

华中科技大学计算机科学与技术学院

机器学习结课项目报告

人脸识别系统

- 计科校交1801
- 车春池
- U201816030

引言

课题背景及意义

人脸识别，是基于人的脸部特征信息进行身份识别的一种生物识别技术。用摄像机或者摄像头采集含有人脸的图像或视频流，并自动在图像中检测和跟踪人脸，进而对检测到的人脸进行脸部识别的一系列相关技术。

人脸识别是图像分析与理解的一种最成功的应用，因其在商业，安全，身份认证，法律执行等众多方面的广泛应用，以及对人脸识别技术可行性的三十多年研究，使其越来越得到重视，并逐渐成为一个充满活力的研究领域。

随着人工智能领域特别是深度学习领域的发展，人脸识别技术逐渐走向成熟，进入了初级的应用阶段。

本项目将顺应人工智能发展的潮流，利用本学期机器学习课程上学到的知识，结合本人的专业技能，打造一个轻量型人脸识别系统，可用于家庭，学校，公司等场所。

另外，本项目将会把完善好的人脸识别系统移植到树莓派上，树莓派接上摄像头，实现人脸识别技术的落地。

可行性分析

人脸识别的四个步骤：

1. 人脸检测

2. 人脸对齐
3. 人脸编码
4. 人脸匹配

人脸识别的五个难点：

1. 头部姿势
2. 年龄
3. 遮挡
4. 光照条件
5. 人脸表情

经过多年的发展，人脸识别技术已有多种算法和模型支撑，基于独立成分分析和核向量的人脸识别算法，基于双向PCA和k近邻的人脸识别算法，基于深度学习卷积神经网络的人脸识别算法等等。而人脸识别的前三个步骤已有不少开源框架可以较好地完成任务，人脸匹配也已经有许多已经较为成熟的算法能够实现。

系统需求分析

系统功能需求

1. 实现人脸识别的四个步骤：人脸检测，人脸对齐，人脸编码，人脸匹配功能
2. 能进行实时人脸识别，测试时，人脸库中的照片可以是家庭成员，也可以是班级同学
3. 可以调整人脸库中单个人注册的人脸照片数
4. 可以设置不同距离函数和核函数
5. 测试不同参数设置情况下，参数变化对人脸识别系统的影响

系统性能需求

1. 每张图片提取特征值时间为毫秒级别
2. 在数千张图片的训练集上的准确度 $\geq 95\%$
3. 在目前多个主流的人脸数据集上测试准确度 $\geq 92\%$
4. 移植树莓派实现小型人脸识别系统

跨平台性需求

该人脸识别系统能在目前多个主流的操作系统Linux,Mac,Windows上编译。

系统实现原理

OpenCV

OpenCV的全称是Open Source Computer Vision Library，是一个跨平台的计算机视觉库，由英特尔公司发起并参与开发，可用于开发实时的图像处理，计算机视觉以及模式识别程序。



OpenCV用C++语言编写，它的主要接口也是C++语言，但是依然保留了大量的C语言接口。该库也有大量的Python, JAVA的接口。

操作系统支持Linux, Mac

OS,iOS,OpenBSD,FreeBSD,Maemo,Andorid,Windows，使用者可以在Github获得官方版本，或者从Git获得开发者版本，使用CMake编译。

中科视拓SeetaFace2开源框架

SeetaFace2人脸识别引擎包括了搭建一套全自动人脸识别系统所需的三个核心模块，即：人脸检测模块FaceDetector,面部关键点定位模块FaceLandmarker以及人脸特征提取与比对模块FaceRecognizer。以及两个辅助模块FaceTracker和QualityAssessor用于人脸跟踪和质量评估。



人脸检测 ==> 关键点定位 ==> 人脸特征提取 ==> 特征对比
SeetaFace2采用标准C++开发，全部模块均不依赖第三方库，支持x86架构

(Linux, Windows) 和Arm架构。SeetaFace2支持的上层应用包括但不限于人脸识别，无感考勤，人脸比对等。



人脸识别



人脸闸机



动态识别



人脸检索



人脸门禁



人证对比

SeetaFace框架性能：

模块	方法概述	基础技术指标	典型平台速度
人脸检测	Cascaded CNN	Fddb 上召回率达到 92% (100个误检情况下)。	I7: 70FPS(1920x1080) RK3399: 25FPS(640x480)
面部关键点定位(81点和5点)	FEC-CNN	平均定位误差 (根据两眼中心距离归一化) 300-W Challenge Set 上达到 0.069。	I7: 450FPS 和 500FPS RK3399: 110FPS 和 220FPS
人脸特征提取与比对	ResNet50	识别：通用1：N+1场景下，错误接受率1%时，1000人底库，首选识别率超过98%，5000人底库，首选识别率超过95%。	I7: 8FPS RK3399: 2.5FPS

本项目将使用SeetaFace2开源框架完成人脸检测，面部关键点定位，人脸特征提取等工作。

CMake

CMake是一个开源的跨平台自动化建构系统，用来管理软件建置的程序，并不依赖与某特定的编译器，并可支援多层目录，多个应用程序与多个函数库。它用配置文件构建过程（build process）的方式和Unix的make相似，只是CMake的配置文件取名为CmakeLists.txt。CMake并不直接构建出最终的软件，而是产生标准的配置文件（如Unix的Makefile或Windows Visual C++的projects/workspaces），然后再依一般的建构方式使用。



使用CMake的软件有Boost C++ Libraries,Blender 3D,OpenSceneGraph等等，还有就是本项目使用的OpenCV,SeetaFace2。

本项目基于CMake进行编译，因此具备跨平台性，可以在目前主流的操作系统Linux,Mac OS,Windows上编译，后面能将该项目移植到树莓派上运行，也是得益于CMake的存在。

KNN算法原理

在模式识别领域中，最近邻居法（即KNN算法，又称K-近邻算法）是一种用于分类和回归的统计方法。本项目中利用KNN算法进行分类任务。输入包含特征空间中的K个最接近的训练样本。

- 在K-NN分类中，输出是一个分类族群。一个对象的分类是由其邻居的“多数表决”确定的，k个最近邻居中最常见的分类决定了赋予该对象的类别。若k=1，则该对象的类别直接由最近的一个节点赋予。
- 在K-NN回归中，输出是该对象的属性值，该值是其k个邻居的值的平均值

KNN算法采用向量空间模型来分类，概念为相同类别的案例，彼此的相似度高，而可以借由计算与已知的类别的样本之相似度，来评估未知类别的样本的分类。

k最近邻分类器可以被视为k最近邻居分配权重 $1/k$ 以及为所有其他邻居分配0权重。这可以推广到加权最近邻分类器。也就是说，第*i*近的邻居被赋予权重 w_{ni} ，其中 $\sum_{i=1}^n w_{ni} = 1$ 。关于加权最近邻分类器的强一致性的类似结果也成立。
设 C_n^{wnn} 表示权重为 $\{w_{ni}\}_{i=1}^n$ 的加权最近邻分类器。根据类别分布的规律性条件，超额风险具有以下渐近展开。

$$R_R(C_n^{wnn}) - R_R(C^{Bayes}) = (B_1 s_n^2 + B_2 t_n^2) \{1 + o(1)\} \quad (1)$$

对常数 B_1 and B_2 当 $s_n^2 = \sum_{i=1}^n w_{ni}^2$ 并且

$$t_n = n^{-2/d} \sum_{i=1}^n w_{ni} \{i^{1+2/d} - (i-1)^{1+2/d}\}$$

最佳加权方案 $\{w_{ni}^*\}_{i=1}^n$ 用于平衡上面显示中的两个项，如下所示：

$$k^* = \lfloor B_n^{4/(d+4)} \rfloor \quad (2)$$

$$w_{ni}^* = (1/k^*)[1 + d/2 - (d/2k^{*2/d})\{i^{1+2/d} - (i-1)^{1+2/d}\}], i = 1, 2, \dots, k^* \quad (3)$$

$$w_{ni}^* = 0, i = k^* + 1, \dots, n \quad (4)$$

利用最优权重，超额风险的渐近展开中的主项是 $O(n^{-4/d+4})$ 。

在KNN算法中，有两种常用的距离，分别为曼哈顿距离和欧式距离。

设特征空间 χ 是 n 维实数向量空间

$R_n, x_i, x_j \in \chi, x_i = (x_i^{(1)}, x_i^{(2)}, \dots, x_i^{(n)})^T, x_j = (x_j^{(1)}, x_j^{(2)}, \dots, x_j^{(n)})^T, x_i, x_j$ 的 L_p 距离定义为：

$$L_p(x_i, x_j) = (\sum_{l=1}^n |x_i^{(l)} - x_j^{(l)}|^p)^{1/p}, p >= 1 \quad (5)$$

