

Project Title:
THE FOF-DESIGNER:
DIGITAL DESIGN SKILLS FOR FACTORIES OF THE FUTURE

Project Acronym:
DigiFoF



Grant Agreement number:
2018-2553 / 001-001

Project Nr. 601089-EPP-1-2018-1-RO-EPPKA2-KA


Subject:
ULBS_01 - Workplace safety – Employees emotion recognition

Dissemination Level:
Public

Lead Organisation:
ULBS

Project Coordinator:
ULBS

Trainers:
Eng. Valentin Fleaca

Revision	Preparation date	Period covered	Project start date	Project duration
V1	October 2019	Month 16-36	01/01/2019	36 Months
This project has received funding from the European Union's EACEA Erasmus+ Programme Key Action 2 - Knowledge Alliances under the Grant Agreement No 2018-2533 / 001-001				

Contents

1.	Introduction to Python & OpenCV	2
1.1	Laboratory structure	3
1.2	Setting up the environment.....	3
1.3	Theoretical background.....	7
1.4	Python vs C++	8
	Compilation vs Virtual Machine	8
	Syntax differences	10
	Object-Oriented Programming.....	13
1.5	Python vs Java.....	14
1.6	Exercises	15
1.7	OpenCV.....	16
2.	Face detection	18
2.1.	Laboratory structure	19
2.2.	Theoretical background.....	19
	Haar feature-based cascade classifiers.....	19
	Haar features extraction	20
	Integral Images concept.....	21
	AdaBoost	22
	Cascade of Classifiers	22
	Face detection with OpenCV	23
3.	Detecting facial landmarks.....	25
3.1	Laboratory structure	26
3.2	Theoretical background.....	26
4.	Recognizing facial emotions.....	29
4.1	Laboratory structure	30
4.2	Theoretical background.....	30

1. Introduction to Python & OpenCV

Training specification	Explanation
Organizer	Valentin Fleaca
Training Topic	Introduction to Python & OpenCV
Training objectives	<ul style="list-style-type: none">• getting familiar with Python• handling basic OpenCV API calls
Method	<ul style="list-style-type: none">• individual work
Target groups	<ul style="list-style-type: none">• master students (Computer Science)• software engineers
Recommended composition	Individuals with basic programming knowledge
Recommended size of groups	8 to 12
Training duration	2 hours
Mode of tutoring	Expert input
Mode of provision	Classroom
Tools and resources to be used (technological-support tools)	Computer room with Windows installed
Recommended preparation	Getting familiar with Python and OpenCV
Modes of working in teams	Collective work with distributed role
Communication and cooperation mode	Informal communication
Necessary abilities to tackle the tasks of open problems	Ability to listen and ask questions when something is not clear
Knowledge prerequisites	Basic programming knowledge

1.1 Laboratory structure

Time (minutes)	Objective	Performed by?
5	Presenting the objectives and structure of this laboratory	Teacher
5	Downloading and installing PyCharm & Python	Teacher and students
10	Installing OpenCV, NumPy, SciPy	Teacher and students
15	Python vs C++ vs Java	Teacher
30	Python language exercises	Students
35	OpenCV usage	Teacher and Students

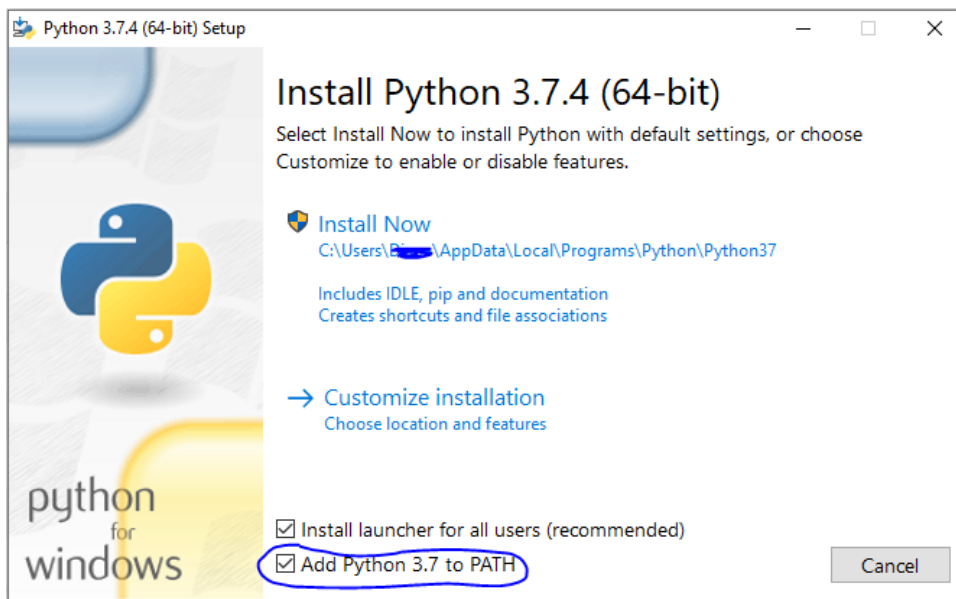
1.2 Setting up the environment

- Download Python 3.7.4 for Windows x64
 - <https://www.python.org/downloads/release/python-374/>

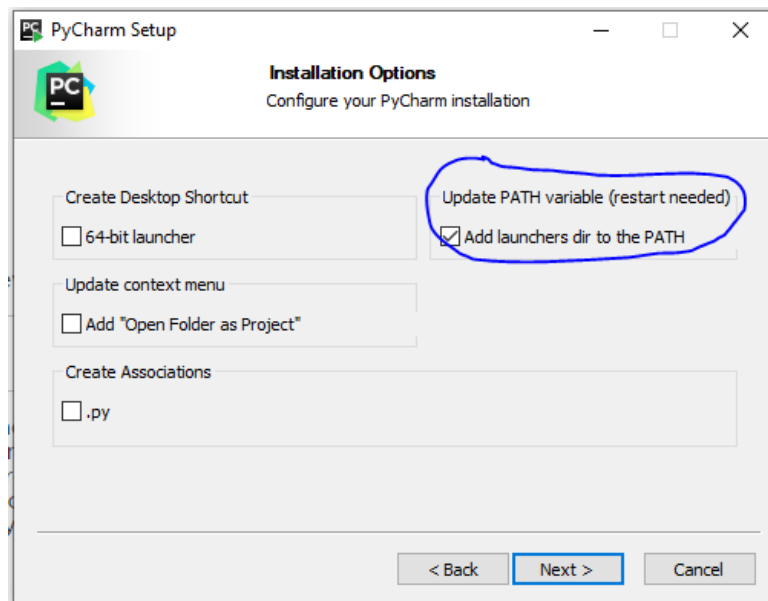
Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		68111671e5b2db4aef7b9ab01bf0f9be	23017663	SIG
XZ compressed source tarball	Source release		d33e4aae66097051c2eca45ee3604803	17131432	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	6428b4fa7583daff1a442cba8cee08e6	34898416	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	5dd605c38217a45773bf5e4a936b241f	28082845	SIG
Windows help file	Windows		d63999573a2c06b2ac56cade6b4f7cd2	8131761	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	9b00c8cf6d9ec0b9abe83184a40729a2	7504391	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a702b4b0ad76debd3043a583e563400	26680368	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	28cb1c608bbd73ae8e53a3bd351b+bd2	1362904	SIG
Windows x86 embeddable zip file	Windows		9fab3b81f8841879fda94133574139d8	6741626	SIG
Windows x86 executable installer	Windows		33cc602942a54446a3d6451476394789	25663848	SIG
Windows x86 web-based installer	Windows		1b670cfa5d317df82c30983ea371d87c	1324608	SIG

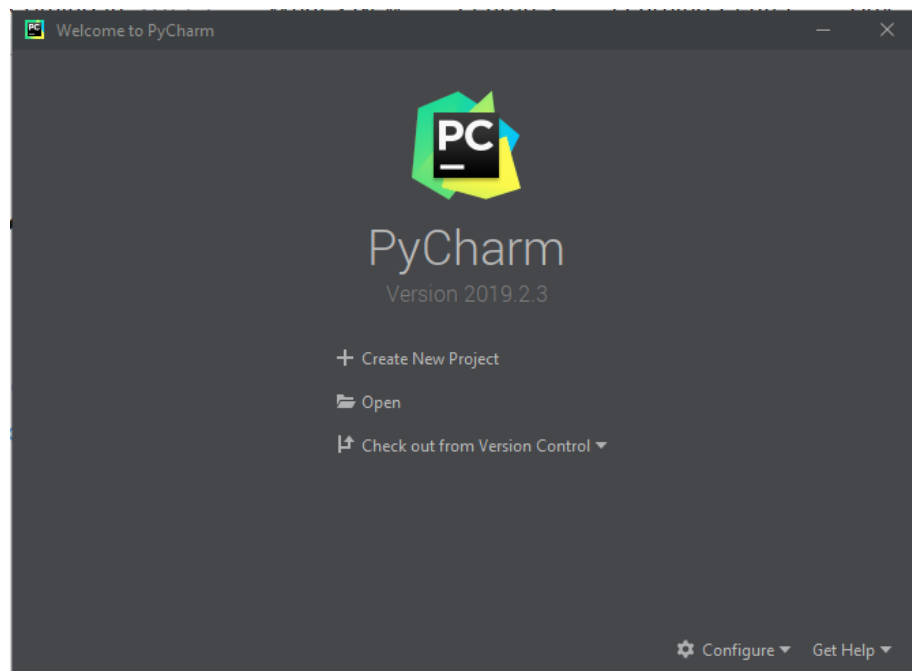
- Install Python and make sure to check „add python to PATH”



