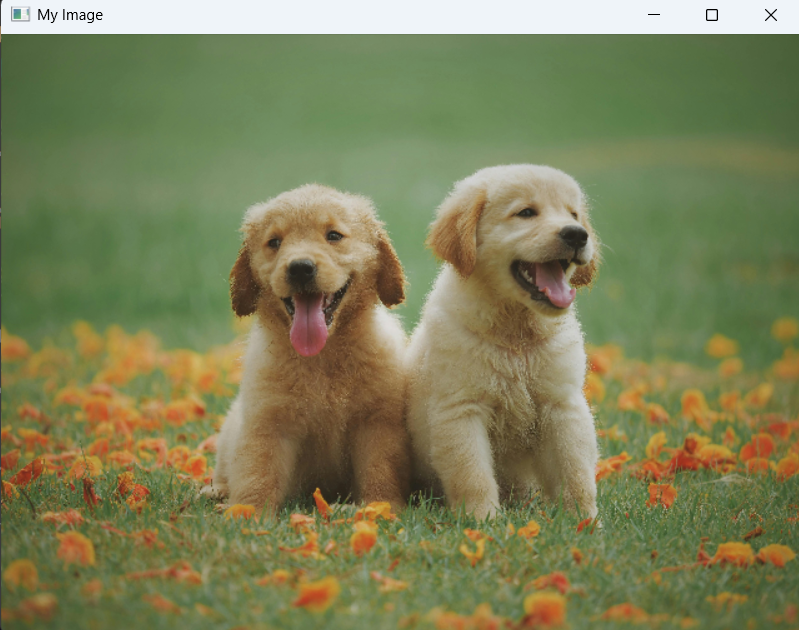
**Input image for all codes :-**

****

# Code 1

**Input:** Live video feed from the webcam.

**Output:** A real-time video display window named **“Camera Stream”** showing the webcam feed.

**Logic:**  
**Webcam live stream (real-time camera)**  
What it does, simply:

* Opens your computer’s webcam (cv2.VideoCapture(0)).
* Enters a loop that repeatedly grabs one frame from the camera (cap.read()).
* Shows each frame in a window called "Camera Stream" using cv2.imshow.
* Keeps running until you press the q key (the code checks cv2.waitKey(1) & 0xFF == ord('q')).
* When stopped, it releases the camera and closes windows.

Step-by-step logic (student style):

* Start the camera.
* If the camera returns a frame, show it; if not, break the loop (camera failed).
* After showing each frame, check quickly if q was pressed; if yes, exit loop.
* Clean up: free the camera and close GUI windows so other programs can use the camera again.

# Code 2

**Input:** Live video feed from the webcam.

**Output:** A real-time video display window showing the webcam feed, and individual frames saved as images inside a folder named **“frames”**.

**Logic:**  
What it does:

* Same as Code 1 (opens webcam and displays frames), but additionally saves each frame as an image file in a frames/ folder.

Step-by-step logic:

* Create a frames directory if it doesn’t exist.
* Start reading frames in a loop.
* For each frame: display it, save it as frames/frame\_000000.jpg, frame\_000001.jpg, etc.
* Increase a counter so every saved filename is unique.
* Stop when q is pressed, then release the camera and close windows.

