MODULE 3

Computer Vision & Image Processing

Code Walkthrough Notes — Labs 4.1 · 4.2 · 5.1 · 5.2

Lab	Title	File	Lines of Code
4.1	Face Detection & Recognition (DeepFace)	lab4_1_face_recognition.py	~334
4.2	OCR Pipeline with Tesseract	lab4_2_ocr_tesseract.py	~463
5.1	Azure Computer Vision API	lab5_1_azure_computer_vision.py	~280
5.2	Azure Face API	lab5_2_azure_face_api.py	~310

B.Tech AI Specialization · Chitkara University · February 26–27, 2026

LAB 4.1

Face Detection & Recognition

Duration: 120 minutes · Target: ≥ 90% recognition accuracy on 20 test photos

What This File Does

lab4_1_face_recognition.py builds a complete face recognition system in six self-contained sections. Each section is a function you call independently. You run them in order: set up the database folder, capture photos, verify pairs, identify individuals, run a live webcam loop, then evaluate accuracy.

Section	Function	What it Does
Section 1	Setup (runs at import)	Creates database/ folder; prints path
Section 2	capture_face_photos()	Opens webcam; saves N photos per person to database/
Section 3	verify_two_faces()	Answers: are these two specific images the same person?
Section 4	identify_face()	Answers: who is this person from the whole database?
Section 5	run_realtime_recognition()	Live webcam loop; press c to identify, q to quit
Section 6	evaluate_accuracy()	Batch-tests all images in test_images/; saves JSON report

Imports & Global Setup (Lines 1–33)

```
import os
import cv2
import json
import time
import numpy as np
from deepface import DeepFace

DATABASE_PATH = './database'
os.makedirs(DATABASE_PATH, exist_ok=True)
```

Import	Why It's Needed
os	Builds folder paths; lists directory contents; checks if folders exist
cv2 (OpenCV)	Reads webcam frames; draws bounding boxes and text on screen; saves images
json	Serialises the accuracy report dictionary to a .json file

time	Not heavily used here — available for rate-limiting if needed
numpy as np	Required internally by OpenCV and DeepFace for array operations
DeepFace	The main library — wraps VGG-Face, ArcFace, FaceNet; handles detection + embedding + comparison

Why os.makedirs with exist_ok=True?

exist_ok=True means: create the folder if it doesn't exist, but do NOT raise an error if it already exists. Without it, running the script twice would crash on the makedirs line with FileExistsError.

Section 2 — capture_face_photos() (Lines 42–84)

This function opens the webcam and lets you manually capture face photos by pressing the spacebar. It saves each photo into database/<person_name>/ so DeepFace can use it as a registered identity.

The Webcam Loop

```
cap = cv2.VideoCapture(0)      # 0 = first connected camera

while count < num_photos:
    ret, frame = cap.read()    # ret = success bool, frame = numpy array (H×W×3)
    if not ret: break

    display = frame.copy()     # copy() so we draw on display without changing frame
    cv2.putText(display, ...)  # overlay text on the display copy
    cv2.imshow('Capture Face', display)

    key = cv2.waitKey(1)       # wait 1ms for keypress; returns -1 if none
    if key == 27:  break       # 27 = ESC key ASCII code
    elif key == 32:            # 32 = SPACE key ASCII code
        cv2.imwrite(filename, frame)   # save the ORIGINAL frame (not display copy)
        count += 1

cap.release()                  # free the webcam resource
cv2.destroyAllWindows()        # close all OpenCV windows
```

frame.copy() — Why This Matters

We draw status text on 'display', not on 'frame'.
If we drew on 'frame' directly, the saved JPEG would have overlay text burned into the face photo — ruining the embedding quality.
Always: display = frame.copy() before drawing anything you don't want saved.

ASCII Key Codes to Remember

27 = ESC → used to cancel/exit loops
32 = SPACE → used to trigger capture
ord('q') = 113 → used in the live recognition loop
ord('c') = 99 → used to trigger identification
cv2.waitKey(1) → non-blocking; returns -1 if no key pressed within 1ms

Section 3 — verify_two_faces() (Lines 91–111)

Verification is a 1-to-1 comparison. You give it two specific image paths and it tells you whether those two photos show the same person. It does NOT search a database.

```
result = DeepFace.verify(
    img1_path='database/alice/alice1.jpg',
    img2_path='database/alice/alice2.jpg',
    model_name='VGG-Face',       # Which CNN generates the face embedding
    distance_metric='cosine',    # How similarity between embeddings is
measured
    enforce_detection=True       # Crash if no face found (good for
debugging)
)
```

The result dictionary returned by DeepFace.verify() contains:

Key	Type	What It Means
result['verified']	bool	True = same person (distance < threshold). False = different people
result['distance']	float	Actual cosine distance between the two embeddings. Lower = more similar
result['threshold']	float	The decision boundary. For VGG-Face + cosine this is ~0.40
result['model']	string	Confirms which backbone model was used
result['similarity_metric']	string	Confirms the distance metric used (cosine / euclidean)

How the Decision Is Made

DeepFace extracts a 128-d or 512-d embedding vector from each face.
It computes cosine distance between the two vectors.
distance < threshold → verified = True (same person)
distance >= threshold → verified = False (different people)