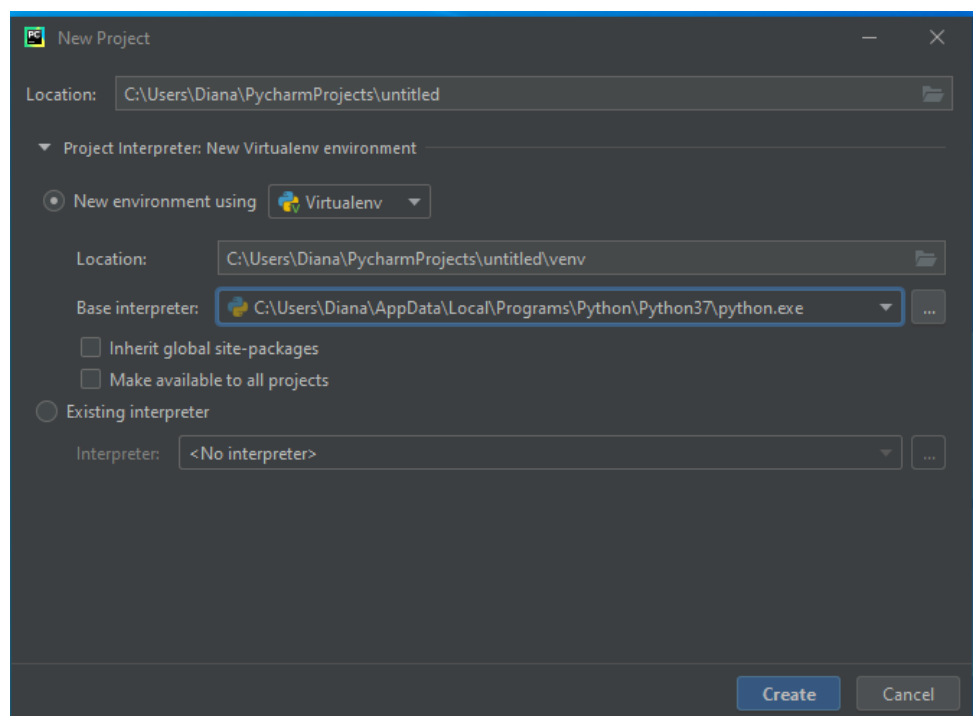
- Download and install PyCharm Community edition
<https://www.jetbrains.com/pycharm/download/#section=windows>
 - Make sure to check the “Update PATH variable” option



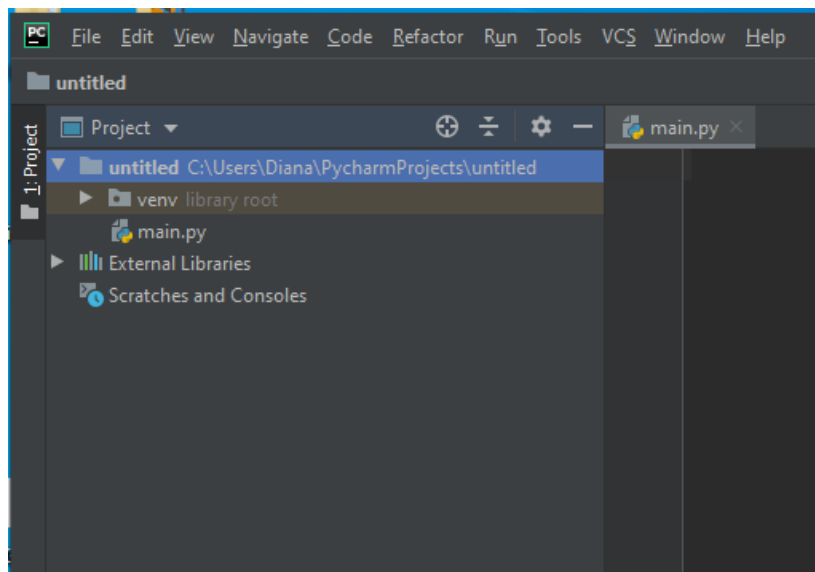
- Open PyCharm
 - Create a new project



- Select the project interpreter as a new Virtualenv environment and select python 3.7 as the base interpreter

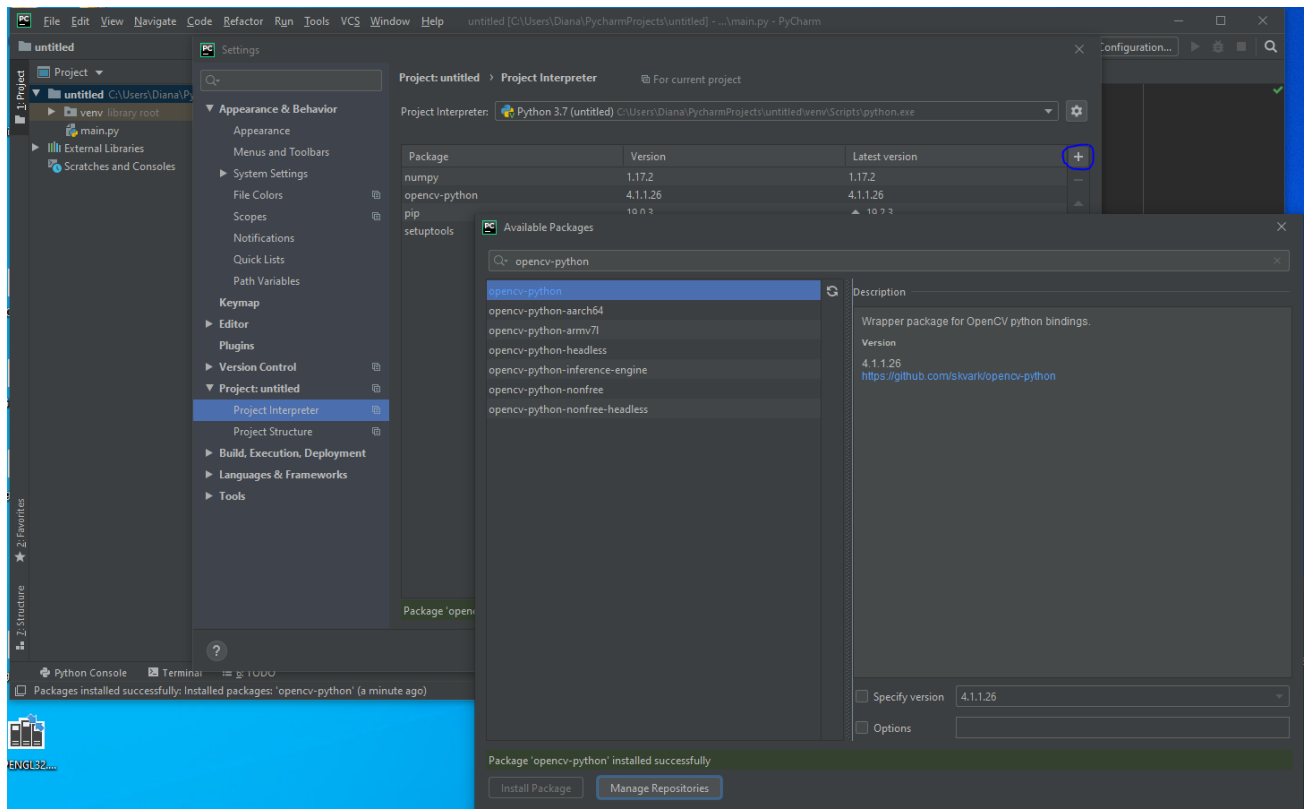


- Right click on the newly created project and create a new file called “main.py”

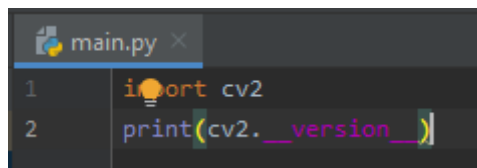


- Go to File -> Settings (Ctrl + Alt + S) -> Project Interpreter -> Install (Alt + Insert) -> Search and install:

- “opencv-python”
- “numpy”
- “scikit-learn”
- “cmake”
- “dlib”



- In main.py write:



```
main.py x
1 import cv2
2 print(cv2.__version__)
```

- Right click on main.py and click “Run ‘main’”

1.3 Theoretical background

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.

An interpreted language is any programming language that isn't already in "machine code" prior to runtime. Unlike compiled languages , an interpreted language's translation doesn't happen beforehand. Translation occurs at the same time as the program is being executed.

For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. The dynamic semantics (also known as execution semantics) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics.

When we declare a variable in C or alike languages, this sets aside an area of memory for holding values allowed by the data type of the variable. The memory allocated will be interpreted as the data type suggests. If it's an integer variable the memory allocated will be read as an integer and so on. When we assign or initialize it with some value, that value will get stored at that memory location. At compile time, initial value or assigned value will be checked. So we cannot mix types. Example: initializing a string value to an int variable is not allowed and the program will not compile.

But Python is a dynamically typed language. It doesn't know about the type of the variable until the code is run. So declaration is of no use. What it does is, It stores that value at some memory location and then binds that variable name to that memory container. And makes the

contents of the container accessible through that variable name. So the data type does not matter. As it will get to know the type of the value at run-time.

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the type of the object determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse.

