->saveCv(v_max, normalizationPath);
}

void Predictor::loadVMax() {

        v_max = io->loadCv<cv::Mat>(normalizationPath);
        dims = v_max.cols;
}

} /* namespace Tagger3D */


/*
 * Predictor.h
 *
 *      Created on: Jul 4, 2013
 *      Author: Adam Kosiorek
 *      Description: Predictor base class declaration
 */

#ifndef PREDICTOR_H_
#define PREDICTOR_H_

#include "ProcessObject.h"
#include "IoUtils.h"

#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

#include <vector>
#include <memory>

namespace Tagger3D {

/**
 * Base class for predictors.
 */
class Predictor : public ProcessObject {
public:

        Predictor() = delete;
        /**
         * Parametric constructor
         * @param _configMap    a key-value map of configuration parameters
         */
        Predictor(const std::map<std::string, std::string> &_configMap, const std::string p
redictorType);

        /**
         * Default descrutors
         */
        virtual ~Predictor();
```

```
        /**
         * Trains the predictor using specified vectors of visual words and their labels
         * @param       vecs a vector of vectors of visual words
         * @param       labels a vector of labels
         */
        virtual void train(cv::Mat &data, const std::vector<int> &labels) = 0;

        /**
         * Predics a class of an input image
         * @param       visualWords     a vec of visual words describing a single image
         * @return      void
         */
    virtual std::vector<float> predict(const cv::Mat &visualWords) = 0;

    virtual void load() = 0;
    virtual void save() = 0;

protected:

        virtual void createSVM() = 0;
        virtual void normaliseData(cv::Mat &mat);
        virtual const cv::Mat computeMaxValues(const cv::Mat& mat) ;
        cv::Mat confusionMatrix(const std::vector<int> &labels, const std::vector<int> &pre
dictions) const;
        void saveVMax();
        void loadVMax();

        /**
         * Name of the module in a config map
         */
        const std::string moduleName;
    std::shared_ptr<IoUtils> io;

    int class_number;
    int dims;
    std::string normalizationPath;

private:

        /**
         * Name of the logger
         */
        const std::string loggerName = "Main.Predictor";
        const std::string class_numberKey = moduleName + "classes";
        const std::string normalizationPathKey = moduleName + "normalizationPath";



         const int REDUCE_TO_ROW = 0;
         cv::Mat v_max;
};

} /* namespace Tagger3D */
#endif /* PREDICTOR_H_ */


/home/adam/workspace/Tagger3D/utils/src/Utilities


/*
 * IoUtils.cpp
 *
 *  Created on: 4 Oct 2013
 *      Author: Adam Kosiorek
 * Description:
 */

#include "IoUtils.h"
```

```
namespace Tagger3D {

IoUtils::IoUtils()
        : logger(lgr::Logger::getLogger(loggerName)) {
        DEBUG(logger, "Creating IoUtils");
}

IoUtils::~IoUtils() {
        DEBUG(logger, "Destroying IoUtils");
}

std::shared_ptr<IoUtils> IoUtils::getInstance() {

        static std::shared_ptr<IoUtils> io = std::shared_ptr<IoUtils>(new IoUtils());
        return io;
}

std::string IoUtils::makePath(const std::string &path, const std::string &filename) const {

        return path + "/" + filename;
}

void IoUtils::saveVectorMatBinaryStats(const std::vector<cv::Mat> &vec, const std::string &
filename) const {
        DEBUG(logger, "Saving std::vector<cv::Mat> statistics");


        std::string filepath = makePath(path, filename + stat);
        std::ofstream os(filepath, std::ios::binary);
        if(!os.is_open() || !os.good()) {
                std::runtime_error e("Unable to open file: " + filepath);
                ERROR(logger, "saveVectorMatBinaryStats: " << e.what());
                throw e;
        }

        int mats = vec.size();
        int cols = vec[0].cols;

        std::vector<int> sizes;
        sizes.reserve(mats);
        for(const auto &mat : vec)
                sizes.push_back(mat.rows);