$p = 1$ 时，称为曼哈顿距离(Manhattan distance)，公式为：

$$L_1(x_i, x_j) = \sum_{l=1}^n |x_i^{(l)} - x_j^{(l)}| \quad (6)$$

$p = 2$ 时，称为欧式距离(Euclidean distance)，即：

$$L_2(x_i, x_j) = (\sum_{l=1}^n |x_i^{(l)} - x_j^{(l)}|^2)^{1/2} \quad (7)$$

$p = \infty$ 时，它是各个坐标距离的最大值，计算公式为：

$$L_\infty(x_i, x_j) = \max_l |x_i^{(l)} - x_j^{(l)}| \quad (8)$$

整理以下KNN算法流程：

输入：训练数据集：

$$T = (x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n) \quad (9)$$

其中， $x_i \in \chi \subseteq R^n$ 为实例的特征向量， $y_i \in \Upsilon = c_1, c_2, \dots, c_k$ 为实例的类别， $i = 1, 2, \dots, N$ ；实例特征向量 x ；

输出：实例 x 所属的类别 y

1. 根据给点的距离度量，在训练集 T 中找出与 x 最近邻的 k 个点，蕴含着 k 个点的领域，记为 $N_k(x)$ ；
2. 在 $N_k(x)$ 中根据决策规则，决定 x 的类别 y ：

$$y = \operatorname{argmax}_{c_j} \sum_{x_i \in N_k(x)} I(y_i = c_j), i = 1, 2, 3, \dots, N \quad (10)$$

核方法，核技巧与核函数

核方法是一类把低维空间的非线性可分问题，转化为高维空间的线性可分问题的方法。核方法不仅仅用于SVM，还可以用于其他数据为非线性可分的算法。

核方法的理论基础是Cover's theorem，指的是对于非线性可分的训练集，可以大概率通过将其非线性映射到一个高维空间来转化为线性可分的训练集。

定义核函数：

设 χ 是输入空间 ($x_i \in \chi, \chi$ 是 R^n 的子集或离散集合)，又设 H 为特征空间 (H 是希尔伯特空间)，如果存在一个从 χ 到 H 的映射

$$\Phi(x) : \chi \rightarrow H \quad (11)$$

使得对所有 $x, z \in \chi$ ，函数 $K(x, z)$ 满足条件

$$K(x, z) = \langle \Phi(x), \Phi(z) \rangle \quad (12)$$

则称 K 为核函数。其中 $\Phi(x)$ 为映射函数， $\langle *, * \rangle$ 为内积。

即核函数输入两个向量，它返回的值跟两个向量分别作 Φ 映射然后点积的结果相同。

核技巧是一种利用核函数直接计算 $\langle \Phi(x), \Phi(z) \rangle$ ，以避开分别计算 $\Phi(x)$ 和 $\Phi(z)$ ，从而加速核方法计算的技巧。

不知道 Φ 的情况下，如何判断某个函数 K 是不是核函数？答案是 K 是核函数当且仅当对任意数据 $D = x_1, x_2, \dots, x_m$ ，核矩阵 (kernel matrix, gram matrix) 总是半正定的：

$$KernelMatirx = \begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & \dots & K(x_1, x_m) \\ K(x_2, x_1) & K(x_2, x_2) & \dots & K(x_2, x_m) \\ \dots & \dots & \dots & \dots \\ K(x_m, x_1) & K(x_m, x_2) & \dots & K(x_m, x_m) \end{bmatrix} \quad (13)$$

常用核函数如下：

名称	表达式	参数
线性核	$K(x_i, x_j) = x_i^T x_j$	
多项式核	$K(x_i, x_j) = (x_i^T x_j)^d$	$d \geq 1$
高斯核	$K(x_i, x_j) = \exp\left(-\frac{\ x_i - x_j\ ^2}{2\sigma^2}\right)$	$\sigma > 0$

名称	表达式	参数
拉普拉斯核	$K(x_i, x_j) = \exp\left(-\frac{\ x_i - x_j\ }{2\sigma^2}\right)$	$\sigma > 0$
Sigmoid核	$\tanh(\beta x_i^T x_j + \theta)$	$\beta > 0, \theta > 0$

KNN核函数优化

KNN算法结合核函数优化的方法很简单，设高维空间 χ 里两个点 x_i, x_j ，对于核函数 $K(x, y)$ ，核化以后欧式距离的公式为：

$$L_2(x_i, x_j) = \|\Phi(x_i) - \Phi(x_j)\|^2 = K(x_i, x_i) - 2K(x_i, x_j) + K(x_j, x_j) \quad (14)$$

经过一次变换后，把 $\Phi(x_i)$ 和 $\Phi(x_j)$ 消除掉了，完全使用关于 x_i, x_j 的核函数来表示距离，并不需要直接将 x_i, x_j 变换到高维空间才求距离，而是直接用核函数计算出来。

Kai Yu 在《Kernel Nearest-Neighbor Algorithm》中论证过基于核方法的KNN分类器比传统KNN分类器表现的好，因为仅仅是距离测量方式改变了一下，所以总体时间和传统KNN分类器依然类似，但是效果好了很多：

	Set1	Set2	Set3	Set4	Set 5	Ave.	Std.
1-nn (%)	72	54	58	68	72	64.3	6.74
Kernel 1-nn (%)	90	86	92	94	90	87.1	4.67

在不同的数据集上，核化KNN都能比传统KNN表现的更精确和稳定，他们使用US Postal Service数据和BUPA Liver Disorder数据进行了验证，结果表明核化过的KNN分类器精度明显好于传统的KNN，和SVM有得一拼：

Table II. Correct Classification Rates (%) of BUPA Liver Disorders

1-nn (%)	Kernel 1-nn (%)	Edited 1-nn (%)	Kernel Edited 1-nn (%)	3-nn (%)	Kernel 3-nn (%)	SVM (%)
60	63	64	64	65	71	68

同样，Shehroz khan等人在《kernels for One-Class Nearest Neighbour Classification》验证了核化KNN在One-Class分类问题上取得了比SVM One-Class更优秀的识别能力，在数个数据集上达到了87%–95%的准确率。

系统实现流程

系统环境



OpenCV环境配置

Required Packages

- GCC 4.4.x or later
- CMake 2.8.7 or higher
- Git
- GTK+2.x or higher, including headers(libgtk2.0-dev)
- pkg-config
- Python 2.6 or later and Numpy 1.5 or later with developer packages (python-dev, python-numpy)
- ffmpeg or libav development packages: libavcodec-dev, libavformat-dev, libswscale-dev

Getting OpenCV Source Code

1. 从官网上下载源码：[OpenCV-Download](#)
2. 从Github上获取源码：

```
git clone https://github.com/opencv/opencv.git
```

Building OpenCV from Source Using CMake

```
cd opencv
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=Release -D
CMAKE_INSTALL_PREFIX=/usr/local ..
make -j7 # runs 7 jobs in parallel
sudo make install # install libs
```

Test

编写一个CPP文件test_opencv.cpp测试OpenCV是否安装成功

```
cd opencv
mkdir test
cd test
vim test_opencv.cpp
```

test_opencv.cpp文件

```
#include<iostream>
#include<opencv2/opencv.hpp>
#include<opencv2/imgproc/imgproc.hpp>
using namespace cv;

int main(int argc,const char *argv[])
{
    Mat image;
    VideoCapture capture(0);
    while(1){
        capture>>image;
        imshow("test",image);
        waitKey(20);
    }
    return 0;
}
```

编写CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)
project(test_opencv)
find_package(OpenCV REQUIRED)
add_executable(test_opencv test_opencv.cpp)
target_link_libraries(test_opencv ${OpenCV_LIBS})
```

编译

```
cmake .
make
```

运行

```
./test_opencv
```

运行结果如下：



中科视拓开源框架SeetaFace2编译使用

编译

```
cd SeetaFace2
mkdir build # 创建build文件夹
cd build
cmake .. -G"Unix Makefiles" -DCMAKE_INSTALL_PREFIX=`pwd`/install
-DCMAKE_BUILD_TYPE=Release -DBUILD_EXAMPLE=ON
cmake --build . --config Release
cmake --build . --config Release --target install/strip # 安装
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`/bin # 把生成库的目录
加入到变量LD_LIBRARY_PATH中
cd bin
mkdir model
cp fd_2_00.dat pd_2_00_pts5.dat pd_2_00_pts81.dat . # 拷贝模型文件
到程序执行目录的model目录下
```

因为每次修改代码后都得重新编译一遍，以上编译步骤太多，因此我写了一个shell脚本compile.sh，放在seetaFace2/build/bin/目录下，一次性执行以上步骤，简化编译过程。