Cosine distance formula: $d = 1 - (A \cdot B / |A||B|)$
Range: 0.0 (identical) to 2.0 (complete opposite)
VGG-Face threshold: ~0.40 FaceNet: ~0.40 ArcFace: ~0.68

Section 4 — identify_face() (Lines 119–151)

Identification is a 1-to-N search. You give it one query image and it scans every photo in the entire database folder to find the closest match. This is the core function for the live recognition system.

```
results_df = DeepFace.find(
    img_path=query_image_path,    # The unknown face you want to identify
    db_path=DATABASE_PATH,        # Root of your registered faces folder
    model_name='VGG-Face',
    distance_metric='cosine',
    enforce_detection=True,
    silent=True                   # Suppresses the tqdm progress bar
)
```

results_df Is a LIST — Most Common Bug in This Lab

DeepFace.find() returns a LIST of DataFrames, one per detected face.
If your query image has 1 face, the list has 1 element: results_df[0]
If your query image has 3 faces, the list has 3 elements.

WRONG (will crash): results_df.iloc[0]['identity']
RIGHT: results_df[0].iloc[0]['identity']
↑
You must index into the LIST first, then into the DataFrame.

Extracting the Person's Name from the File Path

```
top_match   = results_df[0].iloc[0]          # Best match row (lowest
distance)
matched_path = top_match['identity']          # e.g.
'./database/alice/alice2.jpg'

# os.path.dirname removes the filename:  './database/alice'
# os.path.basename takes the last part:  'alice'
person_name = os.path.basename(os.path.dirname(matched_path))

distance = top_match['distance']              # Similarity score (lower =
closer match)
```

The database DataFrame is sorted by distance ascending — row 0 is always the best (closest) match. If the distance is very high (> 0.7), the match is unreliable and you should treat it as Unknown.

Section 5 — run_realtime_recognition() (Lines 159–224)

This is the live webcam loop. It continuously shows the camera feed with a face bounding box drawn around detected faces. When you press 'c', it captures the frame, saves it to a temp file, identifies the face, and displays the name on screen.

Two-Layer Face Detection

The live loop uses two different detection systems for two different purposes:

Detection System	Purpose	Speed	Accuracy
Haar Cascade (cv2)	Draw a green bounding box on screen in real time — runs every frame	Very fast — runs at 30 FPS	Lower — just for visual feedback
DeepFace.find()	Actually identify who the person is — runs only on keypress	Slow — 1–5 seconds	High — uses CNN embeddings

> **Why Use Two Systems?**
> DeepFace identification is too slow to run 30 times per second.
> Haar Cascade is fast enough to run every frame but not accurate enough for identity.
> So: Haar Cascade draws the box live (good UX), DeepFace identifies on demand (press c).
> This is a common pattern in production face recognition systems.

Haar Cascade Detection Code

```
face_cascade = cv2.CascadeClassifier(
    cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'
)

# Every frame:
gray  = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  # Cascade needs grayscale
faces = face_cascade.detectMultiScale(
    gray,
    scaleFactor=1.1,   # How much the image is reduced at each scale (1.1 = 10%)
    minNeighbors=5,    # How many neighbours a detection needs before it's kept
    minSize=(80, 80)   # Minimum face size in pixels — ignores tiny detections
)
for (x, y, w, h) in faces:
    cv2.rectangle(display, (x, y), (x+w, y+h), (0, 255, 0), 2)  # Green box
```

Identification on Keypress

```
elif key == ord('c'):
    temp_path = '/tmp/temp_query.jpg'
    cv2.imwrite(temp_path, frame)           # Save the current frame to disk
    name, dist = identify_face(temp_path)   # Run DeepFace identification
    last_result = (name, dist)              # Store result; display it next frame
```

last_result persists across frames — so the name stays on screen after identification until you press 'c' again. The label colour is green for a known person, red for Unknown.

Section 6 — evaluate_accuracy() (Lines 232–298)

This function walks through a test_images/ folder, runs identify_face() on every image, compares the predicted name against the true name (from the folder name), and saves a full JSON accuracy report.

```
for person_name in os.listdir(test_folder):          # Loop over person
folders
    person_dir = os.path.join(test_folder, person_name)
    if not os.path.isdir(person_dir): continue       # Skip files, only
process folders

    for img_file in os.listdir(person_dir):          # Loop over test images
        if not img_file.lower().endswith(('.jpg','.jpeg','.png')): continue

        img_path = os.path.join(person_dir, img_file)
        predicted_name, distance = identify_face(img_path)

        match = predicted_name.lower() == person_name.lower()  # Case-
insensitive compare
        correct += int(match)    # int(True)=1, int(False)=0
        total   += 1
```

The JSON Report Structure

```
# lab4_1_accuracy_report.json will look like this:
{
  'total_images': 20,
  'correct': 18,
  'accuracy_percent': 90.0,
  'target_percent': 90,
  'passed': true,
  'details': [
    {
      'image': 'alice_test1.jpg',
      'true_label': 'alice',
      'predicted': 'alice',
      'distance': 0.2341,
      'correct': true
    },
    ...
  ]
}
```

> **How to Run the Full Lab in Order**
>
> 1. Uncomment the capture loop and run once per person to build database/
>
> for name in ['alice','bob','carol','david','eve']:
>
> capture_face_photos(name, num_photos=5)