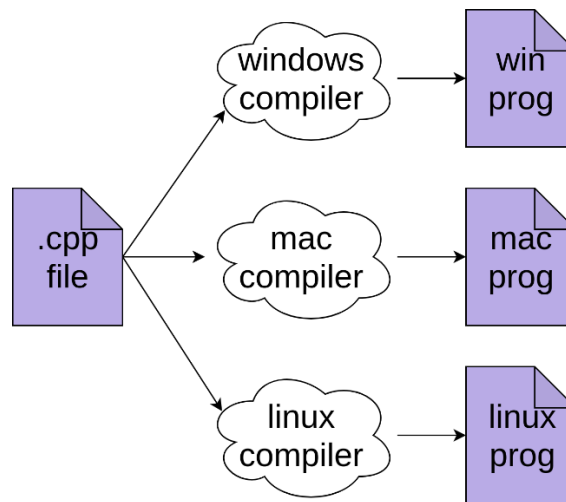
1.4 Python vs C++

Compilation vs Virtual Machine

In C++, you use a compiler that converts your source code into machine code and produces an executable. The executable is a separate file that can then be run as a stand-alone program:

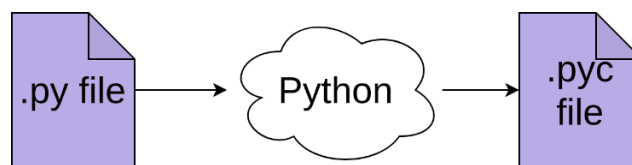


This process outputs actual machine instructions for the specific processor and operating system it's built for. In this drawing, it's a Windows program. This means you'd have to recompile your program separately for Windows, Mac, and Linux:



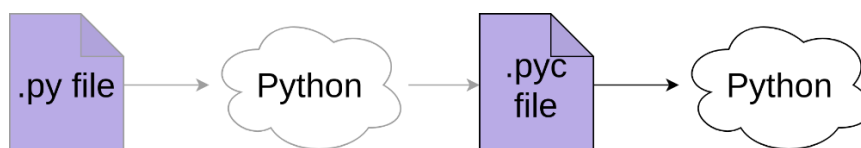
You'll likely need to modify your C++ code to run on those different systems as well.

Python, on the other hand, uses a different process, it runs each time you execute your program. It compiles your source just like the C++ compiler. The difference is that Python compiles to bytecode instead of native machine code. Bytecode is the native instruction code for the Python virtual machine. To speed up subsequent runs of your program, Python stores the bytecode in .pyc files:

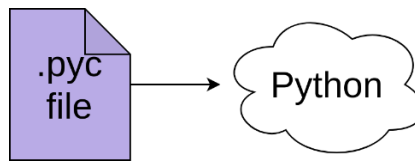


If you're using Python 2, then you'll find these files next to the .py files. For Python 3, you'll find them in a `__pycache__` directory.

The generated bytecode doesn't run natively on your processor. Instead, it's run by the Python virtual machine. This is similar to the Java virtual machine or the .NET Common Runtime Environment. The initial run of your code will result in a compilation step. Then, the bytecode will be interpreted to run on your specific hardware:



As long as the program hasn't been changed, each subsequent run will skip the compilation step and use the previously compiled bytecode to interpret:



Interpreting code is going to be slower than running native code directly on the hardware. So why does Python work that way? Well, interpreting the code in a virtual machine means that only the virtual machine needs to be compiled for a specific operating system on a specific processor. All of the Python code it runs will run on any machine that has Python.

Syntax differences

The first thing most developers notice when comparing Python vs C++ is the “whitespace issue.” Python uses leading whitespace to mark scope. This means that the body of an if block or other similar structure is indicated by the level of indentation. C++ uses curly braces ({}) to indicate the same idea.

Instead of relying on a lexical marker like ; to end each statement, Python uses the end of the line. If you need to extend a statement over a single line, then you can use the backslash (\) to indicate that. (Note that if you're inside a set of parentheses, then the continuation character is not needed.)

The way you'll use Boolean expressions changes slightly in Python vs C++. In C++, you can use numeric values to indicate true or false, in addition to the built-in values. Anything that evaluates to 0 is considered false, while every other numeric value is true.

Python has a similar concept but extends it to include other cases. The basics are quite similar. The Python documentation states that the following items evaluate to False:

Constants defined as false:

- None
- False

Zeros of any numeric type:

- 0
- 0.0
- 0j
- Decimal(0)
- Fraction(0, 1)

Empty sequences and collections:

- "
- ()
- []
- {}
- set()
- range(0)

C++ OPERATOR	PYTHON OPERATOR
&&	and
 	or
!	not
&	&
 	

Unlike Python, C++ has variables that are assigned to a memory location, and you must indicate how much memory that variable will use:

C++

```
int an_int;
float a_big_array_of_floats[REALLY_BIG_NUMBER];
```

In Python, all objects are created in memory, and you apply labels to them. The labels themselves don't have types, and they can be put on any type of object:

Python

>>>

```
>>> my_flexible_name = 1
>>> my_flexible_name
1
>>> my_flexible_name = 'This is a string'
>>> my_flexible_name
'This is a string'
>>> my_flexible_name = [3, 'more info', 3.26]
>>> my_flexible_name
[3, 'more info', 3.26]
>>> my_flexible_name = print
>>> my_flexible_name
<built-in function print>
```

Python has a language feature called list comprehensions. While it's possible to emulate list comprehensions in C++, it's fairly tricky. In Python, they're a basic tool that's taught to beginning programmers.

One way of thinking about list comprehensions is that they're like a super-charged initializer for lists, dicts, or sets. Given one iterable object, you can create a list, and filter or modify the original as you do so:

Python

>>>

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
```

This script starts with the iterable `range(5)` and creates a list that contains the square for each item in the iterable. It's possible to add conditions to the values in the first iterable:

Python

>>>

```
>>> odd_squares = [x**2 for x in range(5) if x % 2]
>>> odd_squares
[1, 9]
```

While it's true that you can create a vector of the squares of the odd numbers in C++, doing so usually means a little more code:

C++

```
std::vector<int> odd_squares;
for (int ii = 0; ii < 10; ++ii) {
    if (ii % 2) {
        odd_squares.push_back(ii);
    }
}
```

C++ has a rich set of algorithms built into the standard library. Python has a similar set of built-in functions that cover the same ground.

The first and most powerful of these is the `in` operator, which provides a quite readable test to see if an item is included in a list, set, or dictionary:

Python

>>>

```
>>> x = [1, 3, 6, 193]
>>> 6 in x
True
>>> 7 in x
False
>>> y = { 'Jim' : 'gray', 'Zoe' : 'blond', 'David' : 'brown' }
>>> 'Jim' in y
True
>>> 'Fred' in y
False
>>> 'gray' in y
False
```

Object-Oriented Programming

Inheritance between classes works similarly in Python vs C++. A new class can inherit methods and attributes from one or more base classes, just like you've seen in C++. Some of the details are a bit different, however.

Base classes in Python do not have their constructor called automatically like they do in C++. This can be confusing when you're switching languages.

Multiple inheritance also works in Python, and it has just as many quirks and strange rules as it does in C++.

Similarly, you can also use composition to build classes, where you have objects of one type hold other types. Considering everything is an object in Python, this means that classes can hold anything else in the language.

Python has no concept of access modifiers for classes. Everything in a class object is public. Thus, Python has far weaker encapsulation support than C++.

Python does not have operator overloads, but it has its own garbage collection.

<i>Feature</i>	<i>Python</i>	<i>C++</i>
<i>Faster execution</i>		X
<i>Cross-Platform Execution</i>	X	
<i>Single-Type Variables</i>		X
<i>Multiple-Type Variables</i>	X	
<i>Comprehensions</i>	X	
<i>Rich set of Built-In Algorithms</i>	X	X
<i>Static Typing</i>		X
<i>Dynamic Typing</i>	X	
<i>Strict Encapsulation</i>		X
<i>Direct Memory Control</i>		X
<i>Garbage Collection</i>	X	

1.5 Python vs Java

Python programs are generally expected to run slower than Java programs, but they also take much less time to develop. Python programs are typically 3-5 times shorter than equivalent Java programs. This difference can be attributed to Python's built-in high-level data types and its dynamic typing. For example, a Python programmer wastes no time declaring the types of arguments or variables, and Python's powerful polymorphic list and dictionary types, for which rich syntactic support is built straight into the language, find a use in almost every Python program. Because of the run-time typing, Python's run time must work harder than Java's. For

example, when evaluating the expression $a+b$, it must first inspect the objects a and b to find out their type, which is not known at compile time. It then invokes the appropriate addition operation, which may be an overloaded user-defined method. Java, on the other hand, can perform an efficient integer or floating point addition, but requires variable declarations for a and b , and does not allow overloading of the $+$ operator for instances of user-defined classes.

For these reasons, Python is much better suited as a "glue" language, while Java is better characterized as a low-level implementation language. In fact, the two together make an excellent combination. Components can be developed in Java and combined to form applications in Python; Python can also be used to prototype components until their design can be "hardened" in a Java implementation. To support this type of development, a Python implementation written in Java is under development, which allows calling Python code from Java and vice versa. In this implementation, Python source code is translated to Java bytecode (with help from a run-time library to support Python's dynamic semantics).

1.6 Exercises

- A. Sort a list by the last element in each tuple from a given list.

Sample List : [(2, 5), (1, 2), (4, 4), (2, 3), (2, 1)]

Expected Result : [(2, 1), (1, 2), (2, 3), (4, 4), (2, 5)]

- B. Find the second smallest number in a list.

Sample List: [1, 1, 1, 0, 0, 0, 2, -2, -2]

Expected Result: 1

- C. Find common items from two lists.

Sample Lists:

["Red", "Green", "Orange", "White"]

["Black", "Green", "White", "Pink"]

Expected result: ['Green', 'White']

- D. Generate and print a dictionary that contains a number between 1 and n in the form $(x, x * x)$

Input: 10

Expected Result: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}

E. Write a function to calculate the geometric sum of n-1

Example:
$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \dots$$

Expected output: `geometric_sum(7)` => 1.9921875

F. Count the frequency of words in a file

G. Find the largest prime factor of a given number

The prime factors of 330 are 2, 3, 5 and 11. Therefore 11 is the largest prime factor of 330.

H. Count the number of letters and digits of a given sequence

Example: hello world! 123

Expected output: Letters 10, Digits 3

1.7 OpenCV

Reading an image using OpenCV:

```
import cv2
import numpy as np
img = cv2.imread(your_image.jpg')
```

Displaying the image:

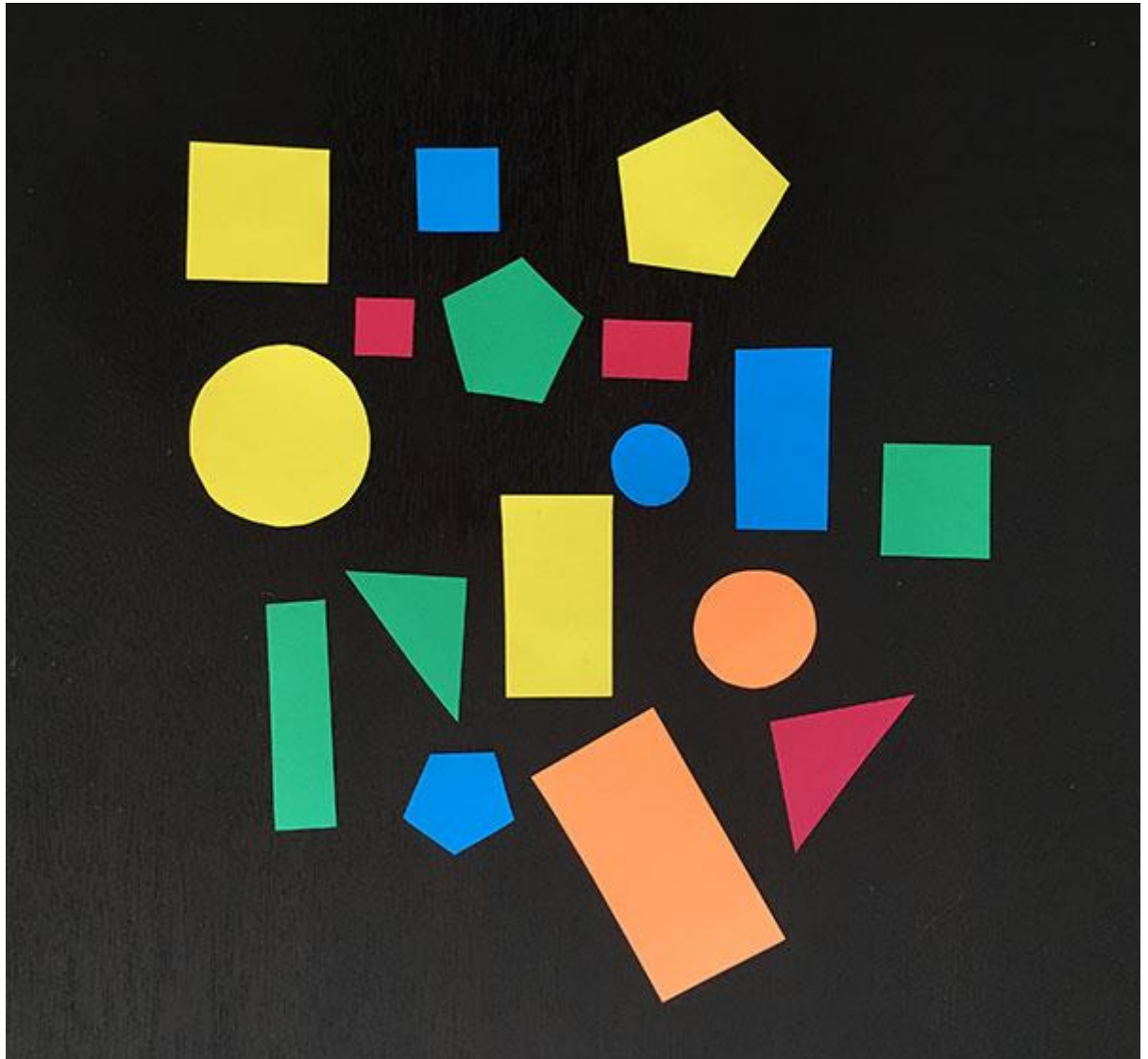
```
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

cv2.waitKey() is a keyboard binding function. Its argument is the time in milliseconds. The function waits for specified milliseconds for any keyboard event. If you press any key in that time, the program continues. If 0 is passed, it waits indefinitely for a key stroke. It can also be set to detect specific key strokes like, if key a is pressed etc which we will discuss below.

cv2.destroyAllWindows() simply destroys all the windows we created. If you want to destroy any specific window, use the function `cv2.destroyWindow()` where you pass the exact window name as the argument.

Exercises

- A. Apply the geometric transformations over an image
- B. Apply morphological operations over an image (dilation, erosion)
- C. Transform an image to grayscale
- D. Count the number of frames in a video
- E. Find the center of each shape in the following image (hint: lookup **cv2.findContours**)



2. Face detection

Training specification	Explanation
Organizer	Valentin Fleaca
Training Topic	Implementing a face detection system
Training objectives	<ul style="list-style-type: none"> • Face detection in static images • Face detection in a video stream
Method	<ul style="list-style-type: none"> • individual work
Target groups	<ul style="list-style-type: none"> • master students (Computer Science) • software engineers
Recommended composition	Individuals with basic programming knowledge
Recommended size of groups	8 to 12
Training duration	2 hours
Mode of tutoring	Expert input
Mode of provision	Classroom
Tools and resources to be used (technological-support tools)	Computer room with Windows installed
Recommended preparation	Getting familiar with Python and OpenCV
Modes of working in teams	Collective work with distributed role
Communication and cooperation mode	Informal communication
Necessary abilities to tackle the tasks of open problems	Ability to listen and ask questions when something is not clear
Knowledge prerequisites	Basic Python and OpenCV understanding

2.1. Laboratory structure

Time (minutes)	Objective	Performed by?
5	Presenting the objectives and structure of this laboratory	Teacher
25	Face detection: theory	Teacher
20	Detecting faces in images	Teacher and students
30	Detecting faces in video sequences	Students
20	Project architecture setup	Teacher and students

2.2. Theoretical background

Face detection is a technique that identifies or locates human faces in digital images. A typical example of face detection occurs when we take photographs through our smartphones, and it instantly detects faces in the picture. Face detection is different from Face recognition. Face detection detects merely the presence of faces in an image while facial recognition involves identifying whose face it is. In this article, we shall only be dealing with the former.

Face detection is performed by using classifiers. A classifier is essentially an algorithm that decides whether a given image is positive(face) or negative(not a face). A classifier needs to be trained on thousands of images with and without faces. Fortunately, OpenCV already has two pre-trained face detection classifiers, which can readily be used in a program. The two classifiers are:

- Haar Classifier
- Local Binary Pattern Classifier

Haar feature-based cascade classifiers

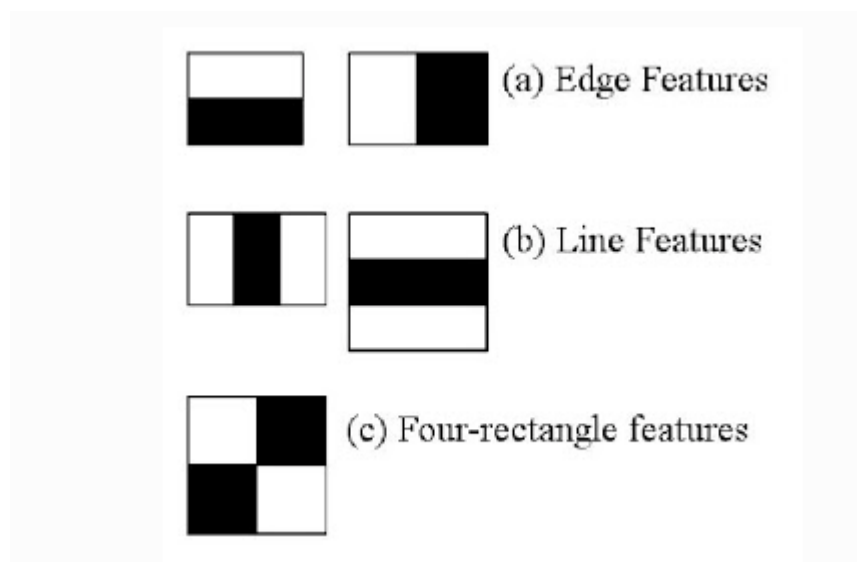
Haar-like features are digital image features used in object recognition. They owe their name to their intuitive similarity with Haar wavelets and were used in the first real-time face detector. Paul Viola and Michael Jones in their paper titled "Rapid Object Detection using a Boosted Cascade of Simple Features" used the idea of Haar-feature classifier based on the Haar wavelets. This classifier is widely used for tasks like face detection in computer vision industry.

Haar cascade classifier employs a machine learning approach for visual object detection which is capable of processing images extremely rapidly and achieving high detection rates. This can be attributed to three main reasons:

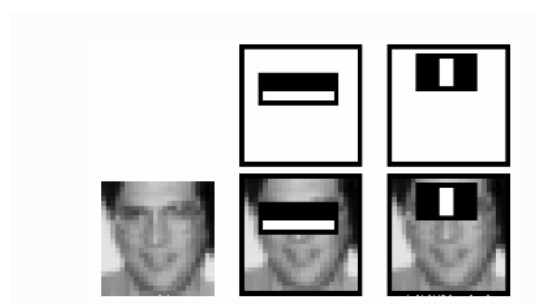
- Haar classifier employs 'Integral Image' concept which allows the features used by the detector to be computed very quickly.
- The learning algorithm is based on AdaBoost. It selects a small number of important features from a large set and gives highly efficient classifiers.
- More complex classifiers are combined to form a 'cascade' which discard any non-face regions in an image, thereby spending more computation on promising object-like regions.