        // write number of images, dims and number of keypoints per image
        os.write((char*)&mats, sizeof(int));
        os.write((char*)&cols, sizeof(int));
        os.write((char*)&sizes[0], sizes.size() * sizeof(int));
        os.close();
}

void IoUtils::initTextFile(const std::string& filename) {

        if(outStream.is_open()) {
                std::runtime_error e("Previus file has not been finalized");
                ERROR(logger, "initTextFile: " << e.what());
                throw e;
        }
        outStream.open(makePath(path, filename).c_str());
}

void IoUtils::finalizeTextFile() {

        outStream.close();
}

std::vector<int> IoUtils::loadVectorMatBinaryStats(const std::string &filename) const {
        DEBUG(logger, "Loading std::vector<cv::Mat> statistics");
```

```
        std::string filepath = makePath(path, filename + stat);
        std::ifstream is(filepath, std::ios::binary);
        if(!is.is_open() || !is.good()) {
                std::runtime_error e("Unable to open file: " + filepath);
                ERROR(logger, "loadVectorMatBinaryStats: " << e.what());
                throw e;
        }

        int mats = 0;
        int cols = 0;

        is.read((char*)&mats, sizeof(int));
        is.read((char*)&cols, sizeof(int));
        std::vector<int> sizes;
        sizes.resize(mats + 2);
        sizes[0] = mats;
        sizes[1] = cols;

        is.read((char*)&sizes[2], mats* sizeof(int));
        is.close();

        return sizes;
}

} /* namespace Tagger3D */


/*
 * IoUtils.h
 *
 *  Created on: 4 Oct 2013
 *      Author: Adam Kosiorek
 * Description:
 */

#ifndef IOUTILS_H_
#define IOUTILS_H_

#include "logger.h"
#include "TypeTraits.h"
#include "VectorIO.h"

#include <opencv2/core/core.hpp>
#include <fstream>
#include <stdexcept>
#include <memory>
#include<iterator>
 #include <iostream>

extern "C" {
    #include <vl/generic.h>
    #include <vl/fisher.h>
    #include <vl/gmm.h>
}

namespace Tagger3D {

/*
 *
 */
class IoUtils {

//      Methods ----------------------------------------------------------------
-------
public:
        virtual ~IoUtils();
```

```
    VlGMM* loadGMMText(const std::string &filename, int& dimension, int& numClusters) const
{
        FILE * pFile;
        pFile = fopen (filename.c_str(),"r");
        fscanf (pFile, "dimension: %d numClusters: %d", &dimension, &numClusters);
        int& DIM = dimension;
        int& NUMCLUST = numClusters;
        VlGMM* gmm = vl_gmm_new (VL_TYPE_FLOAT, DIM, NUMCLUST) ;

        //loading means
        float* means = new float[DIM*NUMCLUST];
        for(int clust = 0; clust < NUMCLUST; ++clust){
            for(int dim = 0; dim < DIM; ++dim){
                fscanf(pFile, "%f", &means[clust*DIM+dim]);
            }
        }
        vl_gmm_set_means (gmm, means);
        delete [] means;

        //loading priors
        float* priors = new float[DIM*NUMCLUST];
        for(int clust = 0; clust < NUMCLUST; ++clust){
            for(int dim = 0; dim < DIM; ++dim){
                fscanf(pFile, "%f", &priors[clust*DIM+dim]);
            }
        }
        vl_gmm_set_priors (gmm, priors);
        delete [] priors;