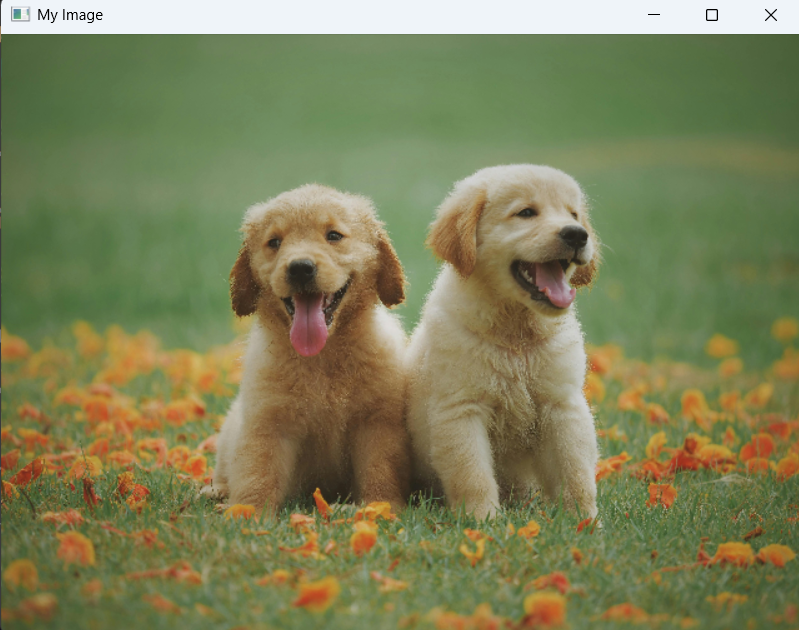
Why useful:

* Useful when you need to collect a dataset of camera images or debug what the camera saw.

# Code 3

**Input:** A single image file located at the specified file path.

**Output:** The selected image is displayed in a window named **“My Image”**.

****

**Logic:**  
What it does:

* Loads an image file from disk (cv2.imread(path)).
* If loading failed, print an error; otherwise display the image in a window.
* Wait for any key press, then close the window.

Student steps:

* cv2.imread reads the file into memory as a NumPy array (pixels).
* cv2.imshow displays it; cv2.waitKey(0) pauses until a key press.
* Finally cv2.destroyAllWindows() closes the display window so the program ends cleanly.

# Code 4

**Input:** A single image file from the specified file path.

**Output:** Four windows showing the original image, a vertically flipped version, a horizontally flipped version, and a version flipped both ways.

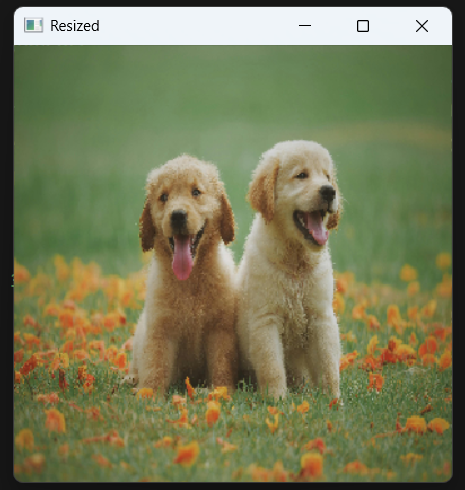
|  |  |  |
| --- | --- | --- |
|  |  |  |

**Logic:**  
This code reads an image and creates three flipped copies — one vertical, one horizontal, and one both ways. It then displays all versions side by side for comparison until a key is pressed.

# Code 5

**Input:** A single image file from the specified file path.

**Output:** Two windows showing the original image and the resized image (300×300 pixels). A resized copy is also saved as **“resized\_output.jpg”**.

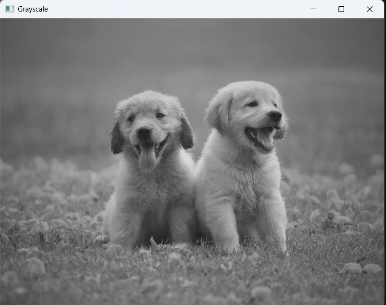
****

**Logic:**  
This code loads an image, resizes it to 300×300 pixels, displays both the original and resized versions, and saves the resized image to your computer.

# Code 6

**Input:** A single image file from the specified file path.

**Output:** Two windows showing the original image and its grayscale version. A grayscale copy is also saved as **“grayscale\_output.jpg”**.