Haar features extraction

After the tremendous amount of training data (in the form of images) is fed into the system, the classifier begins by extracting Haar features from each image. Haar Features are kind of convolution kernels which primarily detect whether a suitable feature is present on an image or not. Some examples of Haar features are mentioned below:



These Haar Features are like windows and are placed upon images to compute a single feature. The feature is essentially a single value obtained by subtracting the sum of the pixels under the white region and that under the black. The process can be easily visualized in the example below.

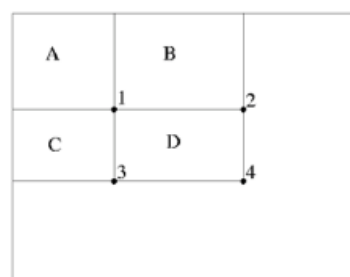


For demonstration purpose, let's say we are only extracting two features, hence we have only two windows here. The first feature relies on the point that the eye region is darker than the adjacent cheeks and nose region. The second feature focuses on the fact that eyes are kind of darker as compared to the bridge of the nose. Thus, when the feature window moves over the eyes, it will calculate a single value. This value will then be compared to some threshold and if it passes that it will conclude that there is an edge here or some positive feature.

Integral Images concept

The algorithm proposed by Viola Jones uses a 24X24 base window size, and that would result in more than 180,000 features being calculated in this window. Imagine calculating the pixel difference for all the features? The solution devised for this computationally intensive process is to go for the Integral Image concept. The integral image means that to find the sum of all pixels under any rectangle, we simply need the four corner values.

Integral image



$$\begin{aligned} \text{Sum of all pixels in } D &= 1 + 4 - (2 + 3) \\ &= A + (A + B + C + D) - (A + C + A + B) \\ &= D \end{aligned}$$

This means, to calculate the sum of pixels in any feature window, we do not need to sum them up individually. All we need is to calculate the integral image using the 4 corner values. The example below will make the process transparent.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

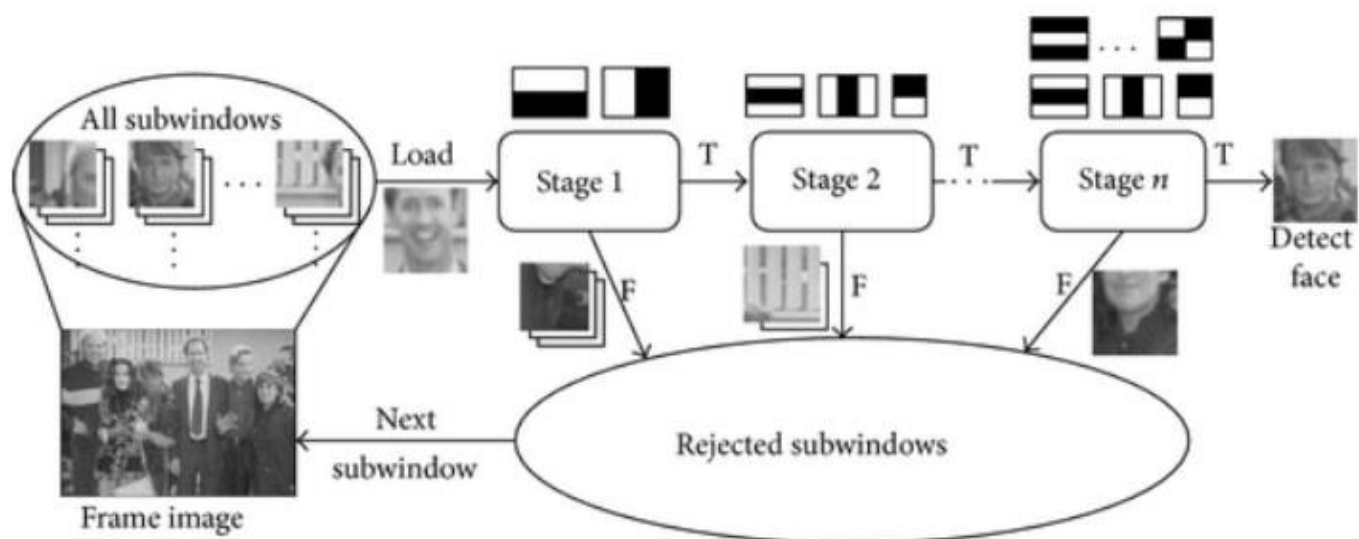
$$\begin{aligned} &15 + 16 + 14 + 28 + 27 + 11 = \\ &101 + 450 - 254 - 186 = 111 \end{aligned}$$

AdaBoost

As pointed out above, more than 180,000 features values result within a 24X24 window. However, not all features are useful for identifying a face. To only select the best feature out of the entire chunk, a machine learning algorithm called Adaboost is used. What it essentially does is that it selects only those features that help to improve the classifier accuracy. It does so by constructing a strong classifier which is a linear combination of a number of weak classifiers. This reduces the amount of features drastically to around 6000 from around 180,000.

Cascade of Classifiers

Another way by which Viola Jones ensured that the algorithm performs fast is by employing a cascade of classifiers. The cascade classifier essentially consists of stages where each stage consists of a strong classifier. This is beneficial since it eliminates the need to apply all features at once on a window. Rather, it groups the features into separate sub-windows and the classifier at each stage determines whether or not the sub-window is a face. In case it is not, the sub-window is discarded along with the features in that window. If the sub-window moves past the classifier, it continues to the next stage where the second stage of features is applied. The process can be understood with the help of the diagram below.



face_detection_harcascades.webp

Click ->

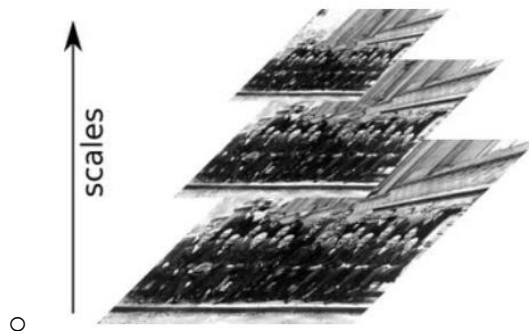
Face detection with OpenCV

```
import cv2

# Load the cascade
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# Read the input image
img = cv2.imread('test.jpg')
# Convert into grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Detect faces
faces = face_cascade.detectMultiScale(gray, 1.1, 4)
# Draw rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
# Display the output
cv2.imshow('img', img)
cv2.waitKey()
```

cv2.CascadeClassifier.detectMultiScale parameters are:

- image : Matrix of the type CV_8U containing an image where objects are detected.
- scaleFactor : Parameter specifying how much the image size is reduced at each image scale.



- This scale factor is used to create scale pyramid as shown in the picture. Suppose, the scale factor is 1.03, it means we're using a small step for resizing, i.e. reduce size by 3 %, we increase the chance of a matching size with the model for detection is found, while it's expensive.
- minNeighbors : Parameter specifying how many neighbors each candidate rectangle should have to retain it. This parameter will affect the quality of the detected faces: higher value results in less detections but with higher quality.
- flags : Parameter with the same meaning for an old cascade as in the function cvHaarDetectObjects. It is not used for a new cascade.
- minSize : Minimum possible object size. Objects smaller than that are ignored.
- maxSize : Maximum possible object size. Objects larger than that are ignored.

Exercises

- A. Test the code for face detection over a single image
- B. Given the *face_benchmarks.rar*
 - parse the *annotations.txt* file in the following way:
 - i. Each line contains a path to an image and delimited by TAB the number of faces in the respective image. Example:
1. 2002/07/19/big/img_423 1
 - Open each one of the 100 .jpg's from the two folders 2002 and 2003 and try to apply your face detection over the image. Validate your result with the actual result from the annotations.txt.
 - Hints:
 - i. try to tune the parameters of detectMultiScale
 - ii. try to use multiple haarcascade models (.xmls) from inside *haar-cascade-files.zip* over the same image.
- C. Write your own code to detect faces in each frame of a live or static video
- D. Implement your own face tracking algorithm in a video sequence. You need to determine if a face detected in frame n is the same as the face detected in frame $n+1$. Hint: Think of Euclidian distances between the faces from the two frames.

3. Detecting facial landmarks

Training specification	Explanation
Organizer	Valentin Fleaca
Training Topic	Understanding what facial landmarks are and how they can be detected.
Training objectives	<ul style="list-style-type: none"> Understanding what facial landmarks are and how they can be detected. Getting familiar with SciKit API calls
Method	<ul style="list-style-type: none"> individual work
Target groups	<ul style="list-style-type: none"> master students (Computer Science) software engineers
Recommended composition	Individuals with basic programming knowledge
Recommended size of groups	8 to 12
Training duration	2 hours
Mode of tutoring	Expert input
Mode of provision	Classroom
Tools and resources to be used (technological-support tools)	Computer room with Windows installed
Recommended preparation	Getting familiar with Python and OpenCV
Modes of working in teams	Collective work with distributed role
Communication and cooperation mode	Informal communication
Necessary abilities to tackle the tasks of open problems	Ability to listen and ask questions when something is not clear
Knowledge prerequisites	Basic Python and machine learning knowledge

3.1 Laboratory structure

Time (minutes)	Objective	Performed by?
5	Presenting the objectives and structure of this laboratory	Teacher
45	Detecting facial landmarks	Teacher
50	OpenCV exercises	Students

3.2 Theoretical background

What are facial landmarks?