2. Uncomment verify_two_faces() calls and test a same-person and different-person pair

3. Uncomment identify_face() and test a single image

4. Uncomment run_realtime_recognition() for the live demo

5. Create test_images/ with 4 NEW photos per person (not used in database/)
   Then: evaluate_accuracy('./test_images')
   This produces lab4_1_accuracy_report.json for submission

LAB 4.2

OCR Pipeline with Tesseract

Duration: 90 minutes · Target: Extract & structure key invoice fields with regex

What This File Does

lab4_2_ocr_tesseract.py builds a complete document OCR pipeline in six sections. It starts with a raw image, cleans it through five preprocessing steps, runs Tesseract to extract text, then uses regular expressions to pull structured fields like dates, amounts, and invoice numbers into a JSON output. A batch processor at the end handles the 50-invoice Mini Project.

Section	Function	What it Does
Section 1	basic_ocr()	Runs raw Tesseract on an image with zero preprocessing
Section 2	preprocess_for_ocr()	5-step pipeline: grayscale→denoise→binarise→deskew→upscale
Section 2b	deskew_image()	Helper: detects skew angle via Hough lines; rotates to correct
Section 3	compare_ocr_accuracy()	Runs both raw and preprocessed OCR; reports character count and accuracy
Section 4	extract_invoice_fields()	Uses 6 regex patterns to extract dates, amounts, invoice no., emails, phones, vendor
Section 5	process_invoice()	Full pipeline for one image: preprocess → OCR → extract → return dict
Section 6	process_invoice_batch()	Loops over a folder; calls process_invoice() on each; saves combined JSON

Imports & Startup Check (Lines 1–34)

```
import cv2, re, json, os, numpy as np
import pytesseract
from PIL import Image

# Verify Tesseract binary is reachable
try:
    version = pytesseract.get_tesseract_version()
    print(f'[INFO] Tesseract version: {version}')
except Exception:
    print('[ERROR] Tesseract not found.')
```

Import	Why It's Needed
cv2 (OpenCV)	Loads images; applies grayscale, denoising, thresholding, deskew, resize

re	Regular expressions — finds patterns like dates, currencies, invoice numbers in text
json	Saves structured output as .json files
pytesseract	Python wrapper around the Tesseract OCR binary
PIL.Image	Converts numpy arrays to PIL format that pytesseract accepts
numpy as np	Array operations; used in deskew_image() angle calculations

pytesseract is Just a Wrapper — Tesseract Must Be Installed Separately

pip install pytesseract only installs the Python wrapper.
You ALSO need the Tesseract binary installed at the system level:

Windows: download from https://github.com/UB-Mannheim/tesseract/wiki
Then add to PATH or set: pytesseract.pytesseract.tesseract_cmd
Mac: brew install tesseract
Linux: sudo apt install tesseract-ocr

If Tesseract binary is missing, every OCR call throws TesseractNotFoundError.

Section 1 — basic_ocr() (Lines 41–60)

Runs Tesseract on a raw image with no preprocessing at all. Used as a baseline to compare against the preprocessed version.

```
def basic_ocr(image_path):
    img = cv2.imread(image_path)          # Loads as BGR numpy array

    # Convert BGR → RGB because Tesseract expects RGB channel order
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    pil_img = Image.fromarray(img_rgb)    # pytesseract works with PIL Images

    # --psm 6: treat image as a single uniform block of text
    # Good for invoices and documents with consistent layout
    text = pytesseract.image_to_string(pil_img, config='--psm 6')
    return text
```

PSM Mode	Meaning	Best For
--psm 3	Fully automatic page segmentation (default)	Complex layouts with mixed text/images
--psm 6	Single uniform block of text	Invoices, forms, typed documents
--psm 11	Sparse text — find as much text as possible	Screenshots, mixed media images

--psm 13	Raw line — treat image as a single text line	Single-line labels, licence plates

Section 2 — preprocess_for_ocr() (Lines 67–126)

This is the most important function in the file. Five chained steps transform a raw, noisy image into a clean binary image that Tesseract can read accurately.

Step 1 — Grayscale Conversion

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# img shape was (H, W, 3)   →   gray shape is now (H, W)
# Reduces 3-channel colour noise to a single luminance channel
# Tesseract internally converts to grayscale anyway — doing it first gives
# us control over how the conversion is done
```

Step 2 — Denoising

```
denoised = cv2.fastNlMeansDenoising(
    gray,
    h=10,                  # Filter strength. Higher = more smoothing but
blurs edges
    templateWindowSize=7,  # Size of patch used to compute weights (must be
odd)
    searchWindowSize=21    # Window searched for similar patches (must be
odd)
)
# Removes scanner/camera 'salt-and-pepper' noise — random bright/dark pixels
# that Tesseract would misread as punctuation or characters
```

Step 3 — Adaptive Thresholding

```
binary = cv2.adaptiveThreshold(
    denoised,
    255,                                # Max output pixel value (white)
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,    # Each pixel's threshold = Gaussian-
weighted
                                        # mean of its neighbourhood minus C
    cv2.THRESH_BINARY,                  # Output: pixel > threshold → 255, else
→ 0
    blockSize=11,    # Size of the neighbourhood used to compute local
threshold
                     # MUST be an odd number (11, 21, 31...)
    C=2              # Constant subtracted from the neighbourhood mean
                     # Higher C = more aggressive binarisation
)
# Why adaptive and not global?
# Global: _, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
#   Problem: one shadow across the invoice → entire shadow region goes black
# Adaptive: computes a different threshold for each 11×11 pixel region
```