        //loading covariances
        float* covariances = new float[DIM*NUMCLUST];
        for(int clust = 0; clust < NUMCLUST; ++clust){
            for(int dim = 0; dim < DIM; ++dim){
                fscanf(pFile, "%f", &covariances[clust*DIM+dim]);
            }
        }
        vl_gmm_set_covariances(gmm, covariances);
        delete [] covariances;

        fclose (pFile);
        return gmm;
    }


    int saveGMMText(const VlGMM& gmm, const std::string &filename, int dimension, int numCl
usters) const{
        std::fstream fs;
        //std::cout<<"filename: "<<filename<<std::endl;
        fs.open (filename, std::fstream::out);
        if (fs.is_open()){
            fs << "dimension: "<<dimension<< " numClusters: "<<numClusters<<"\n";
            float* means = (float*)vl_gmm_get_means (&gmm);
            for(int i = 0; i< numClusters; ++i){
                for(int j =0; j<dimension;++j){
                    fs << means[i*dimension+j]<< " ";
                }
                fs<<"\n";
            }
            float* priors = (float*)vl_gmm_get_priors (&gmm);
            for(int i = 0; i< numClusters; ++i){
                for(int j =0; j<dimension;++j){
                    fs << priors[i*dimension+j]<< " ";
                }
                fs<<"\n";
            }
            float* covariances = (float*)vl_gmm_get_covariances (&gmm);
            for(int i = 0; i< numClusters; ++i){
                for(int j =0; j<dimension;++j){
```

```
                    fs << covariances[i*dimension+j]<< " ";
                }
                fs<<"\n";
            }
            fs.close();
        }
        else{
            //std::cout << "Error opening file";
        }

        return 1;

    }



        template<typename T>
        void saveCv(const T &obj, const std::string &filename) const;

        template<typename T>
        T loadCv(const std::string &filename) const;

        template<typename T>
        void saveMatBinary(const cv::Mat &mat, const std::string &filename, bool saveStats
= false) const;

        template<typename T>
        cv::Mat loadMatBinary(const std::string &filename) const;

        template<typename T>
        cv::Mat loadMatBinary(const std::string &filename, int rows, int cols) const;

        template<typename T, typename E = void>
        void saveVector(const std::vector<T> &vec, const std::string& filename) const;

        template<typename T, typename E = void>
        std::vector<T> loadVector(const std::string& filename) const;

        void initTextFile(const std::string &filename);
        void finalizeTextFile();

        template<typename T>
        void appendToTextFile(const std::vector<T> &vec);

        template<typename T>
        void appendToTextFile(const T &el);

    template <typename T>
    std::vector<std::vector<T>> loadTextFileAsMat(const std::string& filename);


        const std::string& getPath() const {
                return path;
        }

        void setPath(const std::string& path) {
                this->path = path;
        }

        static std::shared_ptr<IoUtils> getInstance();

private:
        IoUtils();
        IoUtils(const IoUtils &);
        std::string makePath(const std::string &path, const std::string &filename) const;

        void saveVectorMatBinaryStats(const std::vector<cv::Mat> &vec, const std::string &f
```

```
ilename) const;
        std::vector<int> loadVectorMatBinaryStats(const std::string &filename) const;

        template<typename T>
        std::string appendDataType(const std::string &filename) const;

//      Fields  ------------------------------------------------------------------------
-------
public:
private:

        std::string path;
        const std::string key = "key";
        const std::string stat = ".stat";
        const std::string cvExtension = ".yml";

        const std::string loggerName = "Main.IoUtils";
        const lgr::LoggerPtr logger;

        std::ofstream outStream;
        const std::string sep = " ";
        const std::string newline = "\n";
};

template<typename T>
inline void IoUtils::saveCv(const T& obj, const std::string& filename) const {
        TRACE(logger, "Saving an object under \"" << filename << "\"");

        std::string filepath = makePath(path, filename + cvExtension);
        cv::FileStorage fs(filepath, cv::FileStorage::WRITE);
        if( !fs.isOpened()) {
                std::runtime_error e("Cannot initialise a cv::FileStorage at " + filepath);
                ERROR(logger, "saveCV: " << e.what());
                throw e;
        }

        fs << key << obj;
        fs.release();
}

template<typename T>
inline T IoUtils::loadCv(const std::string& filename) const {
        TRACE(logger, "Loading an object from \"" << filename << "\"");

        std::string filepath = makePath(path, filename + cvExtension);
        cv::FileStorage fs(filepath, cv::FileStorage::READ);
        if( !fs.isOpened()) {
                if( !fs.open(filename, cv::FileStorage::READ)) {
                        std::runtime_error e("Could not open the following file: " + filepa
th);
                        ERROR(logger, ": " << e.what());
                        throw e;
                }
        }