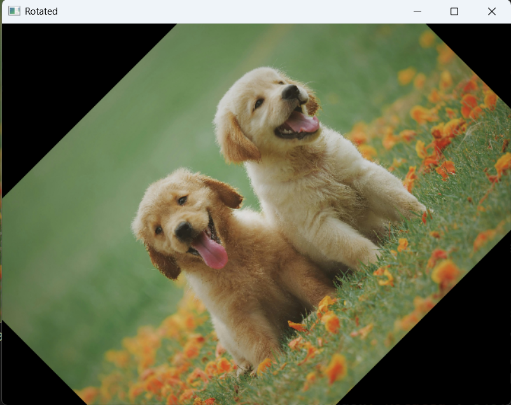
****

**Logic:**  
This code converts a colored image into a grayscale version, displays both images for comparison, and saves the grayscale output as a separate file.

# Code 7

**Input:** A single image file from the specified file path.

**Output:** Two windows showing the original image and the blurred image. The blurred version is also saved as **“blurred\_output.jpg”**.

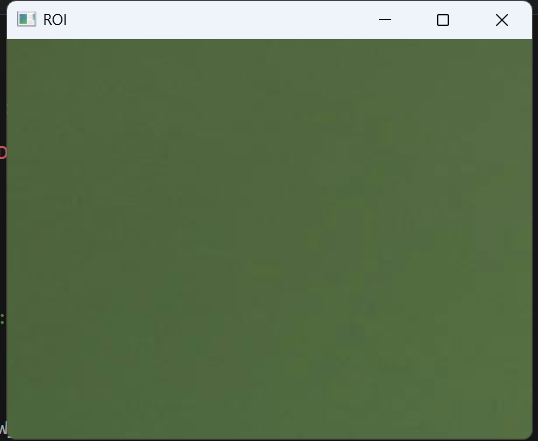
****

**Logic:**  
This code applies a **Gaussian blur** to the image using a 15×15 kernel, making the image smoother by reducing sharpness and noise. It then displays and saves the blurred image.

# Code 8

**Input:** A blank (black) image canvas created using NumPy.

**Output:** A window showing a line, rectangle, circle, and text drawn on the blank image.

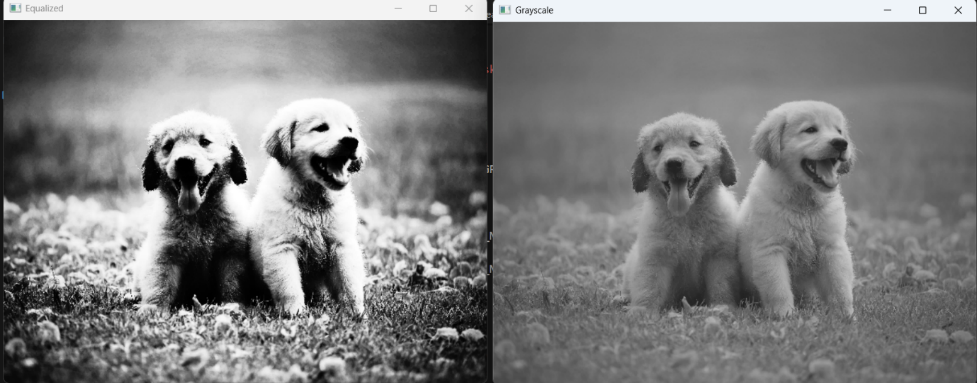
****

**Logic:**  
This code creates a black canvas and draws different geometric shapes (line, rectangle, circle) and text on it using OpenCV drawing functions.

# Code 9

**Input:** A grayscale image file from the specified file path.

**Output:** Two windows — one showing the original grayscale image and the other showing its binary (black and white) version.