```
#   Result: handles uneven lighting, shadows, aged paper perfectly
```

Step 4 — Deskewing (via deskew_image())

```
def deskew_image(gray_img):
    # 1. Detect edges in the binary image
    edges = cv2.Canny(gray_img, 50, 150, apertureSize=3)

    # 2. Find straight lines using Probabilistic Hough Transform
    lines = cv2.HoughLinesP(edges,
        rho=1,               # Distance resolution (pixels)
        theta=np.pi/180,     # Angle resolution (1 degree)
        threshold=100,       # Minimum votes to count as a line
        minLineLength=100,   # Minimum line length (ignores short lines)
        maxLineGap=10        # Max gap between segments to join them
    )

    # 3. Compute each line's angle from horizontal
    angles = []
    for line in lines:
        x1, y1, x2, y2 = line[0]
        angle = np.degrees(np.arctan2(y2-y1, x2-x1))
        if abs(angle) < 45:     # Only nearly-horizontal text lines
            angles.append(angle)

    # 4. Take the median angle (robust to outliers from non-text lines)
    median_angle = np.median(angles)
    if abs(median_angle) < 0.5: return gray_img  # Already straight — skip

    # 5. Rotate image by the negative of the detected skew
    center = (gray_img.shape[1]//2, gray_img.shape[0]//2)
    M = cv2.getRotationMatrix2D(center, median_angle, scale=1.0)
    rotated = cv2.warpAffine(gray_img, M, (gray_img.shape[1],
gray_img.shape[0]),
        flags=cv2.INTER_CUBIC,
        borderMode=cv2.BORDER_REPLICATE  # Fill new edges with border pixels
    )
    return rotated
```

Step 5 — Upscaling

```
h, w = deskewed.shape[:2]
if w < 1000:                    # Proxy for 'is this image low resolution?'
    deskewed = cv2.resize(
        deskewed, None,
        fx=2, fy=2,             # Scale both dimensions by 2×
        interpolation=cv2.INTER_CUBIC   # Cubic: best quality for upscaling
    )
# Tesseract's LSTM model needs characters to be at least 20–30px tall
# Low-res scans (<150 DPI) often have 10–15px characters — too small
# Doubling the image size makes characters large enough to recognise
```

Section 4 — extract_invoice_fields() (Lines 212–281)

After Tesseract produces raw text, this function uses six different regex patterns to find and extract structured fields. Understanding each pattern is important for the exam.

Date Patterns

```
# Pattern 1: DD/MM/YYYY or MM/DD/YYYY
dates = re.findall(r'\b\d{2}/\d{2}/\d{4}\b', text)
#   \b = word boundary (matches edge of a number sequence)
#   \d{2} = exactly 2 digits
#   / = literal slash
#   \d{4} = exactly 4 digits

# Pattern 2: ISO format YYYY-MM-DD
dates += re.findall(r'\b\d{4}-\d{2}-\d{2}\b', text)

# Pattern 3: '15 January 2024' or '15 Jan 2024'
dates += re.findall(r'\b\d{1,2}\s+\w{3,9}\s+\d{4}\b', text, re.IGNORECASE)
```

Currency Amount Patterns

```
# Matches: $1,234.56  $1234  ₹1,200  ₹800.00
amounts = re.findall(r'[\$₹][\d,]+\.?\d{0,2}', text)
#   [\$₹]    = dollar sign OR rupee symbol
#   [\d,]+   = one or more digits or commas (handles 1,234 format)
#   \.?      = optional decimal point
#   \d{0,2}  = 0, 1, or 2 decimal digits

# Matches labelled total: 'Total: Rs. 2,150.00' or 'Amount Due: $500'
total = re.search(
    r'(?:total|amount due|grand total)[:\s]*[\$₹Rs.]*\s*([\d,]+\.?\d{0,2})',
    text, re.IGNORECASE
)
# (?:...) = non-capturing group — matches but doesn't save the label
# ([\d,]+\.?\d{0,2}) = CAPTURING group — this is what .group(1) returns
```

Invoice Number Pattern

```
inv = re.search(
    r'invoice\s*(?:no|number|#)[:\s]*([\w\-]+)',
    text, re.IGNORECASE
)
# \s* = zero or more whitespace between 'invoice' and 'no'
# (?:no|number|#) = matches any of these three labels (non-capturing)
# [:\s]* = optional colon and/or spaces after the label
# ([\w\-]+) = captures the actual invoice number (letters, digits, hyphens)
```

```
    fields['invoice_number'] = inv.group(1) if inv else None
    # .group(0) = the full match  |  .group(1) = first capturing group
```

Section 5 — process_invoice() (Lines 288–331)

This is the end-to-end pipeline for a single invoice image. It chains all previous functions together and returns a structured dictionary.