        T obj;
        try {
                fs[this->key] >> obj;
        } catch(...) {

                std::runtime_error e("Cannot read the filestorage: " + path);
        }
        return obj;
}

template<typename T>
inline void IoUtils::saveMatBinary(const cv::Mat& mat,  const std::string& filename, bool s
aveStats) const {
```

```
        TRACE(logger, "Saving a cv::Mat in a binary file \"" << filename << "\"")

        if(!mat.data)
                std::runtime_error e("Corrupted image");

        std::string filepath = makePath(path, appendDataType<T>(filename));
        std::ofstream os(filepath, std::ios::binary);
        if(!os.is_open() || !os.good()) {
                std::runtime_error e("Unable to open file: " + filepath);
                ERROR(logger, "saveMatBinary: " << e.what());
                throw e;
        }

        int rows = mat.rows;
        int cols = mat.cols;

        // write rows and cols
        if(saveStats) {
                os.write((char*)&rows, sizeof(int));
                os.write((char*)&cols, sizeof(int));
        }

        for(int i = 0; i < rows; ++i)
                os.write((char*)mat.ptr<T>(i), cols * sizeof(T));

        os.close();
}

template<typename T>
inline cv::Mat IoUtils::loadMatBinary(const std::string& filename) const {
        TRACE(logger, "Loading a cv::Mat from a binary file \"" << filename << "\"");

        std::string filepath = makePath(path, appendDataType<T>(filename));
        std::ifstream is(filepath, std::ios::binary);
        if(!is.is_open() || !is.good()) {
                std::runtime_error e("Unable to open file: " + filepath);
                ERROR(logger, "loadMatBinary: " << e.what());
                throw e;
        }

        int rows, cols;

        // read rows and cols data
        is.read((char*)&rows, sizeof(int));
        is.read((char*)&cols, sizeof(int));
        cv::Mat mat(rows, cols, CvTypeTraits<T>::type());

        for(int i = 0; i < rows; ++i)
                is.read((char*)mat.ptr<T>(i), cols * sizeof(T));

        is.close();

        if(!mat.data)
                std::runtime_error e("Corrupted image");

        return mat;
}

template<typename T>
inline cv::Mat IoUtils::loadMatBinary(const std::string& filename, int rows, int cols) cons
t {
        TRACE(logger, "Loading a cv::Mat from a binary file \"" << filename << "\"");

        std::string filepath = makePath(path, appendDataType<T>(filename));
        std::ifstream is(filepath, std::ios::binary);
        if(!is.is_open() || !is.good()) {
                std::runtime_error e("Unable to open file: " + filepath);
                ERROR(logger, "loadMatBinary: " << e.what());
```

```
                throw e;
        }
        cv::Mat mat(rows, cols, CvTypeTraits<T>::type());

        for(int i = 0; i < rows; ++i)
                is.read((char*)mat.ptr<T>(i), cols * sizeof(T));

        is.close();

        if(!mat.data)
                std::runtime_error e("Corrupted image");

        return mat;
}

template<typename T>
inline void IoUtils::appendToTextFile(const std::vector<T>& vec) {

        if(!outStream.is_open() || !outStream.good()) {
                std::runtime_error e("File has not been initialized");
                ERROR(logger, "appendToTextFile: " << e.what());
                throw e;
        }
        std::copy(vec.begin(), vec.end(), std::ostream_iterator<T>(outStream, sep.c_str()))
;
        outStream << newline;
}