Facial landmarks represent salient regions of the face, such as:

- Eyes
- Eyebrows
- Nose
- Mouth
- Jawline

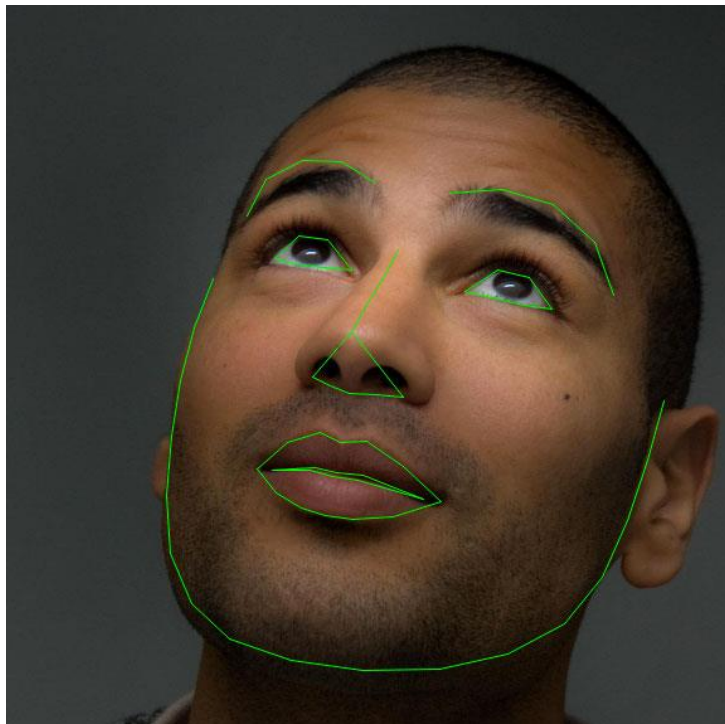


Fig 3.2. Facial landmarks are used to label and identify key facial attributes in an image

Detecting facial landmarks is a subset of the shape prediction problem. Given an input image (and normally an ROI that specifies the object of interest), a shape predictor attempts to localize key points of interest along the shape.

In the context of facial landmarks, our goal is detect important facial structures on the face using shape prediction methods.

Detecting facial landmarks is therefore a two step process:

- **Step #1:** Localize the face in the image.
- **Step #2:** Detect the key facial structures on the face ROI.

The facial landmark detector included in the **dlib** library is an implementation of the One Millisecond Face Alignment with an Ensemble of Regression Trees paper by Kazemi and Sullivan (2014).

This method starts by using:

- A training set of labeled facial landmarks on an image. These images are manually labeled, specifying specific (x, y)-coordinates of regions surrounding each facial structure.
- Priors, or more specifically, the probability on distance between pairs of input pixels.

Given this training data, an ensemble of regression trees are trained to estimate the facial landmark positions directly from the pixel intensities themselves (i.e., no “feature extraction” is taking place).

The end result is a facial landmark detector that can be used to **detect facial landmarks in real-time** with **high quality predictions**.

For more information and details on this specific technique please check the powerpoint presentation “Understanding Kazemi’s and Sullivan’s paper.pptx”.

Exercises

A. Fill in the gaps for the facial landmarks predictor code:

```
import dlib
import cv2
import numpy as np

def drawLandmarks(image):
    #load the face detector
    face_cascade = ...
    #load the landmarks predictor
    landmarksPredictor = dlib.shape_predictor(...)

    #convert to grayscale
    gray = ...
    #detect the faces
    faces = ...

    for (x, y, w, h) in faces:
        #convert face rectangle to dlib.rectangle
        rect = dlib.rectangle(left = x, top = y, right = x + w, bottom = y + h)
        #detect the facial landmarks
        shape = landmarksPredictor(gray, rect)
        #convert result to (x,y) list of points
        shape = _shape_to_np(shape)

        #draw the face ROI
        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 1)

        currentMark = 0
        #iterate over the facial landmarks
        for (x, y) in shape:
            cv2.putText(image, str(currentMark), (x - 10, y - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.2, (0, 255, 0), 1)
            cv2.circle(image, (x, y), 1, (0, 0, 255), -1)
            currentMark += 1

def _shape_to_np(shape, dtype="int"):
    coords = np.zeros((68, 2), dtype=dtype)

    for i in range(0, 68):
        coords[i] = (shape.part(i).x, shape.part(i).y)

    return coords

if __name__ == '__main__':
    #read a test image
    image = ...

    #draw landmarks on image
    drawLandmarks(image)

    #display the image
    ...
    cv2.waitKey()
```

B. Connect the facial landmark points into regions so that it would look similar to image 3.1.

C. Create a function that will calculate the Euclidian distance between two 2D points given as input. Test it to find out the distance between the eyes and the mouth.

4. Recognizing facial emotions

Training specification	Explanation
Organizer	Valentin Fleaca
Training Topic	Recognizing facial emotions
Training objectives	<ul style="list-style-type: none"> Understanding facial emotions Recognize human emotions from live video sequences
Method	<ul style="list-style-type: none"> individual work
Target groups	<ul style="list-style-type: none"> master students (Computer Science) software engineers
Recommended composition	Individuals with basic programming knowledge
Recommended size of groups	8 to 12
Training duration	2 hours
Mode of tutoring	Expert input
Mode of provision	Classroom
Tools and resources to be used (technological-support tools)	Computer room with Windows installed
Recommended preparation	Getting familiar with Python and OpenCV
Modes of working in teams	Collective work with distributed role
Communication and cooperation mode	Informal communication
Necessary abilities to tackle the tasks of open problems	Ability to listen and ask questions when something is not clear
Knowledge prerequisites	Basic Python and SciKit knowledge

4.1 Laboratory structure

Time (minutes)	Objective	Performed by?
5	Presenting the objectives and structure of this laboratory	Teacher
20	Facial expressions and emotions	Teacher
25	Learning a classifier to recognize facial emotions from a dataset	Teacher and students
30	Tuning the classifier parameters to increase accuracy	Students
20	Live face emotion recognition system	Students

4.2 Theoretical background

What are facial emotions?

Facial expressions are the voluntary and involuntary movements occurring when engaging one or more of the 43 facial muscles on the face. They are a rich source of nonverbal communication and display a vast amount of information about emotion and cognition.

There are seven basic human emotions (BEs): happiness, surprise, anger, sadness, fear, disgust, and neutral, as shown in figure 4.2.

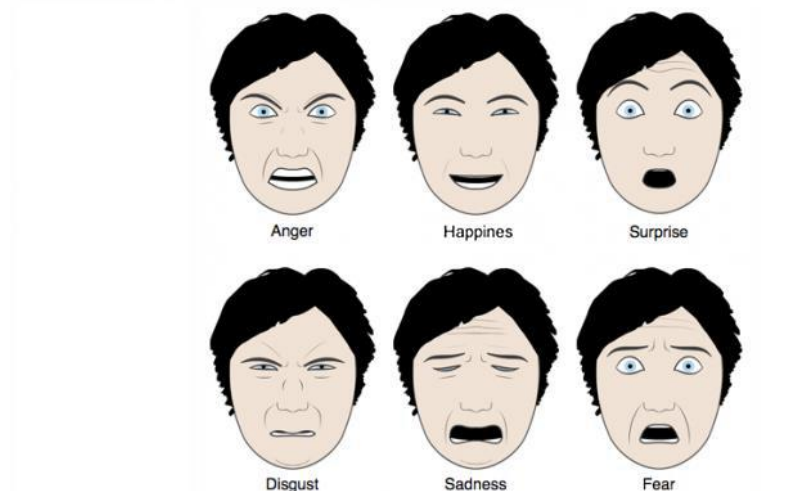


Fig 4.2.1 Six of the seven (neutral missing) basic human emotions

Recognizing facial emotions

Recognizing facial emotions can be obtained by either approach:

- Conventional FER (facial emotion recognition) methods
- Deep-learning FER methods

Conventional FER methods

The shared trait of these methodologies is detecting the face area and extracting geometric characteristics, appearance characteristics, or a hybrid of geometric and appearance characteristics on the respective face.

In conventional FER approaches, the system is made of three main steps: (1) face and facial component or landmark detection, (2) feature extraction and (3) expression classification. Initial, a face picture is recognized from an input image, and facial parts (e.g. nose and eyes) or landmarks are distinguished from the face region. Second, different features (characteristics) are extracted from the facial parts. Third, the pre-trained FE classifiers, such as an AdaBoost classifier, support vector machine (SVM) or random forest, give the recognition output based on the extracted features.

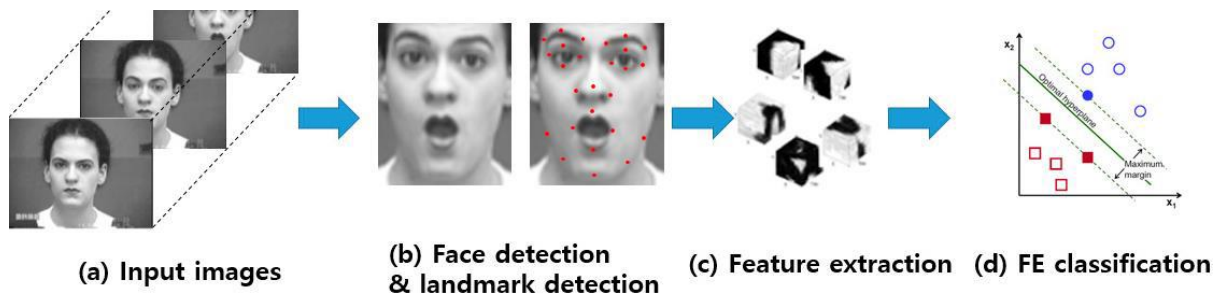


Fig. 4.2.2 Steps for a conventional FER procedure

Conventional approaches require computing power and memory to be relatively lower than deep learning approaches. Because of their low computational complexity and high degree of accuracy, these approaches are still being studied for use in real-time embedded systems.