```
def process_invoice(image_path):
    # Step 1: Clean the image
    preprocessed = preprocess_for_ocr(image_path, save_debug=True)

    # Step 2: Run OCR on the clean image
    pil_img  = Image.fromarray(preprocessed)
    raw_text = pytesseract.image_to_string(pil_img, config='--psm 6 --oem 3')
    # --oem 3: use both Legacy and LSTM engines; highest accuracy

    # Step 3: Extract structured fields via regex
    fields = extract_invoice_fields(raw_text)

    # Step 4: Build the output dictionary
    invoice_data = {
        'source_file': os.path.basename(image_path),
        'raw_text': raw_text,
        'extracted_fields': fields,
        'validation': {
            'has_date':   bool(fields.get('dates')),
            'has_amount': bool(fields.get('amounts') or fields.get('total')),
            'has_inv_no': bool(fields.get('invoice_number')),
            'fields_found': sum([bool(fields.get(k))
                                 for k in
['dates','amounts','invoice_number','emails']])
        }
    }
    return invoice_data
```

Section 6 — process_invoice_batch() (Lines 338–386)

This is the Mini Project function. It loops over every image in a folder, calls process_invoice() on each, collects all results, and saves one combined JSON file. This is how you process all 50 invoices.

```
def process_invoice_batch(folder_path, output_file='invoices_output.json'):
    supported_ext = ('.jpg', '.jpeg', '.png', '.tiff', '.bmp')

    # Get all supported image filenames, sorted alphabetically
    image_files = [f for f in os.listdir(folder_path)
                   if f.lower().endswith(supported_ext)]
    image_files.sort()

    all_results = []
```

```
    for i, filename in enumerate(image_files, 1):      # enumerate starts at
1
        img_path = os.path.join(folder_path, filename)
        try:
            result = process_invoice(img_path)
            all_results.append(result)
        except Exception as e:
            # Don't crash the whole batch if one invoice fails
            all_results.append({'source_file': filename, 'error': str(e)})

    # Save everything in one JSON file
    output = {
        'total_invoices': len(image_files),
        'successfully_processed': success_count,
        'invoices': all_results
    }
    with open(output_file, 'w') as f:
        json.dump(output, f, indent=2)
```

The Main Block — How to Run Everything

The main block runs automatically when you execute the script. It creates a synthetic test invoice so you can verify everything works without needing a real invoice file first.

```
# A synthetic invoice is created with cv2.putText():
img = np.ones((600, 800, 3), dtype=np.uint8) * 255   # White 800×600 image
cv2.putText(img, 'ACME SUPPLIES LTD', (80, 60), cv2.FONT_HERSHEY_SIMPLEX,
1.0, (0,0,0), 1)
# ... more lines drawn ...
cv2.imwrite('sample_invoice.jpg', img)

# Then the four steps run on this sample image:
# Step 1: basic_ocr('sample_invoice.jpg')         — raw Tesseract, no
preprocessing
# Step 2: preprocess_for_ocr('sample_invoice.jpg')  — clean image, then OCR
# Step 3: extract_invoice_fields(sample_text)      — test regex on known
string
# Step 4: process_invoice('sample_invoice.jpg')    — full pipeline end-to-end

# For Mini Project — point at your invoices folder:
# process_invoice_batch('./invoices/', 'all_invoices.json')
```

LAB 5.1

Azure Computer Vision API

Duration: 90 minutes · Target: Analyse 10 images · tags ≥ 0.85 confidence · save JSON

What This File Does

lab5_1_azure_computer_vision.py connects to Microsoft's Azure Computer Vision service and uses it to analyse images. Azure's pre-trained models return tags, captions, detected objects, and colour information — all in a single API call, no training required.

Section	Function	What it Does
Section 1	create_client()	Authenticates and returns the Azure CV client object
Section 2	analyze_image_url()	Sends an image URL to Azure; extracts tags, captions, objects, colours
Section 2b	analyze_local_image()	Same as above but reads from a local file path using a binary stream
Section 3	extract_text_from_image_url()	Uses the async Read API for OCR on images and PDFs
Section 4	run_batch_analysis()	Loops over 10 image URLs; saves all results to JSON
Section 5	evaluate_tag_accuracy()	Compares Azure tags against your manually-assigned ground truth

Configuration & Credentials (Lines 1–42)

```
AZURE_ENDPOINT = 'https://YOUR_RESOURCE_NAME.cognitiveservices.azure.com/'
AZURE_KEY      = 'YOUR_API_KEY_HERE'

# Get these from: Azure Portal → your Computer Vision resource → Keys and Endpoint
# Key 1 or Key 2 both work — they are interchangeable
# The endpoint URL always ends with a trailing slash
```

Azure Free Tier Limits (F0)

20 transactions per minute per resource
5,000 transactions per month
Max image size: 4 MB
Min image size: 50 × 50 pixels

The code adds time.sleep(0.5) between calls in the batch loop.
This keeps you safely under 20/min (0.5s gap = max 120/min, but with

> processing time included you will be well under the limit).

Section 1 — create_client() (Lines 44–56)

```
from azure.cognitiveservices.vision.computervision import
ComputerVisionClient
from msrest.authentication import CognitiveServicesCredentials

def create_client():
    # CognitiveServicesCredentials wraps the key in the format Azure expects
    client = ComputerVisionClient(
        endpoint=AZURE_ENDPOINT,
        credentials=CognitiveServicesCredentials(AZURE_KEY)
    )
    return client