```
#script for compile seetaFace2 example
cd ..
cmake .. -G"Unix Makefiles" -DCMAKE_INSTALL_PREFIX=`pwd`/install
-DCMAKE_BUILD_TYPE=Release -DBUILD_EXAMPLE=ON
cmake --build . --config Release
cmake --build . --config Release --target install/strip
cd bin/
```

每次修改代码后cd进入SeetaFace2/build/bin/目录，执行：

```
./compile.sh
```

编译自己的代码

在理解CMake编译原理的基础上可以很容易地添加自己的代码并编译。

```
cd SeetaFace2/example
mkdir FaceRecognition
cd FaceRecognition
vim FaceRecognition.cpp # 自己的代码
vim CMakeLists.txt
```

其中CMakeListst.txt内容如下：

```
#
# Author: hustccc<1276675421@qq.com>
# Date  : 2020/6/10
#
cmake_minimum_required(VERSION 2.8)

project(FaceRecognition)

if(UNIX)
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fPIC")
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -fPIC")
endif()

find_package(OpenCV REQUIRED)

# add library
# add_executable(${PROJECT_NAME} example.cpp)
add_executable(${PROJECT_NAME} KNNFaceRecognizer.cpp)
target_include_directories(${PROJECT_NAME} PRIVATE
${CMAKE_CURRENT_SOURCE_DIR})
```

```

${CMAKE_SOURCE_DIR}/SeetaNet/include
${CMAKE_SOURCE_DIR}/FaceDetector/include
${CMAKE_SOURCE_DIR}/FaceLandmarker/include
${CMAKE_SOURCE_DIR}/FaceRecognizer/include
${CMAKE_SOURCE_DIR}/QualityAssessor/include
${CMAKE_BINARY_DIR}
)
target_include_directories(${PROJECT_NAME} PRIVATE
${OpenCV_INCLUDE_DIRS}
)

target_link_libraries(${PROJECT_NAME} PRIVATE
SeetaNet SeetaFaceDetector SeetaFaceLandmarker
SeetaFaceRecognizer SeetaQualityAssessor)
target_link_libraries(${PROJECT_NAME} PRIVATE ${OpenCV_LIBS})

set_target_properties(${PROJECT_NAME} PROPERTIES
RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
set_target_properties(${PROJECT_NAME} PROPERTIES VERSION
${BUILD_VERSION})
if(UNIX AND NOT ANDROID)
    set_target_properties(${PROJECT_NAME} PROPERTIES
RUNTIME_OUTPUT_NAME ${PROJECT_NAME})
endif()
INSTALL(TARGETS ${PROJECT_NAME}
        RUNTIME DESTINATION "${CMAKE_INSTALL_BINDIR}"
        LIBRARY DESTINATION "${CMAKE_INSTALL_LIBDIR}"
    )

# Find user and system name
# SET(SYSTEM_NAME $ENV{USERDOMAIN} CACHE STRING SystemName)
# SET(USER_NAME $ENV{USERNAME} CACHE STRING UserName)

if (MSVC_IDE)
    # Configure the template file
    SET(_CMAKE_LocalDebuggerEnvironment
"PATH=${CMAKE_BINARY_DIR}/bin/${Configuration};${CMAKE_BINARY_DIR}/lib/${Configuration};${OpenCV_LIB_PATH}/../bin;%PATH%")
    SET(USER_FILE ${PROJECT_NAME}.vcxproj.user)
    SET(OUTPUT_PATH ${CMAKE_CURRENT_BINARY_DIR}/${USER_FILE})
    CONFIGURE_FILE(${CMAKE_CURRENT_SOURCE_DIR}/UserTemplate.xml
${USER_FILE} @ONLY)

    UNSET(USER_FILE)
    UNSET(OUTPUT_PATH)
    UNSET(_CMAKE_LocalDebuggerEnvironment)

```

```
endif()
```

修改SeetaFace2/example目录下的CMakeLists.txt文件，添加一行：

```
echo "add_subdirectory(FaceRecognition)" >> CMakeLists.txt
```

这样每次运行SeetaFace2/build/bin/目录下的compile.sh脚本的时候就会在bin目录下编译生成FaceRecognition可执行文件和对应的动态链接库。

人脸检测 FaceDetector

包含头文件：

```
#include<seeta/FaceDetector.h>
#include<seeta/Struct_cv.h>
```

人脸检测：

```
seeta::FaceDetector
FD(seeta::ModelSetting("model/fd_2_00.dat"));
seeta::cv::ImageData image = cv::imread("i.png");
SeetaFaceInfoArray faces = FD.detect(image);
```

图像格式：

```
struct SeetaImageData
{
    int width;
    int height;
    int channels;
    unsigned char *data;
};
```

检测结果：

```
struct SeetaFaceInfo
{
    SeetaRect pos;
    float score;
};

struct SeetaFaceInfoArray
{
    struct SeetaFaceInfo *data;
    int size;
};
```

人脸关键点定位 FaceLandmarker

包含头文件

```
#include<seeta/FaceLandmarker.h>
#include<seeta/Struct_v.h>
```

人脸关键点定位

```
sseeta::FaceLandmarker
FL(seeta::ModelSetting("model/pd_2_00_pts5.dat"));
seeta::cv::ImageData image = cv::imread("1.png");
std::vector<SeetaPointF> points =
FL.mark(image, faces.data[0].pos);
```

检测结果

```
struct SeetaFaceF
{
    double x;
    double y;
};
```

人脸特征值提取 Extract

包含头文件

```
#include<seeta/FaceRecognizer.h>
#include<seeta/Struct_v.h>
```

特征提取

```

seeta::FaceRecognizer
FR(seeta::ModelSetting("model/fr_2_10.dat"));
std::shared_ptr<float> features(new
float[FR.GetExtractFeatureSize()], std::default_delete<float[]>
());
FR.Extract(image, points, features.get());
# features是一个大小为1028的向量

```

KNN算法实现

KNN传入参数：

1. 训练集图片特征值张量
2. 训练集图片标签
3. 测试集图片张量
4. 特征向量大小
5. K值

KNN返回参数：测试集图片预测标签

```

vector<string> KNNClassifier(vector<float *> train_data,
vector<string> train_label, vector<float *> test_data, int
feature_length, int k)
{
    printf("Run KNN Classifier..\n");
    int i, j, m;
    int predictLabelIndex;
    float a = Parameter_A;
    float c = Parameter_C;
    int d = Parameter_D;
    float gama = Parameter_gama;
    float r = Parameter_R;
    vector<double> LabelCount;
    for (i = 0; i < k; i++)
        LabelCount.push_back(0);
    vector<int> kMinIndex;
    //feature*MulRate
    for (i = 0; i < train_data.size(); i++)
        for (j = 0; j < feature_length; j++)
            *(train_data.at(i) + j) *= MulRate;

    for (i = 0; i < test_data.size(); i++)
        for (j = 0; j < feature_length; j++)
            *(test_data.at(i) + j) *= MulRate;

    vector<string> predict_result;

```

```

vector<vector<double>> DistList;
vector<double> zerodist;
//init DistList
for (i = 0; i < train_data.size(); i++)
    zerodist.push_back(0);
for (i = 0; i < test_data.size(); i++)
    DistList.push_back(zerodist);

for (i = 0; i < test_data.size(); i++)
    for (j = 0; j < train_data.size(); j++)
        //DistList.at(i).at(j) =
EuclideanDist(test_data.at(i), train_data.at(j),
feature_length);
        //DistList.at(i).at(j) =
ManhanttenDist(test_data.at(i), train_data.at(j),
feature_length);
        DistList.at(i).at(j) = KernelDist_1(test_data.at(i),
train_data.at(j), c, feature_length);
        //DistList.at(i).at(j) =
KernelDist_2(test_data.at(i), train_data.at(j), a,c,d,
feature_length);
        //DistList.at(i).at(j) =
KernelDist_3(test_data.at(i), train_data.at(j), gama,
feature_length);
        //DistList.at(i).at(j) =
KernelDist_4(test_data.at(i), train_data.at(j), gama, r,
feature_length);
for (i = 0; i < DistList.size(); i++)
{
    kMinIndex = GetKMinIndex(DistList.at(i), k);
    //cout<<"KMinIndex size: "<<kMinIndex.size()<<endl;
    for (j = 0; j < k; j++)
        LabelCount.at(j) = 0;
    for (j = 0; j < k; j++)
        for (m = 0; m < k; m++)
            if (train_label.at(kMinIndex.at(j)) ==
train_label.at(kMinIndex.at(m)))
                //LabelCount.at(j)++;
                LabelCount.at(j) += 100 /
DistList.at(i).at(kMinIndex.at(m));
    predictLabelIndex =
kMinIndex.at(max_element(LabelCount.begin(), LabelCount.end()) -
LabelCount.begin()));

predict_result.push_back(train_label.at(predictLabelIndex));
}

```

```
    return predict_result;  
}
```

核函数实现

LinearKernel :