template<typename T>
inline void IoUtils::appendToTextFile(const T& el) {

        if(!outStream.is_open() || !outStream.good()) {
                std::runtime_error e("File has not been initialized");
                ERROR(logger, "appendToTextFile: " << e.what());
                throw e;
        }
        outStream << el << sep.c_str();
}

template<typename T>
inline std::vector<std::vector<T>> IoUtils::loadTextFileAsMat(const std::string& filename)
{
    std::vector<std::vector<float>> result_mat;
    std::ifstream source;                    // build a read-Stream
    source.open(filename, std::ios_base::in);  // open data
    if (!source)  {                      // if it does not work
        std::cerr << "Can't open Data!\n";
    }
    for(std::string line; std::getline(source, line); ){   //read stream line by line
        line.erase(std::remove(line.begin(), line.end(), '\n'), line.end());
        std::istringstream reader(line);      //make a stream for the line itself
        std::vector<T> values_vec;
        do
        {
            // read as many numbers as possible.
            int i = 0;
            for (T value; reader >> value;) {
                //std::cout<< "value "<<i<<" " << value<<std::endl;
                values_vec.push_back(value);
                i++;
            }
            result_mat.push_back(values_vec);
            // consume and discard token from stream.
            if (reader.fail())
            {
                reader.clear();
                std::string token;
                reader >> token;
```

```
            }
        }
        while (!reader.eof());
    }
    return result_mat;
}


template<typename T>
std::string IoUtils::appendDataType(const std::string &filename) const {

        return filename + "_" + TypeStr<T>::str();
}

template<typename T, typename E>
void IoUtils::saveVector(const std::vector<T> &vec, const std::string& filename) const {

        std::string filepath = makePath(path, appendDataType<E>(filename));
        VectorIO<T, E>::save(vec, filepath);
}

template<typename T, typename E>
std::vector<T> IoUtils::loadVector(const std::string& filename) const {

        std::string filepath = makePath(path, appendDataType<E>(filename));
        return VectorIO<T, E>::load(filepath);
}


} /* namespace Tagger3D */
#endif /* IOUTILS_H_ */


/*
 * CvTypeTraits.h
 *
 *  Created on: 4 Oct 2013
 *      Author: Adam Kosiorek
 * Description:
 */

#ifndef CVTYPETRAITS_H_
#define CVTYPETRAITS_H_

#include <opencv2/core/types_c.h>
#include <opencv2/core/core.hpp>
#include <stdexcept>

//      CvTypeTraits ----------------------------------------------------------------------
---------------

template<typename T> struct CvTypeTraits {
static int type() { throw std::logic_error("Unimplemented"); }
};

template<> struct CvTypeTraits<uchar> {
        static int type() {     return CV_8UC1; }
};

template<> struct CvTypeTraits<ushort> {
        static int type() {     return CV_16UC1; }
};

template<> struct CvTypeTraits<int> {
        static int type() {     return CV_32SC1; }
};

template<> struct CvTypeTraits<float> {
```

```
        static int type() {     return CV_32FC1; }
};

template<> struct CvTypeTraits<double> {
        static int type() {     return CV_64FC1; }
};


//      CvMatTraits ---------------------------------------------------------------
--------------


template<int i> struct CvMatTraits {
        typedef int type;
};

template<> struct CvMatTraits<CV_8UC1> {
        typedef uchar type;
};

template<> struct CvMatTraits<CV_16UC1> {
        typedef ushort type;
};

template<> struct CvMatTraits<CV_32SC1> {
        typedef int type;
};

template<> struct CvMatTraits<CV_32FC1> {
        typedef float type;
};

template<> struct CvMatTraits<CV_64FC1> {
        typedef double type;
};

//      Type traits ---------------------------------------------------------------
--------------