# The client object holds the connection config.
# You create it once and reuse it for every API call.
```

Section 2 — analyze_image_url() (Lines 58–108)

This is the core function. It sends an image URL to Azure and receives back a rich analysis result object containing all the visual features you requested.

Requesting Features

```
from azure.cognitiveservices.vision.computervision.models import
VisualFeatureTypes

features = [
    VisualFeatureTypes.tags,         # What is in the image? Labels +
confidence
    VisualFeatureTypes.description,  # Natural language caption
    VisualFeatureTypes.objects,      # Detected objects with bounding boxes
    VisualFeatureTypes.color,        # Dominant colours, accent colour
    VisualFeatureTypes.categories,   # High-level category (animal_,
building_...)
]

result = client.analyze_image(image_url, visual_features=features)
# This is one API call — Azure processes all features simultaneously
# Requesting more features does NOT make multiple calls — still costs 1
transaction
```

Extracting Tags with Confidence Threshold

```
tags = []
for tag in result.tags:
```

```
    if tag.confidence >= 0.85:          # Only keep high-confidence tags
        tags.append({
            'name': tag.name,           # e.g. 'cat', 'animal', 'mammal'
            'confidence': round(tag.confidence, 3)  # e.g. 0.987
        })

# Why 0.85?
# Tags below 0.7 are often generic ('object', 'outdoor') and unhelpful
# 0.85 keeps specific, reliable tags and discards uncertain ones
# In production, tune this per use case — stricter for critical applications
```

Extracting Captions

```
captions = []
if result.description and result.description.captions:
    for caption in result.description.captions:
        captions.append({
            'text': caption.text,           # e.g. 'a cat sitting on a couch'
            'confidence': round(caption.confidence, 3)
        })

# result.description.captions is a list — Azure can return multiple captions
# ranked by confidence. In practice, the first one (index 0) is always best.
```

Extracting Detected Objects

```
objects = []
if result.objects:
    for obj in result.objects:
        objects.append({
            'object': obj.object_property,    # Class name e.g. 'cat',
'chair'
            'confidence': round(obj.confidence, 3),
            'bounding_box': {
                'x': obj.rectangle.x,     # Left edge in pixels
                'y': obj.rectangle.y,     # Top edge in pixels
                'w': obj.rectangle.w,     # Width in pixels
                'h': obj.rectangle.h,     # Height in pixels
            }
        })

# Note: 'objects' is more precise than 'tags'
# Tags = anything in the image (background, style, mood)
# Objects = discrete, locatable items with bounding boxes
```

Section 3 — extract_text_from_image_url() (Lines 110–148)

Azure's Read API extracts text from images asynchronously. You start the operation, then poll for the result. This is used for documents, handwriting, and dense text.

```
# Step 1: Start the async operation
```

```
read_response = client.read(url=image_url, raw=True)

# Step 2: Get the operation ID from the response header
# The 'Operation-Location' header contains a URL with the ID at the end
operation_id = read_response.headers['Operation-Location'].split('/')[-1]

# Step 3: Poll until complete
while True:
    read_result = client.get_read_result(operation_id)
    if read_result.status not in ['notStarted', 'running']:
        break
    time.sleep(1)     # Wait 1 second before polling again

# Step 4: Extract text from all pages
if read_result.status == 'succeeded':
    for page in read_result.analyze_result.read_results:
        for line in page.lines:
            print(line.text)
```

Section 4 — run_batch_analysis() (Lines 150–176)

Loops over a list of 10 image URLs, calls analyze_image_url() on each, and saves all results to a single JSON file. This is what you submit for the lab.

```
for i, url in enumerate(image_urls, 1):
    print(f'Processing image {i}/{len(image_urls)}')
    result = analyze_image_url(client, url, confidence_threshold=0.85)
    all_results.append(result)
    time.sleep(0.5)       # 0.5 second gap — respects 20 req/min free tier
limit

with open(output_file, 'w') as f:
    json.dump(all_results, f, indent=2)
```

Section 5 — evaluate_tag_accuracy() (Lines 178–222)

For 3 images, you manually specify what tags you believe are correct. The function compares these against Azure's tags and computes Precision, Recall, and F1.

```
# You fill in this dictionary manually based on what you see in each image
ground_truth = {
    0: ['cat', 'animal', 'fur', 'whiskers'],
    1: ['dog', 'animal', 'breed'],
    2: ['bicycle', 'outdoor', 'wheel'],
}

api_tags   = {t['name'].lower() for t in results[idx].get('tags', [])}
expected   = {t.lower() for t in ground_truth[idx]}

true_positives  = api_tags & expected    # Tags in BOTH sets (set
intersection)
```