Deep-learning FER methods

Deep-learning based FER approaches greatly decrease the reliance on models, based on face physics, and other pre-processing procedures by allowing “end-to-end” learning to happen in the pipeline directly from the input images. Among the several deep-learning models

available, the most popular network model is the convolutionary neural network (CNN). In CNN-based methodologies, the input image is converted to produce a feature map through a filter collection in the convolution layers. Then each feature map is combined to fully connected networks and the face expression is recognized as belonging to the Softmax algorithm's output based on a particular class.

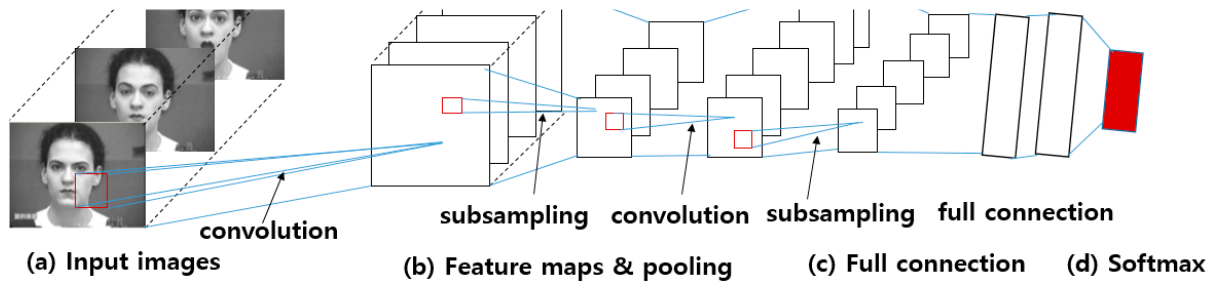


Fig. 4.2.3 Steps for a CNN-based FER approach: (a) input images are convolved using filters in the convolution layers. (b) Feature maps are constructed from the convolution results and max-pooling (subsampling) layers decrease the spatial resolution of the given feature maps. (c) CNNs apply completely associated neural network layers behind the convolutional layers and (d) recognize a single face expression based on the Softmax output

Unlike conventional approaches, deep learning-based approaches commonly determine features and classifiers by deep neural networks experts. Deep learning-based approaches extract optimal features using deep convolutionary neural networks with the desired features directly from data. However, gathering a large amount of training data for facial emotion under the various conditions is not easy enough to learn deep neural networks. In addition, in order to operate training and testing, deep learning-based approaches require a higher-level and massive computing device than conventional approaches.

Classification algorithms

Most known supervised learning algorithms are:

- 1 Multilayer perceptron classifier (MLPC)
- 2 Support Vector Classifier (SVC)
- 3 K-Nearest Neighbors Classifier (kNN)
- 4 Decision Tree Classifier (DTC)

- 5 Random Forest Classifier (RFC)
- 6 AdaBoost Classifier (ABC)
- 7 Gaussian Naïve Bayes Classifier (GNBC)
- 8 Quadratic Discriminant Analysis (QDA)

Support Vector Machine

Support vector machines (SVMs) are a part of the supervised learning methods used for classification, regression and outliers detections. A support vector machine constructs a hyper-plane in a high dimensional space. Intuitively, a good separation is obtained by the hyper-plane that has the largest distance to the nearest training data points of any class (this is called the functional margin), since in general the larger the margin is, the lower the error of the classifier.

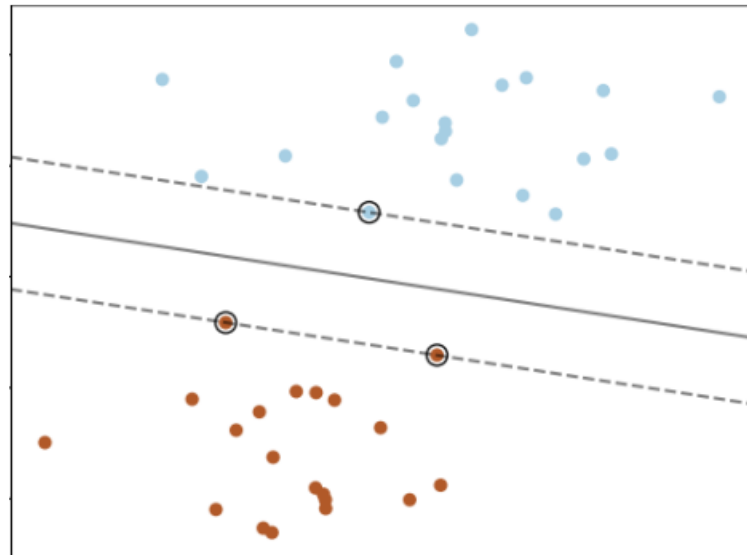
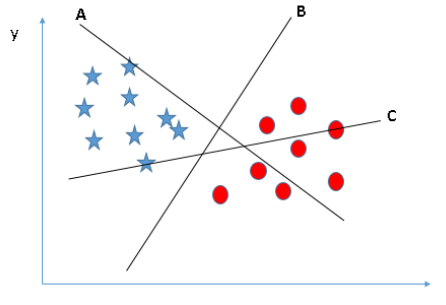


Fig. 4.2.4 SVM hyperplanes. The circled dots are the support vectors.

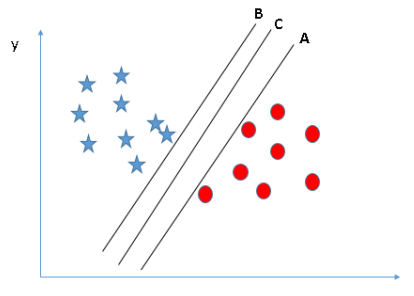
How can we identify the right hyper-plane?

- **Identify the right hyper-plane (Scenario-1):** Here, we have three hyper-planes (A, B and C).

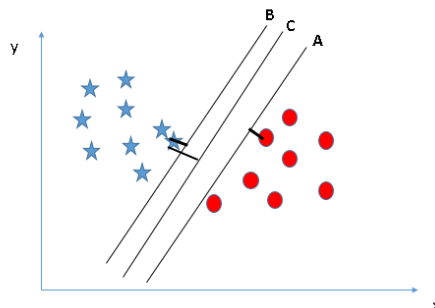


You need to remember a thumb rule to identify the right hyper-plane: “Select the hyper-plane which segregates the two classes better”. In this scenario, hyper-plane “B” has excellently performed this job.

- **Identify the right hyper-plane (Scenario-2):** Here, we have three hyper-planes (A, B and C) and all are segregating the classes well.



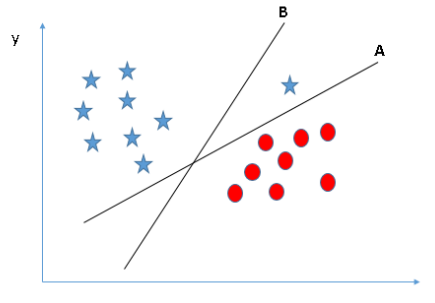
Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as Margin.



Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we

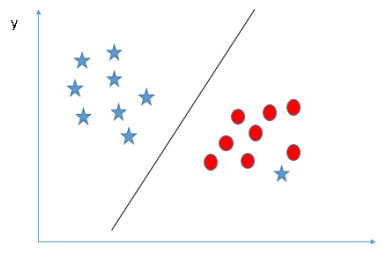
select a hyper-plane having low margin then there is high chance of misclassification.

- **Identify the right hyper-plane (Scenario-3)**



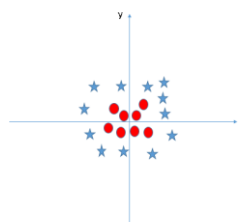
Some of you may have selected the hyper-plane B as it has higher margin compared to A. But, here is the catch, SVM selects the hyper-plane which classifies the classes accurately prior to maximizing margin. Here, hyper-plane B has a classification error and A has classified all correctly. Therefore, the right hyper-plane is A.

- **Can we classify two classes (Scenario-4)?** Below, I am unable to segregate the two classes using a straight line, as one of star lies in the territory of other(circle) class as an outlier.

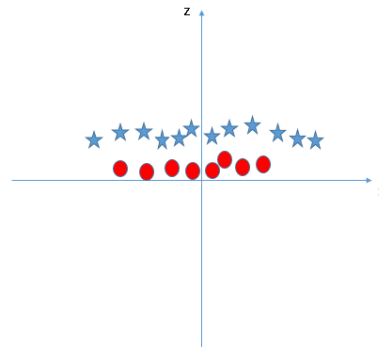


SVM has a feature to ignore outliers and find the hyper-plane that has maximum margin. Hence, we can say, SVM is robust to outliers.

- **Find the hyper-plane to segregate to classes (Scenario-5)** In the scenario below, we can't have linear hyper-plane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyper-plane.