//      Vector
template<typename T>
struct isVector{ static const int value = 0; };

template<typename T, typename A>
struct isVector<std::vector<T, A>>{ static const int value = 1; };

//      Mat
template<typename T>
struct isMat{ static const int value = 0; };

template<>
struct isMat<cv::Mat>{ static const int value = 1; };

//      Type names ----------------------------------------------------------------
--------------

template<typename T> struct TypeStr {
        static std::string str() { throw std::logic_error("Unimplemented"); };
};

template<> struct TypeStr<void> {
        static std::string str() { return ""; }
};

template<> struct TypeStr<uchar> {
        static std::string str() { return "uchar"; }
};
```

```
template<> struct TypeStr<ushort> {
        static std::string str() { return "ushort"; }
};

template<> struct TypeStr<uint> {
        static std::string str() { return "uint"; }
};

template<> struct TypeStr<int> {
        static std::string str() { return "int"; }
};

template<> struct TypeStr<float> {
        static std::string str() { return "float"; }
};

template<> struct TypeStr<double> {
        static std::string str() { return "double"; }
};


#endif /* CVTYPETRAITS_H_ */


/*
 * Utils.cpp
 *
 *  Created on: 21 Nov 2013
 *      Author: Adam Kosiorek
 * Description:
 */

#include "Utilities/Utils.h"

#include "opencv2/highgui/highgui.hpp"

#include <stdexcept>

namespace Tagger3D {

Utils::Utils() {
        // TODO Auto-generated constructor stub

}

Utils::~Utils() {
        // TODO Auto-generated destructor stub
}


int Utils::visualizeKeypoints(const cv::Mat &img, std::vector<cv::KeyPoint> &keys){

    cv::Mat imgWithKeys;    // save image with visualised keypoints
    try {
                cv::drawKeypoints(img, keys, imgWithKeys);
                cv::imshow("KeyPoints Visualisation", imgWithKeys);
    } catch(std::runtime_error &e) {
        return 0;
    }
    cv::waitKey(0);
    return 1;
}

std::string Utils::cvType2Str(int type) {
  std::string r;

  uchar depth = type & CV_MAT_DEPTH_MASK;
  uchar chans = 1 + (type >> CV_CN_SHIFT);
```

```
  switch ( depth ) {
    case CV_8U:  r = "8U"; break;
    case CV_8S:  r = "8S"; break;
    case CV_16U: r = "16U"; break;
    case CV_16S: r = "16S"; break;
    case CV_32S: r = "32S"; break;
    case CV_32F: r = "32F"; break;
    case CV_64F: r = "64F"; break;
    default:     r = "User"; break;
  }

  r += "C";
  r += (chans+'0');

  return r;
}

} /* namespace Tagger3D */


/*
 * Utils.h
 *
 *  Created on: 21 Nov 2013
 *      Author: Adam Kosiorek
 * Description:
 */

#ifndef UTILS_H_
#define UTILS_H_

#include <opencv2/core/core.hpp>
#include <opencv2/features2d/features2d.hpp>

#include <iostream>
#include <vector>


namespace Tagger3D {

/*
 *
 */
class Utils {
public:
        Utils();
        virtual ~Utils();

        /**
         * Utility method for vector's type conversion
         * @param       class input_type        type of an input vector
         * @param       class output_type       type of an output vector
         * @param       vec     vector whose type is to be changed
         * @return      vector beign a copy of the vec but of different type
         */
        template <typename output_type, typename input_type>
        static std::vector<output_type> convertVec (std::vector<input_type> vec) {
                return std::vector<output_type>(vec.begin(), vec.end());
        }

        /**
         * @brief       visualizeKeypoints in image
         * @param       keys    keypoints to visualize
         * @param       img     image that we works on
         * @return      1 on success
         */
        static int visualizeKeypoints(const cv::Mat &img, std::vector<cv::KeyPoint> &keys);
```

