Smile ratio calculator

- For automated lip reading -

Miniproject Report

Andreea Ciontos & Laura M. Fenoy

Aalborg University
Electronics and IT

Contents

Chapter 1

Introduction

1.1 Python as a programming language

Python is an interpreted , object oriented, high-level programming language with dynamic semantics. Meaning it runs through a program line by line and executes each command. Its semantics are similar to written English with some with influence from mathematics and a strong abstraction from computer language.

Among its applications there is: web development (server-side), software development, mathematics and system scripting. Python was designed for readability and has good compatibility across a variety of operating systems, whereas other languages are closer to machine language and therefore harder to understand.

1.2 Code optimisation

Code optimisation is a way of improving code quality and efficiency by making it consume fewer resources less memory, execute more rapidly, or perform fewer input/output operations. Some Language independent optimization techniques are:

Code Movement: The code is moved so that unnecessary code wont be executed more than needed.

Example:Code Movement Optimization	
Code Before Optimization	Code After Optimization

for (int i = 0 ; i < n ; i ++)
{
x = y + z ;
a[i] = x + 5i ;
}
	x = y + z ;
	for (int i = 0 ; i < n ; i ++)
	{
	a[i] = x + 5i ;
	}

Dead Code Elimination: Code which either never executes, is unreachable or their output is never used is eliminated.

Example: Dead Code Elimination	
Code Before Optimization	Code After Optimization

i = 0
if (i == 1)
{
a = x + 5 ;
}
	i = 0;

Strength Reduction: This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example: Strength Reduction	
Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Chapter 2

Analysis

2.1 Application

In the past decades, detection and tracking of facial features has been receiving a lot of attention. In this category falls also mouth detection. This is particularly important as the mouth gives important information about the overall facial expression of the subject. At mouth level it is possible to describe emotions or spoken messages, essential for various applications.

This project presents a way of detecting lips in a pre-recorded video file.The solution is implemented using the Python programming language and OpenCV [2]. Haar Cascades are used for feature extraction, with pre-trained models. Once the mouth is detected, a bounding box is drawn around the region of interest (ROI). The coordinates of the corners of this bounding box are saved into a .cvs file and then we perform calculations in three different ways. A graphic representation of the bounding box can be seen in Figure 2.1. Where the values are the x and y coordinates of the left upper corner of the rectangle and w and h which are the rectangle width and height respectively.

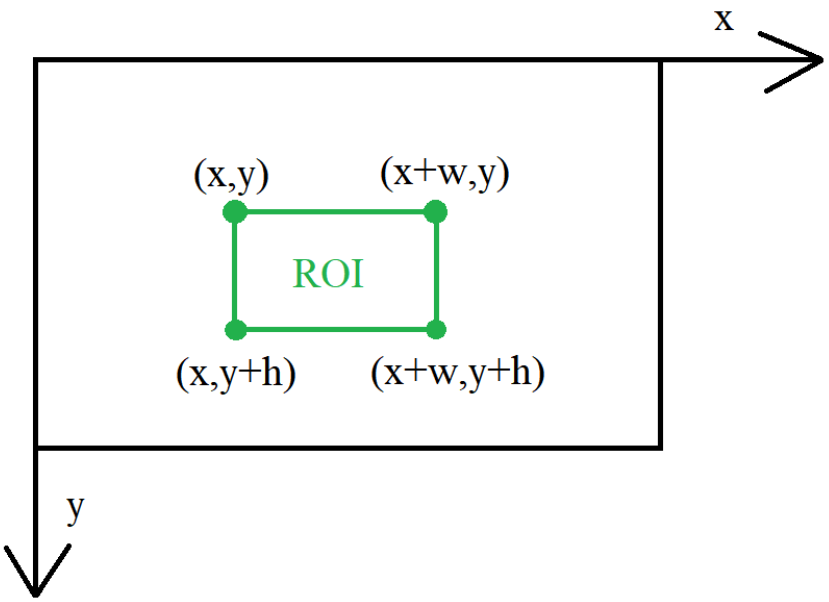

Figure 2.1: Example of bounding box coordinates.