```
false_positives = api_tags - expected    # Azure said yes, you said no
false_negatives = expected - api_tags    # You expected it, Azure missed it

precision = len(true_positives) / len(api_tags)   # Of Azure's tags, how many correct?
recall    = len(true_positives) / len(expected)   # Of expected tags, how many found?
f1 = 2 * precision * recall / (precision + recall)
```

What to Write in Your Evaluation Comment

For each of the 3 images, note:

1. Which tags were spot-on (expected and returned)
2. Which tags Azure added that surprised you (false positives — were they wrong?)
3. Which obvious tags Azure missed (false negatives)
4. Is the caption accurate and natural-sounding?

This written evaluation is worth marks — don't just paste numbers.

LAB 5.2

Azure Face API

Duration: 90 minutes · Target: Register 3 people · identify faces · accuracy report

What This File Does

lab5_2_azure_face_api.py uses Azure's Face API — a separate service from Computer Vision — to build a cloud-based face recognition system. You register people in a 'PersonGroup' stored on Azure's servers, train the model, then identify faces in new photos. The key difference from Lab 4.1: everything lives in the cloud, not on your machine.

Section	Function	What it Does
Section 1	create_face_client()	Authenticates and returns the Azure Face API client
Section 2	detect_faces_in_image()	Detects faces and returns attributes: age, gender, emotion, glasses
Section 3	create_person_group() / delete_person_group()	Creates or wipes the cloud person group container
Section 4	register_person() / register_sample_people()	Adds a person and their face photos to the group
Section 5	train_person_group()	Triggers model training; polls until complete
Section 6	identify_person_in_image()	Detects a face then identifies it against the person group
Section 7	verify_faces()	Checks if two specific images show the same person
Section 8	evaluate_identification_accuracy()	Tests on labelled images and reports accuracy

Configuration & Constants (Lines 1–32)

```
AZURE_FACE_ENDPOINT =
'https://YOUR_RESOURCE_NAME.cognitiveservices.azure.com/'
AZURE_FACE_KEY      = 'YOUR_FACE_API_KEY_HERE'

PERSON_GROUP_ID    = 'chitkara-lab-group'   # Must be: lowercase, no spaces
RECOGNITION_MODEL  = RecognitionModel.recognition04  # Most accurate (2021)
DETECTION_MODEL    = DetectionModel.detection03      # Best for faces-only

# IMPORTANT: RECOGNITION_MODEL must match in:
# - person_group.create()
# - face.detect_with_url() in identify step
# Mismatch → InvalidRecognitionModel error
```

Why Two Separate Models?

Detection Model: controls how faces are found in images (bounding box)
  detection_01: older, faster
  detection_03: current standard — use this

Recognition Model: controls how face embeddings are generated
  recognition_01 → 04: higher number = more accurate
  recognition_04: current best — always use this for new projects

You set Detection Model each time you call detect_with_url().
You set Recognition Model when creating the PersonGroup AND when detecting in the identify step. Both must match or identification fails.

Section 2 — detect_faces_in_image() (Lines 44–96)

Detects all faces in an image and returns their bounding boxes plus attributes. Unlike Lab 4.1 which just draws boxes, Azure Face API can also estimate age, detect emotions, and identify glasses.

```
detected_faces = client.face.detect_with_url(
    url=image_url,
    detection_model=DETECTION_MODEL,
    recognition_model=RECOGNITION_MODEL,
    return_face_id=True,          # True = get temporary FaceId (valid 24
hours)
                                  # FaceId is needed for identify() and
verify()
    return_face_attributes=[
        FaceAttributeType.age,     # Estimated age (float: e.g. 28.5)
        FaceAttributeType.gender,  # 'male' or 'female'
        FaceAttributeType.emotion, # Scores for 8 emotions (0.0 to 1.0 each)
        FaceAttributeType.glasses, # 'NoGlasses', 'ReadingGlasses',
'Sunglasses'
    ]
)
```

Extracting the Dominant Emotion

```
# result.face_attributes.emotion has properties for each emotion type
# e.g. attrs.emotion.happiness = 0.96, attrs.emotion.neutral = 0.03

emotion_dict = {
    'anger':    attrs.emotion.anger,
    'happiness':attrs.emotion.happiness,
    'neutral':  attrs.emotion.neutral,
    'sadness':  attrs.emotion.sadness,
    'surprise': attrs.emotion.surprise,
    # ... etc
}
dominant_emotion = max(emotion_dict, key=emotion_dict.get)
```

```
# max() with key=dict.get finds the key with the highest value
# e.g. if happiness=0.96 is highest → dominant_emotion = 'happiness'
```

FaceIds Expire After 24 Hours

FaceIds returned by detect_with_url() are temporary.
They expire 24 hours after creation.

This means: you CANNOT save FaceIds to a file and reuse them the next day.
Every time you want to identify a face, call detect_with_url() fresh
to get a new, valid FaceId first.

PersonIds (from registered people) are permanent — they never expire.

Sections 3 & 4 — Creating & Registering People (Lines 98–163)

The PersonGroup is a named container on Azure's servers. Inside it, you create Person records. Each Person can have multiple face photos (PersistedFaces). More photos per person = better accuracy.

```
# Step 1: Create the group container
client.person_group.create(
    person_group_id=GROUP_ID,      # Your chosen ID — unique per Azure
resource
    name='Lab 5.2 Group',          # Human-readable name
    recognition_model=RECOGNITION_MODEL
)

# Step 2: Create a Person record inside the group
person = client.person_group_person.create(
    person_group_id=GROUP_ID,
    name='Alice Smith'             # Display name
)
person_id = person.person_id      # Save this UUID — you'll need it later

# Step 3: Add face photos to the person
client.person_group_person.add_face_from_url(
    person_group_id=GROUP_ID,
    person_id=person_id,
    url='https://...alice_photo1.jpg',
    detection_model=DETECTION_MODEL
)
# Repeat add_face_from_url for each additional photo
# Recommended: 5-10 photos per person, varied angles and lighting
```

The person_id_map Dictionary

```
# We store a mapping: person_name → person_id string
person_id_map = {}
person_id_map['Alice'] = str(person.person_id)    # UUID e.g. 'a1b2c3d4-...'
```

```
# Save it to disk so you don't lose it between runs
with open('person_id_map.json', 'w') as f:
    json.dump(person_id_map, f, indent=2)

# Why do we need this?
# identify() returns a PersonId (UUID), not a name.
# We need to reverse-look up: PersonId → name
# We do this with: id_to_name = {v: k for k, v in person_id_map.items()}
```

Section 5 — train_person_group() (Lines 165–186)

Training is mandatory before identification. Azure builds internal face recognition models from all the registered photos. You must re-train whenever you add or remove photos.