```
double LinearKernel(float *x, float *y, float c, int length)  
{  
    int i = 0;  
    double temp = 0;  
    for (i = 0; i < length; i++)  
        temp += (*(x + i)) * (*(y + i));  
    return temp + c;  
}
```

PolynomialKernel :

```
double PolynomialKernel(float *x, float *y, float a, float c,  
int d, int length)  
{  
    int i = 0;  
    double temp = 0;  
    for (i = 0; i < length; i++)  
        temp += (*(x + i)) * (*(y + i));  
    temp = pow(temp * a + c, d);  
    return temp;  
}
```

RadialBasisKernel :

```
double RadialBasisKernel(float *x, float *y, float gama, int  
length)  
{  
    int i = 0;  
    double temp = 0;  
    for (i = 0; i < length; i++)  
        temp += pow(*(x + i) - *(y + i), 2);  
    temp = temp * gama * (-1);  
    //temp = temp * gama ;  
    temp = exp(temp);  
    return temp;  
}
```

SigmoidKernel :

```
double SigmoidKernel(float *x, float *y, float gama, float r,
int length)
{
    int i = 0;
    double temp;
    for (i = 0; i < length; i++)
        temp += (*(x + i)) * (*(y + i));
    temp *= gama;
    temp += r;
    temp = tanh(temp);
    return temp;
}
```

KNN结合核函数优化

修改KNN算法中的距离，使用核函数进行计算：

```
double KernelDist_1(float *x, float *y, float c, int length)
{
    return LinearKernel(x, x, c, length) - LinearKernel(x, y, c,
length) * 2 + LinearKernel(y, y, c, length);
}

double KernelDist_2(float *x, float *y, float a, float c, int d,
int length)
{
    return PolynomialKernel(x, x, a, c, d, length) -
PolynomialKernel(x, y, a, c, d, length) * 2 +
PolynomialKernel(y, y, a, c, d, length);
}

double KernelDist_3(float *x, float *y, float gama, int length)
{
    return RadialBasisKernel(x, x, gama, length) -
RadialBasisKernel(x, y, gama, length) * 2 + RadialBasisKernel(y,
y, gama, length);
}

double KernelDist_4(float *x, float *y, float gama, float r, int
length)
{
    return SigmoidKernel(x, x, gama, r, length) -
SigmoidKernel(x, y, gama, r, length) * 2 + SigmoidKernel(y, y,
gama, r, length);
```

使用LFW数据集

LFW数据集介绍

Labeled Faces in the Wild Home (LFW)

More than 13,000 images of faces collected from the web. Each face has been labeled with the name of the person pictured. 1680 of the people pictured have two or more distinct photos in the data set.

LFW数据集是为了研究非限制环境下的人脸识别问题而建立的。这个数据集包含超过13, 000张人脸图像，均采集于Internet。

每个人脸均被标准了一个人名。其中，大约1680个人包含两个以上的人脸。

这个集合被广泛应用于评价Face Verification算法的性能。

去官网下载数据集压缩包：[Labeled Faces in the Wild Home](#)

LFW数据集目录如下：

```
    ├── Zhu_Rongji_0006.jpg  
    ├── Zhu_Rongji_0007.jpg  
    ├── Zhu_Rongji_0008.jpg  
    └── Zhu_Rongji_0009.jpg  
- Zico  
  └── Zico_0001.jpg  
  └── Zico_0002.jpg  
  └── Zico_0003.jpg  
- Zinedine_Zidane  
  └── Zinedine_Zidane_0001.jpg  
  └── Zinedine_Zidane_0002.jpg  
  └── Zinedine_Zidane_0003.jpg  
  └── Zinedine_Zidane_0004.jpg  
  └── Zinedine_Zidane_0005.jpg  
  └── Zinedine_Zidane_0006.jpg  
- Ziwang_Xu  
  └── Ziwing_Xu_0001.jpg  
- Zoe_Ball  
  └── Zoe_Ball_0001.jpg  
- Zoran_Djindjic  
  └── Zoran_Djindjic_0001.jpg  
  └── Zoran_Djindjic_0002.jpg  
  └── Zoran_Djindjic_0003.jpg  
  └── Zoran_Djindjic_0004.jpg  
- zorica_Radovic  
  └── Zorica_Radovic_0001.jpg  
- Zulfiqar_Ahmed  
  └── Zulfiqar_Ahmed_0001.jpg  
- Zumrati_Juma  
  └── Zumrati_Juma_0001.jpg  
- Zurab_Tsereteli  
  └── Zurab_Tsereteli_0001.jpg  
- Zydrunas_Ilgauskas  
  └── Zydrunas_Ilgauskas_0001.jpg
```

5749 directories, 13234 files

```
[~/Pictures/FaceImage/LFW_Face_DataSet)——(hustccc@Manjaro:pts/5)  
[(05:16:24)→ [ | AN94 | AR15 | M4A1 ] —(火, 6月 23)]
```

数据集附带一个文件lfw-names.txt，其中包含着数据集所有人脸主人的名字，和对应的图片数目：

AJ_Cook 1
AJ_Lamas 1
Aaron_Eckhart 1
Aaron_Guiel 1
Aaron_Patterson 1
Aaron_Peirsol 4
Aaron_Pena 1
Aaron_Sorkin 2
Aaron_Tippin 1
Abba_Eban 1
Abbas_Kiarostami 1
Abdel_Aziz_Al-Hakim 1
Abdel_Madi_Shabneh 1
Abdel_Nasser_Assidi 2
Abdoulaye_Wade 4
Abdul_Majeed_Shobokshi 1
Abdul_Rahman 1
Abdulaziz_Kamilov 1
Abdullah 4
Abdullah_Ahmad_Badawi 1
Abdullah_Gul 19
Abdullah_Nasseef 1
Abdullah_al-Attiyah 3
Abdullatif_Sener 2
Abel_Aguilar 1
Abel_Pacheco 4
Abid_Hamid_Mahmud_Al-Tikriti 3
Abner_Martinez 1
Abraham_Foxman 1
Aby_Har-Even 1
Adam_Ant 1
Adam_Freier 1
Adam_Herbert 1
Adam_Kennedy 1
Adam_Mair 1
Adam_Rich 1
Adam_Sandler 4

"lfw-names.txt.bk" 5749L, 94727C
| AN94 | AR15 | M4A1

1,1

Top

lfw数据集处理

首先编写一个python文件getImgMoreThanOne.py将所有包含两张以上图片的人名（为了划分训练集和测试集）写入到一个文件new-lfw-names里。
getImgMoreThanOne.py文件如下：

```
# py files for get names of more than one img in lfw face data
set
import os

f=open("lfw-names.txt.bk")
newtxt="new-lfw-names"
```

```
newf=open(newtxt, "a+")

lines=f.readlines()
print(len(lines))
num=1
newNum=0
for line in lines:
    array=line.split()
    if(int(array[1])>1):
        #new_context=array[0]+'\n'+array[1]+'\n'
        new_context=array[0]+'\n'
        newf.writelines(new_context)
        newNum+=1
    num+=1
    if(num%1000==0):print("%d / %d"%(num,len(lines)))
print("new lines: %d"%(newNum))

f.close()
newf.close()
```

然后得到new-lfw-names文件：

```
Aaron_Peirsol
Aaron_Sorkin
Abdel_Nasser_Assidi
Abdoulaye_Wade
Abdullah
Abdullah_Gul
Abdullah_al-Attiyah
Abdullatif_Sener
Abel_Pacheco
Abid_Hamid_Mahmud_Al-Tikriti
Adam_Sandler
Adam_Scott
Adel_Al-Jubeir
Adolfo_Aguilar_Zinser
Adolfo_Rodriguez_Saa
Adrian_McPherson
Adrian_Nastase
Adrien_Brody
Ahmad_Masood
Ahmed_Chalabi
Ahmet_Necdet_Sezer
Ai_Sugiyama
Aicha_El_Ouafi
Aitor_Gonzalez
Akbar_Hashemi_Rafsanjani
Akhmed_Zakayev
Al_Davis
Al_Gore
Al_Pacino
Al_Sharpton
Alan_Ball
Alan_Greenspan
Alan_Mulally
Alastair_Campbell
Albert_Costa
Alberto_Fujimori
Alberto_Ruiz_Gallardon
"new-lfw-names" 3360L, 49104C
```

1,1

Top

到了这里，将会用到一些Linux系统上文件批处理的小技巧。编写一个shell脚本getImgPath.sh，结合find,grep,head,echo,sed等命令，输出重定向和管道通信，还有shell数组和流程控制，将lfw数据集中所有图片的相对目录划分为训练集和测试集分别写入到文件TrainImgPath和TestImgPath中。getImgPath.sh脚本如下：