```
        /**
         *      Converts int returned by cv::Mat.type() to a human readable std::string
         *      @param  type    type returned by cv::Mat.type()
         *      @return human readable type string.
         */
        static std::string cvType2Str(int type);

};

} /* namespace Tagger3D */
#endif /* UTILS_H_ */


/*
 * vectors.h
 *
 *  Created on: 6 Nov 2013
 *      Author: Adam Kosiorek
 * Description:
 */

#ifndef VECTORS_H_
#define VECTORS_H_

#include "TypeTraits.h"
#include "logger.h"

#include <opencv2/core/core.hpp>

#include <stdexcept>
#include <vector>
#include <type_traits>
#include <fstream>
#include <iostream>

//      Default ------------------------------------------------------------------------
-------------------------------------------------------
template<typename T, typename E = void> struct VectorIO {



        static void save(const std::vector<T> &vec, const std::string &filepath) {

                static_assert(std::is_fundamental<T>::value || std::is_same<std::string, T>
::value, "Algorithm not defined for the type specified");
                std::ofstream os(filepath, std::ios::binary);
                if(!os.is_open() || !os.good()) {
                        std::runtime_error e("Unable to open file: " + filepath);
                        throw e;
                }

                size_t elems = vec.size();
                os.write((char*)&elems, sizeof(size_t));
                os.write((char*)&vec[0], elems * sizeof(T));
                os.close();
        }

        static std::vector<T> load(const std::string &filepath) {

                static_assert(std::is_fundamental<T>::value || std::is_same<std::string, T>
::value, "Algorithm not defined for the type specified");
                std::ifstream is(filepath, std::ios::binary);
                if(!is.is_open() || !is.good()) {
                        std::runtime_error e("Unable to open file: " + filepath);
                        throw e;
                }
                size_t elems;
```

```
                is.read((char*)&elems, sizeof(size_t));
                std::vector<T> vec(elems);
                is.read((char*)&vec[0], elems * sizeof(T));
                return vec;
        }
};

//      Vector of Vectors ------------------------------------------------------------------
-------------------------------------------------------------
template<typename T> struct VectorIO<std::vector<T>> {

        typedef uint size_type;
        static void save(const std::vector<std::vector<T>> &vec, const std::string &filepat
h) {

                static_assert(std::is_fundamental<T>::value
                                || std::is_same<std::string, T>::value, "Algorithm not defi
ned for the type specified");

                std::ofstream os(filepath, std::ios::binary);
                if(!os.is_open() || !os.good()) {
                        std::runtime_error e("Unable to open file: " + filepath);
                        throw e;
                }

                size_type elems = vec.size();
                std::vector<size_type> sizes;
                sizes.reserve(elems);
                for(const auto &subvec : vec)
                        sizes.push_back(subvec.size());


                os.write((char*)&elems, sizeof(size_type));
                os.write((char*)&sizes[0], elems * sizeof(size_type));
//              std::for_each(std::begin(vec), std::end(vec),
//                              [&](const T &subvec) {os.write((char*)&subvec[0], subvec.si
ze() * sizeof(T));});
                for(const auto &subvec : vec)
                        os.write((char*)&subvec[0], subvec.size() * sizeof(T));

                os.close();
        }

        static std::vector<std::vector<T>> load(const std::string &filepath) {

                static_assert(std::is_fundamental<T>::value
                                || std::is_same<std::string, T>::value, "Algorithm not defi
ned for the type specified");

                        std::ifstream is(filepath, std::ios::binary);
                        if(!is.is_open() || !is.good()) {
                                std::runtime_error e("Unable to open file: " + filepath);
                                throw e;
                        }

                        std::vector<std::vector<T>> vec2;
                        size_type elems;
                        is.read((char*)&elems, sizeof(size_type));
                        std::vector<size_type> sizes(elems);
                        is.read((char*)&sizes[0], elems * sizeof(size_type));
                        for(int i = 0; i < elems; i++) {