```
def train_person_group(client, group_id):
    client.person_group.train(person_group_id=group_id)

    # Poll the training status — do NOT proceed until this is 'succeeded'
    while True:
        status =
client.person_group.get_training_status(person_group_id=group_id)

        if status.status == TrainingStatusType.succeeded:
            print('Training complete!')
            break
        elif status.status == TrainingStatusType.failed:
            print(f'Training failed: {status.message}')
            break

        time.sleep(1)    # 1 second between polls

# Training is fast for small groups (< 10 people) — usually 2-5 seconds
# For 1,000+ people it can take several minutes
```

Section 6 — identify_person_in_image() (Lines 188–240)

This is the two-step identification function. Step 1 detects faces in the query image to get FaceIds. Step 2 sends those FaceIds to Azure's identify endpoint, which compares them against the trained PersonGroup and returns candidate matches.

```
def identify_person_in_image(client, group_id, image_url, person_id_map):

    # STEP 1: Detect faces → get temporary FaceIds
    detected = client.face.detect_with_url(
        url=image_url,
        detection_model=DETECTION_MODEL,
        recognition_model=RECOGNITION_MODEL,
        return_face_id=True          # MUST be True — we need the FaceId
    )
    face_ids = [str(face.face_id) for face in detected]
```

```
    # STEP 2: Identify each FaceId against the PersonGroup
    results = client.face.identify(
        face_ids=face_ids,
        person_group_id=group_id,
        max_num_of_candidates_returned=1,    # Top 1 match per face
        confidence_threshold=0.5             # Min confidence — below this =
Unknown
    )

    # STEP 3: Map PersonId back to person name
    id_to_name = {v: k for k, v in person_id_map.items()}

    for result in results:
        if result.candidates:
            best  = result.candidates[0]
            name  = id_to_name.get(str(best.person_id), 'Unknown')
            conf  = best.confidence
            print(f'Identified: {name} (confidence: {conf:.3f})')
        else:
            print('Unknown person — no match above threshold')
```

How confidence_threshold Works in identify()

The threshold filters which candidate matches are returned.

confidence_threshold=0.5: only return matches where Azure is ≥50% confident
Too low → returns wrong matches; lots of false positives
Too high → returns nothing for partially-matching faces

The returned confidence score is NOT the same as a distance.
Higher confidence = more certain it is a match.
A score of 0.9+ is a very strong match.
A score of 0.5–0.6 is borderline — treat with caution.

Section 7 — verify_faces() (Lines 242–275)

Verification answers the same question as Lab 4.1's verify_two_faces(): are these two images the same person? The difference is that Azure does the comparison in the cloud, not locally.

```
# Detect face in each image to get FaceIds
face1 = client.face.detect_with_url(url=url1, ...)
face2 = client.face.detect_with_url(url=url2, ...)

# Compare the two FaceIds
result = client.face.verify_face_to_face(
    face_id1=face1[0].face_id,
    face_id2=face2[0].face_id
)
```

```
print(result.is_identical)    # True or False
print(result.confidence)      # 0.0 to 1.0

# is_identical = True when confidence exceeds Azure's internal threshold
# (approximately 0.5 for recognition_04)
```

How to Run the Full Lab in Order

Recommended Execution Sequence

1. Paste your AZURE_FACE_ENDPOINT and AZURE_FACE_KEY at the top of the file

2. Run create_face_client() to confirm your credentials work

3. Run detect_faces_in_image() on a group photo — check bounding boxes and attributes are returned correctly

4. Run create_person_group() — creates the container on Azure

5. Run register_sample_people() — registers 3 people using Microsoft's sample images. OR replace URLs with your classmates' photos

6. Run train_person_group() — wait for 'Training complete!'

7. Run identify_person_in_image() on a test photo — check name + confidence

8. Run verify_faces() on a same-person and different-person pair

9. Run evaluate_identification_accuracy() — saves lab5_2_accuracy_report.json

Outputs Generated by the Script

File	Generated By	Submit?	Contents
person_id_map.json	register_sample_people()	Yes	Maps person names to their Azure PersonIds
lab5_2_accuracy_report.json	evaluate_identification_accuracy()	Yes	Accuracy score + per-image details
Console output	All functions	Screenshot	Detection attributes, identification results

If You Don't Have an Azure Account Yet

The lab code will print a clear error message and exit safely if credentials are not set (any value containing 'YOUR_' is detected).

For the submission you can still:

1. Show the complete, commented code to demonstrate understanding
2. Include the sample Microsoft output from the Azure documentation
3. Ask your instructor — they may have a shared demo account for the class