```
#!/bin/bash
# shell script for get img path for lfw face dataset
namefile=new-lfw-names
NameArray=()
index=0
```

```

index_inc=1
maxindex=1679
while(( $index <= $maxindex ))
do
    NameArray[$index]=`sed -n ''${index_inc}'p' $namefile`
    let "index++"
    let "index_inc++"
done
#index=0
#index_inc=1
#while(( $index <= $maxindex ))
#do
#    echo ${NameArray[$index]}
#    let "index++"
#    let "index_inc++"
#done
cd lfw
index=0
index_inc=1
echo "" > ../TrainImgPath
echo "" > ../TestImgPath
while(( $index <= $maxindex ))
do
    tempName="${NameArray[$index]}*"
    find $tempName | grep jpg | head -n -1 >> ../TrainImgPath
    find $tempName | grep jpg | head -n 1 >> ../TestImgPath
    echo "#" >> ../TrainImgPath
    #echo "#" >> ../TestImgPath
    find $tempName | grep jpg
    let "index++"
    let "index_inc++"
done
cd ..

```

运行

```

sudo chmod +x getImgPath.sh
./getImgPath.sh

```


TrainImgPath:

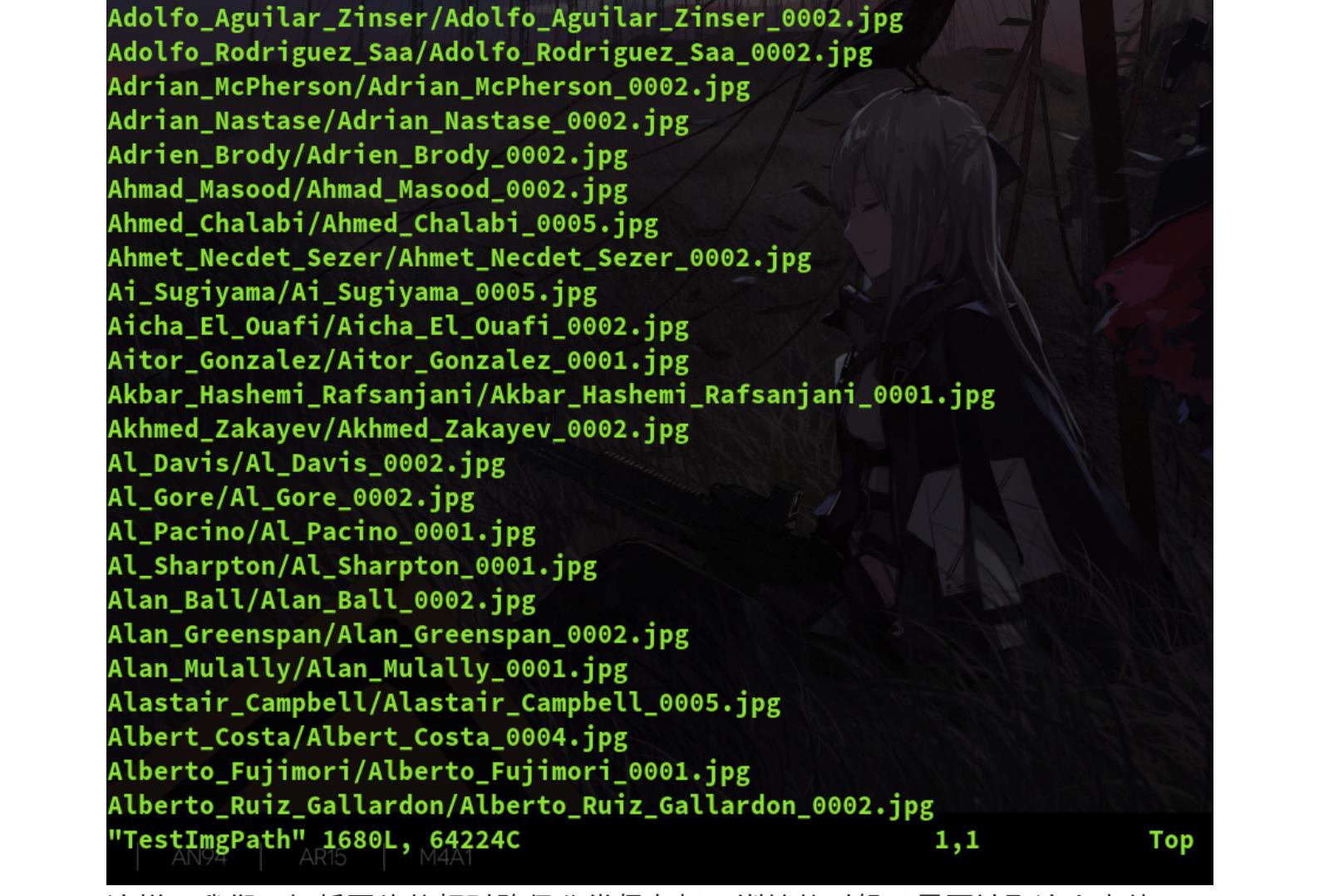
```
Aaron_Peirsol/Aaron_Peirsol_0004.jpg
Aaron_Peirsol/Aaron_Peirsol_0002.jpg
Aaron_Peirsol/Aaron_Peirsol_0003.jpg
#
Aaron_Sorkin/Aaron_Sorkin_0001.jpg
#
Abdel_Nasser_Assidi/Abdel_Nasser_Assidi_0002.jpg
#
Abdoulaye_Wade/Abdoulaye_Wade_0003.jpg
Abdoulaye_Wade/Abdoulaye_Wade_0004.jpg
Abdoulaye_Wade/Abdoulaye_Wade_0001.jpg
#
Abdullah/Abdullah_0001.jpg
Abdullah/Abdullah_0004.jpg
Abdullah/Abdullah_0002.jpg
Abdullah/Abdullah_0003.jpg
Abdullah_Ahmad_Badawi/Abdullah_Ahmad_Badawi_0001.jpg
Abdullah_al-Attiyah/Abdullah_al-Attiyah_0003.jpg
Abdullah_al-Attiyah/Abdullah_al-Attiyah_0002.jpg
Abdullah_al-Attiyah/Abdullah_al-Attiyah_0001.jpg
Abdullah_Gul/Abdullah_Gul_0016.jpg
Abdullah_Gul/Abdullah_Gul_0018.jpg
Abdullah_Gul/Abdullah_Gul_0012.jpg
Abdullah_Gul/Abdullah_Gul_0013.jpg
Abdullah_Gul/Abdullah_Gul_0008.jpg
Abdullah_Gul/Abdullah_Gul_0004.jpg
Abdullah_Gul/Abdullah_Gul_0001.jpg
Abdullah_Gul/Abdullah_Gul_0007.jpg
Abdullah_Gul/Abdullah_Gul_0009.jpg
Abdullah_Gul/Abdullah_Gul_0011.jpg
Abdullah_Gul/Abdullah_Gul_0010.jpg
Abdullah_Gul/Abdullah_Gul_0005.jpg
Abdullah_Gul/Abdullah_Gul_0015.jpg
Abdullah_Gul/Abdullah_Gul_0002.jpg
Abdullah_Gul/Abdullah_Gul_0003.jpg
Abdullah_Gul/Abdullah_Gul_0006.jpg
Abdullah_Gul/Abdullah_Gul_0019.jpg
"TrainImgPath" 9200L, 290976C
```

1,1

Top

TestImgPath:

```
Aaron_Peirsol/Aaron_Peirsol_0004.jpg
Aaron_Sorkin/Aaron_Sorkin_0001.jpg
Abdel_Nasser_Assidi/Abdel_Nasser_Assidi_0002.jpg
Abdoulaye_Wade/Abdoulaye_Wade_0003.jpg
Abdullah/Abdullah_0001.jpg
Abdullah_Gul/Abdullah_Gul_0016.jpg
Abdullah_al-Attiyah/Abdullah_al-Attiyah_0003.jpg
Abdullatif_Sener/Abdullatif_Sener_0001.jpg
Abel_Pacheco/Abel_Pacheco_0002.jpg
Abid_Hamid_Mahmud_Al-Tikriti/Abid_Hamid_Mahmud_Al-Tikriti_0003.jpg
Adam_Sandler/Adam_Sandler_0004.jpg
Adam_Scott/Adam_Scott_0001.jpg
Adel_Al-Jubeir/Adel_Al-Jubeir_0002.jpg
```



```
Adolfo_Aguilar_Zinser/Adolfo_Aguilar_Zinser_0002.jpg
Adolfo_Rodriguez_Saa/Adolfo_Rodriguez_Saa_0002.jpg
Adrian_McPherson/Adrian_McPherson_0002.jpg
Adrian_Nastase/Adrian_Nastase_0002.jpg
Adrien_Brody/Adrien_Brody_0002.jpg
Ahmad_Masood/Ahmad_Masood_0002.jpg
Ahmed_Chalabi/Ahmed_Chalabi_0005.jpg
Ahmet_Necdet_Sezer/Ahmet_Necdet_Sezer_0002.jpg
Ai_Sugiyama/Ai_Sugiyama_0005.jpg
Aicha_El_Ouafi/Aicha_El_Ouafi_0002.jpg
Aitor_Gonzalez/Aitor_Gonzalez_0001.jpg
Akbar_Hashemi_Rafsanjani/Akbar_Hashemi_Rafsanjani_0001.jpg
Akhmed_Zakayev/Akhmed_Zakayev_0002.jpg
Al_Davis/Al_Davis_0002.jpg
Al_Gore/Al_Gore_0002.jpg
Al_Pacino/Al_Pacino_0001.jpg
Al_Sharpton/Al_Sharpton_0001.jpg
Alan_Ball/Alan_Ball_0002.jpg
Alan_Greenspan/Alan_Greenspan_0002.jpg
Alan_Mulally/Alan_Mulally_0001.jpg
Alastair_Campbell/Alastair_Campbell_0005.jpg
Albert_Costa/Albert_Costa_0004.jpg
Alberto_Fujimori/Alberto_Fujimori_0001.jpg
Alberto_Ruiz_Gallardon/Alberto_Ruiz_Gallardon_0002.jpg
"TestImgPath" 1680L, 64224C
```

1,1

Top

这样，我们已经将图片的相对路径分类保存好，训练的时候只需要读取这个文件，然后取文件名的最后一个'/'开始数到第一个'_'之间的字符串作为标签就行了。

使用CelebA数据集

CelebA数据集介绍

Large-scale CelebFaces Attributes (CelebA) Dataset

CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 attribute annotations. The images in this dataset cover large pose variations and background clutter. CelebA has large diversities, large quantities, and rich annotations, including

10,177 number of identities,

202,599 number of face images, and

5 landmark locations, 40 binary attributes annotations per image.

这是由香港中文大学汤晓鸥教授实验室公布的大型人脸识别数据集。该数据集包含有200K张人脸图片，人脸属性有40多种，主要用于人脸属性的识别。

去官网下载数据集：[CelebA](#)

CelebA数据集目录如下：

```
└── 202566.jpg  
└── 202567.jpg  
└── 202568.jpg  
└── 202569.jpg  
└── 202570.jpg  
└── 202571.jpg  
└── 202572.jpg  
└── 202573.jpg  
└── 202574.jpg  
└── 202575.jpg  
└── 202576.jpg  
└── 202577.jpg  
└── 202578.jpg  
└── 202579.jpg  
└── 202580.jpg  
└── 202581.jpg  
└── 202582.jpg  
└── 202583.jpg  
└── 202584.jpg  
└── 202585.jpg  
└── 202586.jpg  
└── 202587.jpg  
└── 202588.jpg  
└── 202589.jpg  
└── 202590.jpg  
└── 202591.jpg  
└── 202592.jpg  
└── 202593.jpg  
└── 202594.jpg  
└── 202595.jpg  
└── 202596.jpg  
└── 202597.jpg  
└── 202598.jpg  
└── 202599.jpg
```

```
0 directories, 202599 files  
[~/Pictures/FaceImage/CelebA) └─(hustccc@Manjaro:pts/5)  
  (05:56:21)→ [AN94 | AR15 | M4A1]
```

数据集附带文件的其中一个，list_attr_celeba.txt，里面包含了所有人脸图片的路径和属性：

5_o_Clock_Shadow Arched_Eyebrows Attractive Bags_Under_Eyes Bald Bangs Big_Lips Big_Nose Black_Hair Blond_Hair Blurry Brown_Hair Bushy_Eyebrows C
hubby Double_Chin Eyeglasses Goatee Gray_Hair Heavy_Makeup High_Cheekbones Male Mouth_Slightly_Open Mustache Narrow_Eyes No_Beard Oval_Face Pale_Skin Pointy_Nose Receding_Hairline Rosy_Cheeks Sideburns Smiling Straight_Hair Wavy_Hair Wearing_Earrings Wearing_Hat Wearing_Lipstick Wearing_Necklace Wearing_Necktie Young

000001.jpg -1 1 1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1 1 -1 1 -1 -1 1 -1 -1 -1 1 1 -1 1 -1 -1 1 1 -1 1 -1 -1 1
000002.jpg -1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 1 -1 1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1
000003.jpg -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1
000004.jpg -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1
000005.jpg -1 1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1
000006.jpg -1 1 1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 1
000007.jpg 1 -1 1 1 1 -1 -1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
000008.jpg 1 1 -1 1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
000009.jpg -1 1 1 -1 -1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
000010.jpg -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
000011.jpg -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
000012.jpg -1 -1 1 1 -1 -1 -1 -1 1 -1 -1 -1 1 -1 -1 -1 -1 1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
000013.jpg -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1 -1 -1 1 1 -1 -1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1
"list_attr_celeba.txt" [dos] 202601L, 26721026C

1,1

Top

其中1表示这张人脸图片有着对应的属性， -1表示没有对应的属性。

CelebA数据集处理

由于CelebA数据集太大，笔记本电脑跑不完全部图片，因此这里编写一个shell脚本get_labels.sh来提取list_attr_celeba.txt中的某部分行来作为训练集和数据集。get_labels.sh脚本如下：

```
#!/bin/bash
# shell script for get train labels for celeba
if [ $# == 2 ]
then
    head -n $1 list_attr_celeba.txt > $2
elif [ $# == 3 ]
then
    tail -n $1 list_attr_celeba.txt | head -n $2 > $3
else
    echo "num of paramters don't match"
fi
```

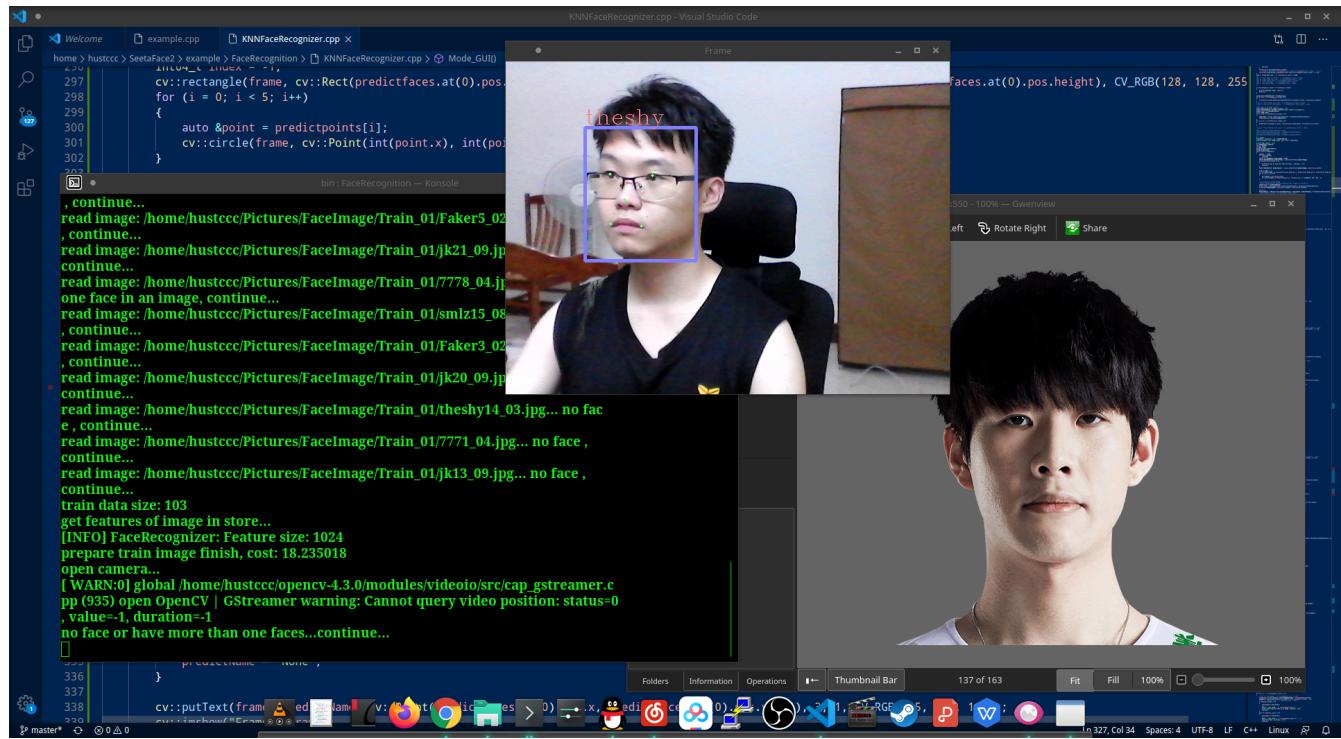
该脚本接收两个或三个参数，如果输入两个参数则第一个参数为从文件首部开始提取的行数，第二个参数为写入的文件名；如果是三个参数，则第一个参数为开始提取的行数序号，第二个参数是从第一个参数指定的行开始提取的行数，第三个参数是写入的文件名。