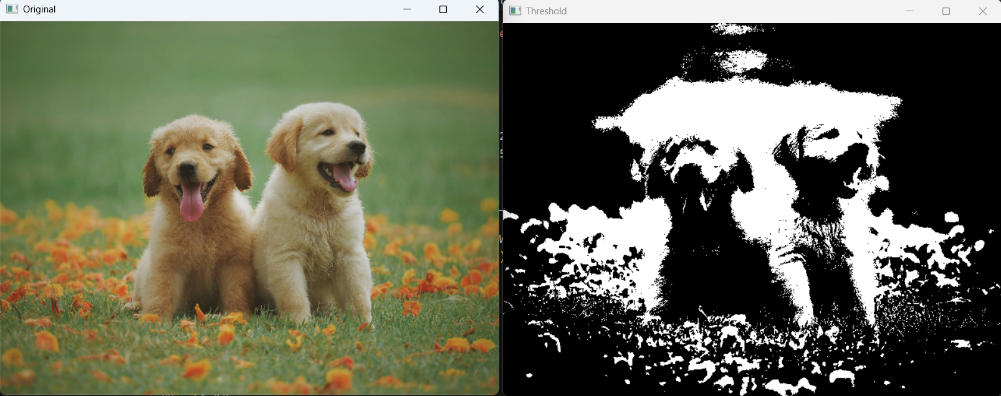
****

**Logic:**  
This code converts the grayscale image into a binary image using a threshold value. Pixels brighter than 127 become white, and darker ones become black.

# Code 10

**Input:** A grayscale image file from the specified file path.

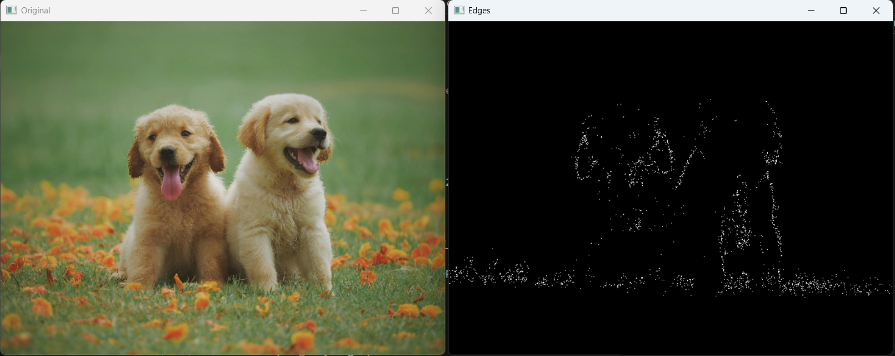
**Output:** A window showing only the detected edges of the image.

****

**Logic:**  
This code performs **edge detection** using the Canny algorithm. It highlights the edges and boundaries present in the image for feature extraction or analysis.

# Code 11

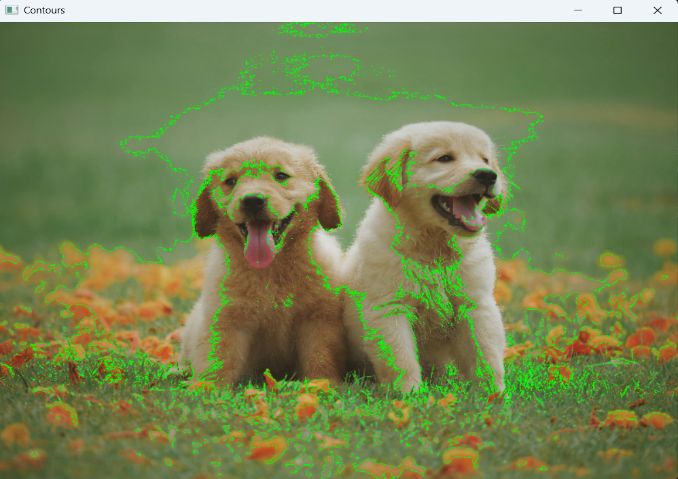
**Input:** A color image file (e.g., face.jpg) containing one or more human faces.  
**Output:** A window displaying the original image with rectangles drawn around each detected face.

****

**Logic:** Converts the image to grayscale, uses a Haar Cascade pre-trained classifier to detect faces with detectMultiScale, then draws bounding rectangles around detected face regions.

# Code 12

**Input:** A color or grayscale image containing distinct shapes (e.g., shapes.png).  
**Output:** A window showing the image with contours of detected shapes overlaid.