                                std::vector<T> vec(sizes[i]);
                                is.read((char*)&vec[0], sizes[i] * sizeof(T));
                                vec2.push_back(vec);
                        }

                        return vec2;
```

```
	}
};

//	Vector of Mats ------------------------------------------------------------
------------------------------------------------------------
template<typename E> struct VectorIO<cv::Mat, E> {

	typedef size_t size_type;

private:
	static void saveVectorMatBinaryStats(const std::vector<cv::Mat> &vec, const std::st
ring &filepath) {

		std::ofstream os(filepath, std::ios::binary);
		if(!os.is_open() || !os.good()) {
			std::runtime_error e("Unable to open file: " + filepath);
			throw e;
		}

		size_type mats = vec.size();

		std::vector<size_type> cols;
		std::vector<size_type> rows;
		cols.reserve(mats);
		rows.reserve(mats);
		for(const auto &mat : vec) {
			rows.push_back(mat.rows);
			cols.push_back(mat.cols);
		}

		// write number of images, dims and number of keypoints per image
		os.write((char*)&mats, sizeof(size_type));
		os.write((char*)&cols[0], mats * sizeof(size_type));
		os.write((char*)&rows[0], mats * sizeof(size_type));
		os.close();
	};

	static std::pair<std::vector<size_type>, std::vector<size_type>> loadVectorMatBinar
yStats(const std::string &filepath) {

		std::ifstream is(filepath, std::ios::binary);
		if(!is.is_open() || !is.good()) {
			std::runtime_error e("Unable to open file: " + filepath);
			throw e;
		}

		size_type mats = 0;
		is.read((char*)&mats, sizeof(size_type));
		std::vector<size_type> rows(mats);
		std::vector<size_type> cols(mats);
		is.read((char*)&cols[0], mats * sizeof(size_type));
		is.read((char*)&rows[0], mats * sizeof(size_type));

		for(int i = 0 ; i < mats; i++) {
			std::cout << "cols = " << cols[i] << " rows = " << rows[i] << std::
endl;
		}

		is.close();

		return std::make_pair(rows, cols);
	}

public:

	static void save(const std::vector<cv::Mat> &vec, const std::string &filepath) {

		static_assert(std::is_arithmetic<E>::value, "Algorithm not defined for the
```

```
type specified");
                if(vec.empty()) {
                        std::runtime_error e("Empty vector");
                        throw e;
                }

                std::ofstream os(filepath, std::ios::binary);
                if(!os.is_open() || !os.good()) {
                        std::runtime_error e("Unable to open file: " + filepath);

                        throw e;
                }
                int mats = vec.size();
                int cols = vec[0].cols;


                for(const auto& mat : vec)
                        os.write((char*)mat.data, mat.rows * mat.cols * sizeof(E));
                os.close();

                saveVectorMatBinaryStats(vec, filepath + ".stat");
        }

        static std::vector<cv::Mat> load(const std::string &filepath) {

                static_assert(std::is_arithmetic<E>::value, "Algorithm not defined for the
type specified");

                std::ifstream is(filepath, std::ios::binary);
                if(!is.is_open() || !is.good()) {
                        std::runtime_error e("Unable to open file: " + filepath);
                        throw e;
                }

                auto rows_cols = loadVectorMatBinaryStats(filepath + ".stat");
                auto &rows = rows_cols.first;
                auto &cols = rows_cols.second;
                size_type mats = rows.size();
                std::vector<cv::Mat> vec;
                vec.reserve(mats);

                for(int i = 0; i < mats; i++) {

                        cv::Mat mat(rows[i], cols[i], CvTypeTraits<E>::type());
                        is.read((char*)mat.data, rows[i] * cols[i] * sizeof(E));

                        if(!mat.data) {
                                std::runtime_error e("Corrupted image #: " + std::to_string
(i));
                                throw e;
                        }

                        vec.push_back(mat.clone());
                }
                return vec;
        }
};

#endif /* VECTORS_H_ */
```