```
sudo chmod +x get_labels.sh  
.get_labels.sh 10000 celeba_train_labels  
.get_labels.sh 50000 5000 celeba_test_labels
```

提取完毕之后使用notepad文本编辑器打开，将无效行删除，将全部-1标签替换为0，并将全部空格去掉，得到两个用于训练celebA数据集的文本。训练的时候只需要读取这个文件，以文件每一行前面的图片名字作为相对路径，后面的01序列作为标签就行了。

摄像头模式

前面所提到的两个数据集用来测试本项目的模型性能和准确率，根据训练结果的反馈开发者可以不断完善系统。作为一个旨在推进人脸识别技术落地的项目来说，最关键的还是在应用层面的实现。本项目封装好了GUI模式，即摄像头模式，通过OpenCV提供的摄像头接口进行人脸数据采用，再通过训练好的模型与图片底库中注册的人脸图片进行比对，给出预测结果，并将预测结果输出到GTK窗口上。通过--GUI参数来调用摄像头模式。



自定义数据集

网上下载了一些LOL电竞明星和体育明星的人脸图片，加上自己和家人的人脸图片，建立来一个自定义的人脸数据集，用于模型测试。数据集目录如下：

```
0 directories, 162 files
└─(~/Pictures/FaceImage) ━━━━━━ hustccc@Manjaro:pts/5
  └─(06:53:01)→ tree Train_01
Train_01
├── 7771_04.jpg
├── 7772_04.jpg
├── 7773_04.jpg
├── 7774_04.jpg
├── 7775_4.png
├── 7777_04.jpg
├── 7778_04.jpg
├── chechunchi_01_00.jpg
├── chechunchi_03_00.jpg
├── chechunchi_04_00.jpg
├── chechunchi_05_00.jpg
├── chechunchi_06_00.jpg
├── chechunchi_07_00.jpg
├── chechunchi_08_00.jpg
├── chechunchi_09_00.jpg
├── chechunchi_10_00.jpg
├── cheqixian10_11.jpg
├── cheqixian11_11.jpg
├── cheqixian1_11.jpg
├── cheqixian2_11.jpg
├── cheqixian3_11.jpg
├── cheqixian4_11.jpg
├── cheqixian5_11.jpg
├── cheqixian6_11.jpg
├── cheqixian7_11.jpg
├── cheqixian8_11.jpg
├── cheqixian9_11.jpg
├── Faker10_02.jpg
├── Faker1_02.jpg
├── Faker11_02.jpg
├── Faker12_02.jpg
├── Faker13_02.jpg
├── Faker14_02.jpg
├── Faker15_02.jpg
└── Faker16_02.jpg
```

每张图片的名字最后的两位十进制数字作为标签。

系统测试

LFW数据集测试

首先设置训练参数为：训练集大小2000,测试集大小400,欧式距离，K值为3

编译代码：

```
-- == Build shared library: ON
-- == CMAKE_SYSTEM_PROCESSOR: x86_64
-- == CMAKE_BUILD_TYPE: Release
-- Configuring done
-- Generating done
-- Build files have been written to: /home/hustccc/SeetaFace2/build
Scanning dependencies of target SeetaNet
[ 1%] Linking CXX shared library ../bin/libSeetaNet.so
[ 25%] Built target SeetaNet
Scanning dependencies of target SeetaFaceDetector
[ 27%] Linking CXX shared library ../bin/libSeetaFaceDetector.so
[ 36%] Built target SeetaFaceDetector
Scanning dependencies of target SeetaFaceTracker
[ 38%] Linking CXX shared library ../bin/libSeetaFaceTracker.so
[ 40%] Built target SeetaFaceTracker
Scanning dependencies of target SeetaFaceLandmarker
[ 41%] Linking CXX shared library ../bin/libSeetaFaceLandmarker.so
[ 50%] Built target SeetaFaceLandmarker
Scanning dependencies of target SeetaFaceRecognizer
[ 52%] Linking CXX shared library ../bin/libSeetaFaceRecognizer.so
[ 65%] Built target SeetaFaceRecognizer
Scanning dependencies of target SeetaQualityAssessor
[ 67%] Linking CXX shared library ../bin/libSeetaQualityAssessor.so
[ 78%] Built target SeetaQualityAssessor
[ 80%] Linking CXX executable ../../bin/points81
[ 81%] Built target points81
[ 83%] Linking CXX executable ../../bin/search
[ 85%] Built target search
[ 87%] Linking CXX executable ../../bin/tracking
[ 89%] Built target tracking
[ 90%] Linking CXX executable ../../bin/crop_face
[ 92%] Built target crop_face
[ 94%] Linking CXX executable ../../bin/test
[ 96%] Built target test
Scanning dependencies of target FaceRecognition
[ 98%] Building CXX object example/FaceRecognition/CMakeFiles/FaceRecognition.dir/KNNFaceRecognizer.cpp.o
[100%] Linking CXX executable ../../bin/FaceRecognition
[100%] Built target FaceRecognition
```

运行结果：

```

not recognized: 260 chung
not recognized: 263 Cindy
not recognized: 283 Courtney
not recognized: 284 Cristina
not recognized: 285 Cristina
not recognized: 290 Damon
not recognized: 292 Dan
not recognized: 293 Daniel
not recognized: 319 David
not recognized: 324 Demetrius
not recognized: 332 Derek
not recognized: 343 Dino
not recognized: 345 Dolma
not recognized: 359 Doug
not recognized: 362 Duane
not recognized: 382 Elisabeth
not recognized: 386 Elizabeth
accuracy: 87.750000%
[] ~/SeetaFace2/build/bin <master> X

```

准确率为87.5%

下面将对不同参数进行训练，由于在实际应用中K值一般较小,因此这里取K值为1,2,3进行训练。

训练集大小1000,测试集大小200：

距离函数&K值	1	2	3
欧式距离	97	97	97
曼哈顿距离	96.5	96.5	96.5
核距离1	97	97	97
核距离2	97	97	97
核距离3	97	97	97
核距离4	97	97	97

训练集大小3000,测试集大小500：

距离函数&K值	1	2	3
欧式距离	95.2	95.2	95.2
曼哈顿距离	94.8	94.8	94.8
核距离1	95.2	95.2	95.2
核距离2	95.2	95.2	95.2

距离函数&K值	1	2	3
核距离3	95.2	95.2	95.2
核距离4	95.2	95.2	95.2

训练集大小5000,测试集大小1000 :

距离函数&K值	1	2	3
欧式距离	93.7	93.7	93.7
曼哈顿距离	93.7	93.7	93.7
核距离1	95.6	95.6	95.6
核距离2	95.6	95.6	95.6
核距离3	95.6	95.6	95.6
核距离4	95.6	95.6	95.6

CelebA数据集测试

首先设置训练参数为：训练集为前1000行,测试集为第20000行到第20200行,欧式距离, K值为3

运行结果：