****

**Logic:** Converts to grayscale, thresholds to a binary image, finds contours with findContours, and draws them to visualize shape boundaries and hierarchies.

# Code 13

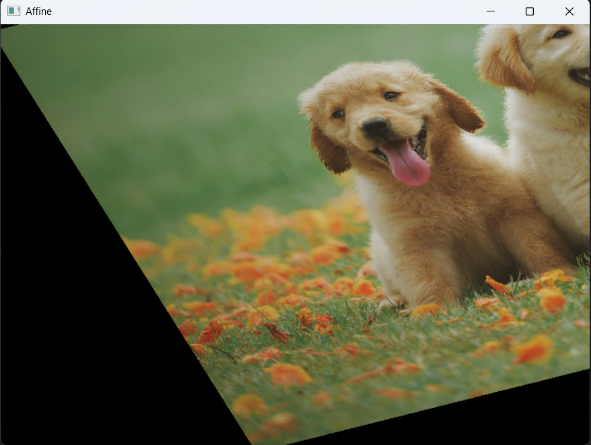
**Input:** A color image containing objects with the target color (e.g., a blue ball in ball.jpg).  
**Output**: Three windows — the original image, the binary mask highlighting pixels in the color range, and the filtered result showing only the masked regions.

****

**Logic:** Converts the image to HSV space (more robust for color filtering), thresholds to a color range with inRange to make a mask, then applies the mask with bitwise\_and to extract only regions matching the color.

# Code 14

**Input:** A color image (e.g., person.jpg) and an initial rectangle defining the approximate foreground ROI.  
**Output:** Windows showing the original image and the image with the extracted foreground (background suppressed).

****

**Logic:** Uses the GrabCut algorithm to iteratively estimate foreground and background. The initial rectangle initializes the model; after run, a refined mask isolates foreground pixels which are multiplied with the original image to produce the extracted subject.

# Code 15

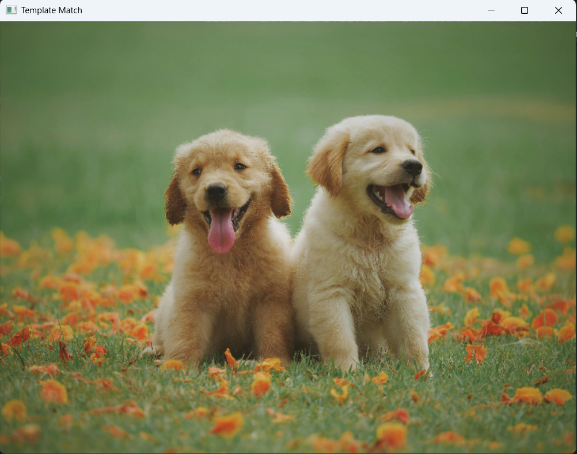
**Input:** Live webcam feed (default camera index 0).  
**Output:** Real-time windows showing the original frame, a binary mask of pixels within the target color range, and the filtered/tracked output showing only the detected color regions.

****

**Logic:** Captures frames in a loop, converts to HSV for color robustness, applies a color range mask with inRange, uses bitwise\_and to display only masked pixels — useful for simple color-based object tracking. Press q to exit.

# Code 16

**Input:** A grayscale image with text or high-contrast shapes (e.g., text.png).  
**Output:** Windows showing the binary inverted image, its erosion result, and its dilation result.

****

**Logic:** Applies binary inverse thresholding to invert foreground/background, then demonstrates morphological operations: erosion (shrinks foreground — removes small noise or thins objects) and dilation (expands foreground — fills gaps and strengthens shapes). Useful for preprocessing before OCR or contour detection.