Figure 2.2 showcases the algorithm running on a webcam video feed. There are two bounding boxes, a blue one for the face and a green one for the lips (smile). The face needs to be detected first for a more accurate detection of the lips.

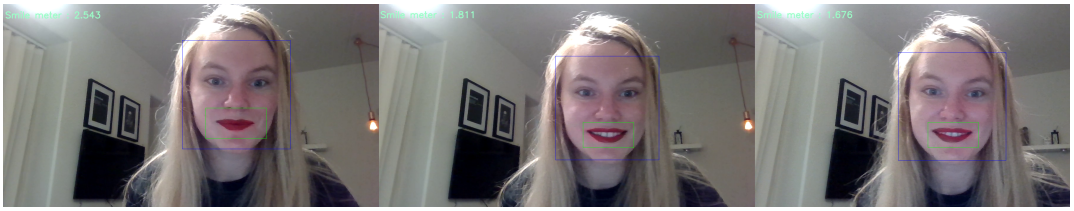

Figure 2.2: Example of bounding box.

Chapter 3

Implementation

3.1 Data acquisition

The data needed for this project is smile coordinates. For this we use openCV with python to detect faces in a video feed, which makes it easier to detect lips and get coordinates. The coordinates are saved in a file to be further operated on.

A comma separated values (csv) file is a text file that uses commas to separate the values that it contains. It is used to store values of a data base, as it is easy to access and manipulate from different programs. Figure 3.1 showcases the data structure of a csv file. The first row represents the headers of a table and the subsequent rows are the values stored for each header, forming columns.

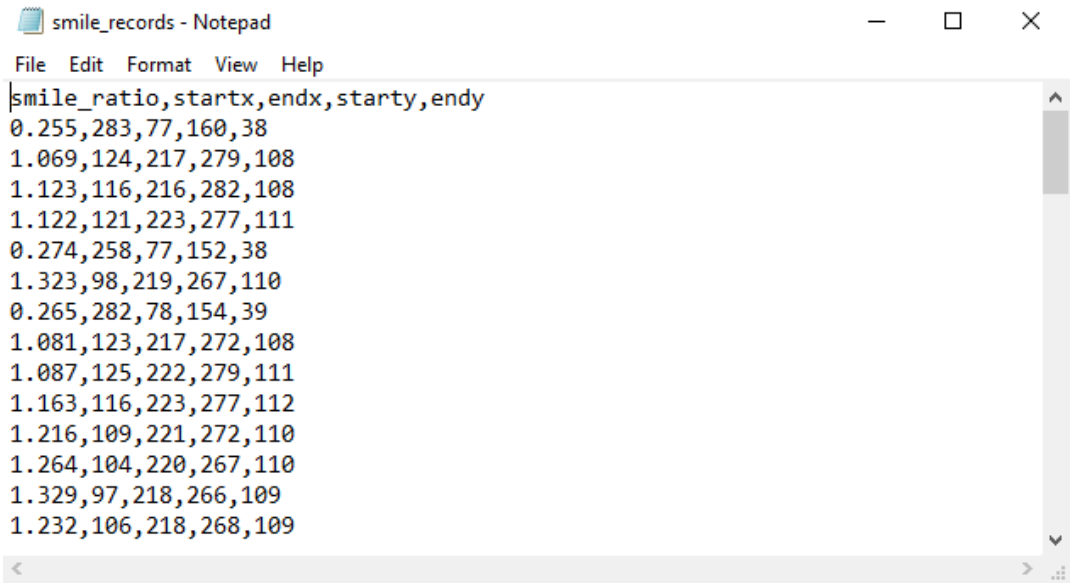

Figure 3.1: csv data structure

3.1.1 Pandas

Pandas are chosen for this project in order to be able to read and work with multi-dimensional data loaded from .csv files. Pandas is a python package used for data analysis and manipulation and it can be used in different forms, varying from traditional "crude" or naive version to vectorised form using numpy arrays. In data science it is important to be able to deal with a vast amount of data as fast as possible. The different ways Pandas can and will be used is explained in the following sections. [3]

3.2 Time Complexity

Time complexity is a property of a computational problem. It can be described as efficiency. And it is a theoretical measure of the time it takes for a function to process a given input. Not to be confused with execution time which is a property of an algorithm and describes the time it takes for a program to execute.