```
test data size: 179
Test Image List size: 179
Test Faces List size: 179
Test Points List size: 179
the size of first points: 5
get features of test image ...
[INFO] FaceRecognizer: Feature size: 1024
Test Features List size: 179
prepare test image finish, cost: 42.727946
predict with knn...
Run KNN Classifier_x...
ok here
predict finish , cost: 0.503988
predict result:

PredictResult size: 179
num: 6292
accuracy: 87.877098%
[] ~/SeetaFace2/build/bin <master> X
```

准确率为87.87%

下面对不同参数进行训练：

训练集为前面2000行, 测试集为第10000行开始的200行：

距离函数&K值	1	2	3
欧式距离	86.6	86.6	86.9
曼哈顿距离	86.9	86.9	86.9
核距离1	86.6	86.6	86.6
核距离2	86.6	86.6	86.6
核距离3	86.6	86.6	86.6
核距离4	86.6	86.6	86.6

训练集为前面3000行，测试集为第20000行开始的500行：

距离函数&K值	1	2	3
欧式距离	78.5	78.5	78.5
曼哈顿距离	78.6	78.6	78.6
核距离1	78.5	78.5	78.5
核距离2	78.5	78.5	78.5
核距离3	78.5	78.5	78.5
核距离4	74.8	74.8	74.8

自定义数据集测试

训练集目录为/home/hustccc/Pictures/FaceImage/Train_01, 测试集目录为/home/hustccc/Pictures/FaceImage/Test_01

训练结果：

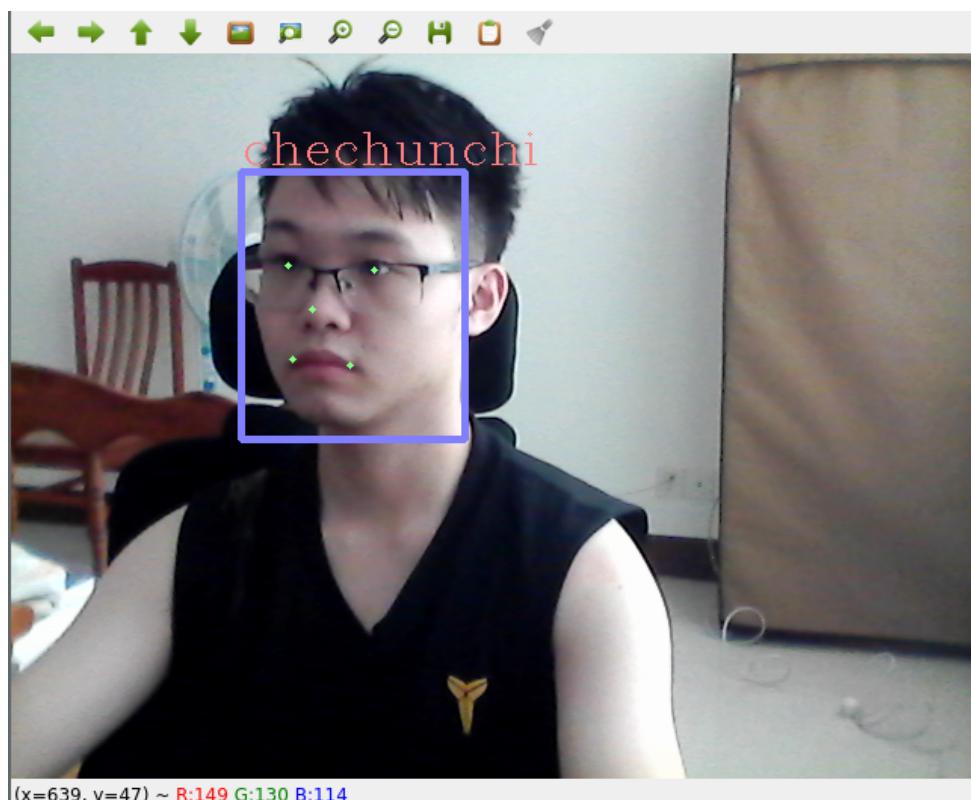
```
read image: /home/hustccc/Pictures/FaceImage/Test_01/Messi3_01.jpg... no face , continue...
test data size: 72
get features of test image...
prepare test image finish, cost: 12.873441
predict with knn...
Run KNN Classifier...
predict finish , cost: 0.014352
predict result:
[0] 1 [1] 7 [2] 3 [3] 6 [4] 8 [5] 9 [6] 8 [7] 0 [8] 9 [9] 3 [10] 6
[11] 1 [12] 2 [13] 2 [14] 7 [15] 9 [16] 1 [17] 4 [18] 4 [19] 9 [20] 6
[21] 9 [22] 1 [23] 7 [24] 1 [25] 7 [26] 0 [27] 3 [28] 6 [29] 1 [30] 5
[31] 1 [32] 3 [33] 4 [34] 1 [35] 8 [36] 9 [37] 9 [38] 6 [39] 0 [40] 3
[41] 8 [42] 0 [43] 0 [44] 1 [45] 7 [46] 7 [47] 5 [48] 2 [49] 7 [50] 7
[51] 0 [52] 2 [53] 2 [54] 3 [55] 1 [56] 4 [57] 3 [58] 9 [59] 1 [60] 1
[61] 0 [62] 4 [63] 1 [64] 3 [65] 2 [66] 9 [67] 2 [68] 0 [69] 4 [70] 9
[71] 6
accuracy: 100.000000%
main exit.
[] ~/SeetaFace2/build/bin <master> X
```

对自定义数据集的预测准确率为100%，应该是在网上找的明星人脸图片之间相差较大，系统比较容易区分。

摄像头模式（GUI模式）测试

PC机，Manjaro系统上摄像头模式测试：

我：



母亲：



测试结果分析

由上面测试结果可知，该模型对LFW数据集的拟合效果较好，对celebA的拟合效果就有点差强人意。毕竟celebA数据集的标签是40个属性，预测难度比较高。

另外对于k值为1,2,3来说训练结果是一样的，没什么区别。在实际应用中k值也是取的比较小的值。

引入核函数对上述训练过程的结果貌似没什么影响，可能是人脸数据集规模较小的原因。

对自定义数据集的测试准确率较高，可以看出该人脸识别系统对于清晰的图片预测效果还是挺好的。

摄像头模式基本上都能准确预测出人脸的主人，因此该系统用于家庭内部人脸识别是完全可以胜任的。

移植树莓派

这部分就不在报告里过多阐述，直接上效果图：



可以看出树莓派摄像头正常工作，完成图像捕捉的任务。

主要实现思路是将该项目上传到github，再在树莓派上用git下载本项目，由于项目是用CMake编译的，很容易实现跨平台编译。CMake的优势就这样体现出来了。在此之前得配置好OpenCV环境，同样是在官网上下载源码后使用CMake编译，编译时间较长，花费了10个小时左右的时间。

心得体会

这学期的机器学习课，由于疫情原因，期末考核方式改为了大作业形式，这对我来说是乐于接收的。我本身就比较喜欢实践，我觉得课堂上学到的理论知识如果没有在代码和项目上体现出来，就相当于学了一些空壳，只有将那些算法落地，才有实际的意义，而且也能加深对知识的理解。

另外相较于枯燥无味的复习，我还是更加享受做课设的过程，不仅能锻炼自己的代码能力，还能学到很多课外的东西。比如这次课设，我不仅学会了SeetaFace2这个开源人脸识别框架的使用，还学会了OpenCV的环境配置和使用，更者还深入了解了Linux系统下CMake这个便利的编译工具，这对于喜欢Linux的我来说是一笔丰厚的财富。

本科阶段打算向深度学习对抗攻防方向发展，以后还会与机器学习打交道的。这学

期因为疫情，不能与各位同学老师见面，感谢老师和助教这学期的教诲和陪伴，希望疫情过后，春暖花开，我们还能在校园内见面，感受大学校园带给我们的美好时光。

附录

因为源码有点长，所以就不贴源码了。这里放上github项目地址：

github

读者可以在github上下载项目，如果在编译方面有什么问题，可以在qq上联系我。

qq : 1276675421

也可以发邮件：1276675421@qq.com

参考文献：

[1]李航 (2012) 统计学习方法. 清华大学出版社, 北京.

[2]Kai Yu.Kernel Nearest-Neighbor Algorithm.

[3]维基百科.<https://zh.wikipedia.org/wiki/K-%E8%BF%91%E9%82%BB%E7%AE%97%E6%B3%95>.