SVM can solve this problem easily! It solves this problem by introducing additional feature. Here, we will add a new feature $z=x^2+y^2$. Now, let's plot the data points on axis x and z:

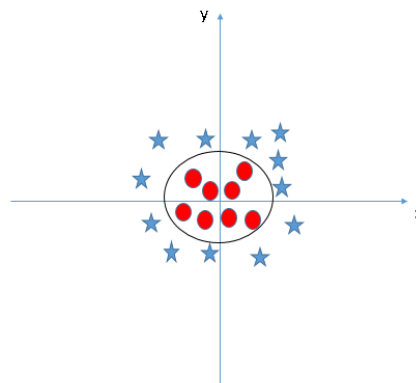


In above plot, points to consider are:

- All values for z would be positive always because z is the squared sum of both x and y
- In the original plot, red circles appear close to the origin of x and y axes, leading to lower value of z and star relatively away from the origin result to higher value of z.

In SVM, it is easy to have a linear hyper-plane between these two classes. But, another burning question which arises is, should we need to add this feature manually to have a hyper-plane. No, SVM has a technique called the kernel trick. These are functions which takes low dimensional input space and transform it to a higher dimensional space i.e. it converts not separable problem to separable problem, these functions are called kernels. It is mostly useful in non-linear separation problem. Simply put, it does some extremely complex data transformations, then find out the process to separate the data based on the labels or outputs you've defined.

When we look at the hyper-plane in original input space it looks like a circle:



Exercises

- A. In the previous laboratory you've implemented a facial landmarks detector able to detect 68 facial points of interest. Using the function from bellow, calculate 39 Euclidian distances in this exact order and return a list.

```
def computeDistances(landmarks):
    current_distances = []
    #left eyebrow - eye
    current_distances.append(euclidianDistance(landmarks[17], landmarks[36]))
    current_distances.append(euclidianDistance(landmarks[18], landmarks[37]))
    current_distances.append(euclidianDistance(landmarks[19], landmarks[38]))
    current_distances.append(euclidianDistance(landmarks[20], landmarks[38]))
    current_distances.append(euclidianDistance(landmarks[21], landmarks[39]))

    #left eye
    current_distances.append(euclidianDistance(landmarks[37], landmarks[41]))
    current_distances.append(euclidianDistance(landmarks[38], landmarks[40]))

    #right eye
    current_distances.append(euclidianDistance(landmarks[43], landmarks[47]))
    current_distances.append(euclidianDistance(landmarks[44], landmarks[46]))

    #right eyebrow - eye
    current_distances.append(euclidianDistance(landmarks[22], landmarks[42]))
    current_distances.append(euclidianDistance(landmarks[23], landmarks[43]))
    current_distances.append(euclidianDistance(landmarks[24], landmarks[44]))
    current_distances.append(euclidianDistance(landmarks[25], landmarks[44]))
    current_distances.append(euclidianDistance(landmarks[26], landmarks[45]))

    #left eyebrow - right eyebrow
    current_distances.append(euclidianDistance(landmarks[21], landmarks[22]))

    #eyebrows - nose
    current_distances.append(euclidianDistance(landmarks[21], landmarks[27]))
    current_distances.append(euclidianDistance(landmarks[22], landmarks[27]))

    #nose
    current_distances.append(euclidianDistance(landmarks[27], landmarks[28]))
    current_distances.append(euclidianDistance(landmarks[28], landmarks[29]))
    current_distances.append(euclidianDistance(landmarks[29], landmarks[30]))

    #nose - nostrils
    current_distances.append(euclidianDistance(landmarks[30], landmarks[31]))
    current_distances.append(euclidianDistance(landmarks[30], landmarks[32]))
    current_distances.append(euclidianDistance(landmarks[30], landmarks[33]))
    current_distances.append(euclidianDistance(landmarks[30], landmarks[34]))
    current_distances.append(euclidianDistance(landmarks[30], landmarks[35]))

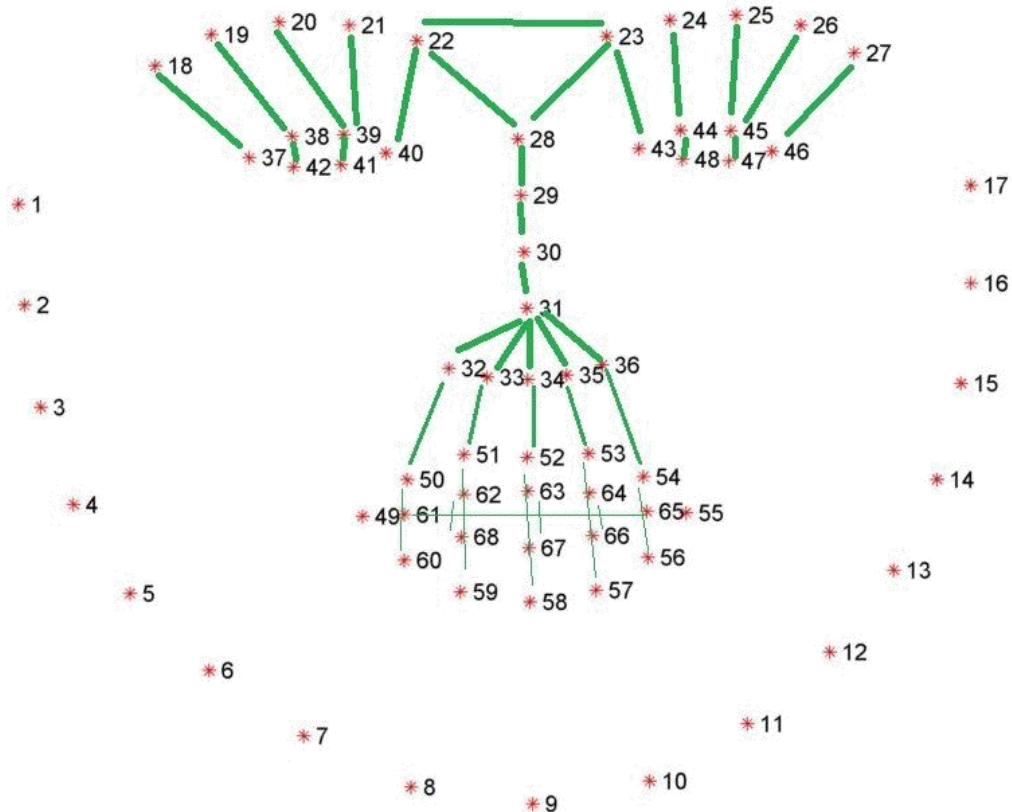
    #mouth
    current_distances.append(euclidianDistance(landmarks[49], landmarks[59]))
    current_distances.append(euclidianDistance(landmarks[50], landmarks[58]))
    current_distances.append(euclidianDistance(landmarks[51], landmarks[57]))
    current_distances.append(euclidianDistance(landmarks[52], landmarks[56]))
    current_distances.append(euclidianDistance(landmarks[53], landmarks[55]))
    current_distances.append(euclidianDistance(landmarks[61], landmarks[67]))
    current_distances.append(euclidianDistance(landmarks[62], landmarks[66]))
    current_distances.append(euclidianDistance(landmarks[63], landmarks[65]))
    current_distances.append(euclidianDistance(landmarks[60], landmarks[64]))

    #mouth - nostrils
    current_distances.append(euclidianDistance(landmarks[31], landmarks[49]))
    current_distances.append(euclidianDistance(landmarks[32], landmarks[50]))
    current_distances.append(euclidianDistance(landmarks[33], landmarks[51]))
    current_distances.append(euclidianDistance(landmarks[34], landmarks[52]))
    current_distances.append(euclidianDistance(landmarks[35], landmarks[53]))

    return current_distances
```

This function does a feature selection over the 68 points. The input parameter “landmarks” is a list of 2D points [(x1,y1), (x2,y2)....,(x68,y68)]. Adapt your “drawLandmarks” code from previous laboratory so that it will return this list.

The function calculates the following distances marked with green:



B. Fill in the gaps for a live emotion detector

```
import numpy as np
import dlib
import cv2
import math
import pickle

def getLandmarks(originalImage, faceImage, faceRect, landmarksPredictor):
    x, y, w, h = faceRect
    # convert face rectangle to dlib.rectangle
    rect = dlib.rectangle(left=x, top=y, right=x + w, bottom=y + h)
    # detect the facial landmarks
    shape = landmarksPredictor(faceImage, rect)
    # convert result to (x,y) list of points
    shape = _shape_to_np(shape)

    currentMark = 0
    # iterate over the facial landmarks
    for (x, y) in shape:
        cv2.putText(originalImage, str(currentMark), (x - 10, y - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.2, (0, 255, 0), 1)
        cv2.circle(originalImage, (x, y), 1, (0, 0, 255), -1)
        currentMark += 1

    return shape
```

```

def runLive():
    #load the face detector
    face_cascade = ...
    #load the landmarks predictor
    landmarksPredictor = ...
    #load the trained classifier for face emotion
    clasifier = pickle.load(open("classifierModel.cache", 'rb'))

    cap = cv2.VideoCapture(0)

    while(True):
        # Capture frame-by-frame
        ret, frame = cap.read()

        # convert to grayscale
        gray = ...

        # detect the faces
        faces = ...

        for (x, y, w, h) in faces:
            # draw the face ROI
            cv2.rectangle(...)

            #select only the face from the whole image
            faceImage = gray[x:x+w, y:y+h]

            #get the landmarks
            landmarks = getLandmarks(frame, faceImage, (x, y, w, h), landmarksPredictor)

            #compute the distances
            distances = computeDistances(landmarks)

            #predict the emotion
            predictedEmotion = clasifier.predict(np.array(distances).reshape(1, -1))
            cv2.putText(frame, str(predictedEmotion), (int(x) - 10, int(y) - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

        # Display the resulting frame
        cv2.imshow('frame', frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # When everything done, release the capture
    cap.release()
    cv2.destroyAllWindows()

if __name__ == '__main__':
    runLive();

```