The time complexity of algorithms is most commonly expressed using the big O notation. The time complexity of an algorithm can be O(1), meaning it is constant. It can also be O(n), linear. Meaning its running time is directly proportional to n. Time complexity can be quadratic O(n^2) when the running time is directly proportional to the square of n. It can also be logarithmic O(log(n)) when it behaves in an exponential way or O(nlog(N)) if the algorithm is a combination of linear and logarithmic.

For the naive implementation, a for loop is implemented and therefore the time complexity is expected to be O(n), since the execution time of will increment / decrement by a constant amount depending on the size of the loop.

Whereas for the vectorised version the time complexity is expected to be O(1). Matrix operations are all executed simultaneously.

And lastly, the time complexity for the multiprocessing version is expected to be O(logn), since the task is divided among processors.

3.3 Naive version

The naive version of the code is just a simple implementation which allows to carry out the task using a `for loop`. In this implementation, the code is executed over multiple iterations in order to calculate the outcome. This method can be time consuming since it operates element by element.

```
import pandas as pd
from pandas import DataFrame
import time
import multiprocessing as mp
from dask import delayed as delay
import dask
import dask.dataframe as dd
import cProfile
import timeit


# read as pandas dataframe
# use only for naive and vectorized version
df = pd.read_csv(r"C:\Users\Bruger\Desktop\P8\NSCmp\smile_records.csv")

# NAIVE VERSION
startn = time.time()

# create array to store values
smile_ratios = []

# go through the arrays and compute element by element
for i in range(0, len(df)):
    smile_ratio = ((df.iloc[i]['endx']/df.iloc[i]['startx']) +
                   (df.iloc[i]['endy']/df.iloc[i]['starty']))/2
    smile_ratios.append(smile_ratio)

    startx = df.startx
    endx = df.endx
```

```
30      starty = df.starty
31      endy = df.endy
32
33      startx.append(startx)
34      endx.append(endx)
35      starty.append(starty)
36      endy.append(endy)
```

In the code above, `startx`, `endx`, `starty`, `endy` are the bounding box coordinates surrounding the lips. *smile_ratio* is running though all the elements of each list corresponding to the coordinates columns and calculates the smile ratios which is then stored in a list *smile_ratios*.

Time Sum = C_1 + C_2n

Time complexity = O(C_1 + C_2n) $\approx$ O(n)

3.4 Vectorised version

Vectorization is the process of executing operations on entire arrays. In the vectorised version, instead of doing calculations element-wise, it handles the task as matrix operations.Pandas include a number of built-in functions that facilitate array operations by vectorisation. These functions inherit fundamental units that operate on the arrays from the .csv file as a whole, instead of going through them element by element (scalars).

Numpy is considered Python's most important scientific computing library. Numpy operates similarly as Pandas on arrays, however, it skips some Pandas specific operations such as indexing or data type checking, thus it becomes faster than its counterpart. It is important to mention that the execution speed can be increased quite significantly by using Numpy instead of Pandas, anyhow, when indexing data is crucial, Pandas is prefered. For this project, indexing is important because the data comes in sequentially, from a video feed, frame by frame and the smile ratios must be also calculated frame by frame for the corresponding coordinates.

```
1
2  import pandas as pd
3  from pandas import DataFrame
4  import time
5  import multiprocessing as mp
6  from dask import delayed as delay
7  import dask
8  import dask.dataframe as dd
9  import cProfile
10 import timeit
11
12
```

```
# read as pandas dataframe
# use only for naive and vectorized version
df = pd.read_csv(r"C:\Users\Bruger\Desktop\P8\NSCmp\smile_records.csv")

# VECTORISED VERSION
startv = time.time()

# define computation in vector form
df['vectorized'] = ((df['endx']/df['startx'])+(df['endy']/df['starty']))/2

endv = time.time()
print (endv-startv)
```

The code above does the same computations as the one presented in the naive version, the difference being that now the computations are done with all the elements of each list at once. This speeds up the code considerably.

Time Sum = C_1

Time complexity = $O(C_1) \approx O(1)$

3.5 Multi-procesing version

Multi-processing refers to the ability of a system to support more than one processor at the same time. Applications in a multi-processing system are broken to smaller routines that run independently. Since the processes don't share memory, they can't modify the same memory concurrently, avoiding data corruption and deadlocks.

For this implementation `DASK` is used. `Dask` is a python library used to parallelize processes for a faster performance. Mainly used for machine learning applications, `Dask` has been chosen for this project for exploration and learning while splitting up the data frame in multiple chunks for parallel computing.[1]

```
import pandas as pd
from pandas import DataFrame
import time
import multiprocessing as mp
from dask import delayed as delay
import dask
import dask.dataframe as dd
import cProfile
import timeit


# read as pandas dataframe
# use only for naive and vectorized version
df = pd.read_csv(r"C:\Users\Bruger\Desktop\P8\NSCmp\smile_records.csv")

```

```
# MULTIPROCESSING VERSION
# read as dask dataframe
# uncomment only for multiprocessing version
# df = dd.read_csv(r"C:\Users\Bruger\Desktop\P8\NSCmp\smile_records.csv")
start = time.time()

df_lazys = []
df_fasts = []

# define computation but do not compute yet
df_lazy = dask.delayed(((df['endx']/df['startx'])+
                        (df['endy']/df['starty']))/2)
df_lazys.append(df_lazy)

df_lazys[0]

# compute here
df_fast = dask.compute(df_lazys)

df_fasts.append(df_fast)

print (df_fast)
end = time.time()
print (end-start)
```

In this code `dask.delayed` defines the operations that yield the smile ratios and store it into df_lazy to be computed later on in df_fast with `dask.compute()`. This is the standard way `Dask` deals with parallelizing processes.

Time complexity = O(log(n))

Chapter 4

Testing

Different types of implementations have been developed. To determine the performance of each implementation, several tests have been carried out. The aim of the code optimization, is to make the algorithm run faster, and therefore have a smaller execution time. In order to verify whether or not this was achieved, the following tests were performed.

4.1 Performance test

Test 1

Purpose: The purpose of this test is to analyse the execution time of all three algorithms when presented with 1/2 of the data.

Protocol: To perform this test, the data selected and fed to the 3 versions. The code runs, and the execution time is printed.

Results :In the table below, the execution time in seconds, of all three implementations are presented. From the numbers, it can be observed that both, the vectorised and the multiprocessing versions significantly improve the execution time. However, the vectorized version is by far the fastest.

Naive	Vectorised	Multiprocessing
1.6378509998321533	0.0009968280792236328	0.03124547004699707

Test 2

Purpose: The purpose of this test is to analyse the execution time of all three algorithms when presented with all the data.

Protocol: To perform this test, the data selected and fed to the 3 versions. The code runs, and the execution time is printed.

Results : When comparing the results from the table of Test 1, with those from Test 2, it can be appreciated that the Naive version does indeed grow linearly, when the data size was doubled, the execution time was almost doubled as well. The vectorized and multiprocessing versions, again, performed better while the vectorized version stays still the best option for code optimization in this specific case.

Naive	Vectorised	Multiprocessing
3.7442500591278076	0.010192632675170898	0.04682302474975586

From these results, we can see that the naive version grows by a factor of 2.29. The vectorised version grows by a factor of 10.23 and the multiprocessing by a factor of 1.50.

Chapter 5

Conclusion

In conclusion, the software optimization was done by vectorizing a looped operation for getting the smile ratios and then applying a multiprocessing method for the algorithm. However, the results obtained deviated from what was originally expected. The vectorized version of the code performed best as it can be observed in the test outcomes, while the naive version performed worst and the multiprocessing version was in the middle. We expected the multiprocessing version to be the fastest because the data set is split in chunks and running on different processing units simultaneously (in this case 8). We believe this did not happen because a too complex method has been applied to a too simple problem. Thus, in order to make the algorithm more robust and computationally less expensive, more research shall be done as future work on what methods fit the problem best. Meanwhile for this specific task, the vectorization fits best.

Bibliography

[1] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org.

[2] OpenCV. *The OpenCV Library*. https://opencv.org/.

[3] PyData Development Team. *Python Data Analysis Library*. https://pandas.pydata.org/.