# File 1

# file: 1\_preprocess.py

import re

from typing import List

from sklearn.feature\_extraction.text import ENGLISH\_STOP\_WORDS

import spacy

nlp = spacy.load("en\_core\_web\_sm") # small, fast English model

def basic\_clean(text: str) -> str:

# lower, strip urls/emails/@mentions/hashtags, keep letters/numbers/space/apostrophe

text = text.lower()

text = re.sub(r"(http\S+|www\.\S+)", " ", text)

text = re.sub(r"\S+@\S+", " ", text)

text = re.sub(r"[@#]\w+", " ", text)

text = re.sub(r"[^a-z0-9\s']", " ", text)

text = re.sub(r"\s+", " ", text).strip()

return text

def tokenize\_stop\_lemma(text: str) -> List[str]:

doc = nlp(text)

out = []

for tok in doc:

if tok.is\_space or tok.is\_punct:

continue

lemma = tok.lemma\_.lower().strip()

if len(lemma) < 3: # drop very short tokens

continue

if lemma in ENGLISH\_STOP\_WORDS: # sklearn's built-in stoplist

continue

out.append(lemma)

return out

def preprocess(text: str) -> List[str]:

return tokenize\_stop\_lemma(basic\_clean(text))

if \_\_name\_\_ == "\_\_main\_\_":

s = "Emails like help@site.com are filtered. I’m LOVING NLP!!! Visit https://x.y."

print(preprocess(s))

**Input:** Raw text containing emails, URLs, mentions, hashtags, punctuation, uppercase letters, and stopwords.

**Output:** A list of cleaned, tokenized, lemmatized words with stopwords and very short tokens removed.

**Logic:**  
This code performs textual preprocessing in multiple steps. First, it converts all text to lowercase for uniformity. It removes URLs, email addresses, mentions (e.g., @username), and hashtags since these often do not contribute meaningfully to NLP tasks. It retains only letters, numbers, spaces, and apostrophes to simplify the text. Next, it tokenizes the cleaned text using spaCy, lemmatizes each token to reduce words to their base form, and removes stopwords (common words like “the” or “and” that add little semantic value). Very short words are also removed to avoid noise. The result is a list of informative, normalized tokens ready for downstream NLP tasks like classification or topic modeling.

# File 2

# file: 2\_classify\_tfidf.py

from sklearn.model\_selection import train\_test\_split

from sklearn.pipeline import Pipeline

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report, confusion\_matrix

import numpy as np

texts = [

"I loved this movie, fantastic acting and great story",

"This film was terrible and boring",

"Absolutely wonderful experience, highly recommend",

"Worst acting ever, do not watch",

"It was okay, some parts were fun",

"I hated the plot, very disappointing",

"Brilliant direction and superb cast",

"Not good, waste of time",

"Enjoyable and engaging from start to finish",

"Awful soundtrack and weak story"

]

labels = np.array([1,0,1,0,1,0,1,0,1,0]) # 1=pos, 0=neg

X\_train, X\_test, y\_train, y\_test = train\_test\_split(texts, labels, test\_size=0.3, random\_state=42, stratify=labels)

pipe = Pipeline([

("tfidf", TfidfVectorizer(ngram\_range=(1,2), min\_df=1)),

("clf", LogisticRegression(max\_iter=1000))

])

pipe.fit(X\_train, y\_train)

y\_pred = pipe.predict(X\_test)

print("Classification report:\n", classification\_report(y\_test, y\_pred, digits=4))

print("Confusion matrix:\n", confusion\_matrix(y\_test, y\_pred))

samples = ["pretty good but slow in places", "utterly awful, I want my time back"]

print("Predictions:", pipe.predict(samples))

print("Class probabilities:", pipe.predict\_proba(samples))

**Input:** Short text samples with labeled sentiment (positive or negative).

**Output:** Predicted sentiment labels for test samples and probabilities for each class.

**Logic:**  
This code demonstrates a basic text classification pipeline using TF-IDF and logistic regression. First, the dataset is split into training and testing sets, maintaining class balance using stratification. The Pipeline combines TF-IDF vectorization (converting raw text into weighted numerical features) and logistic regression for classification. The model is trained on the training data and evaluated on the test data using classification metrics and a confusion matrix. Finally, sample texts are fed to the trained model for prediction, producing both class labels and probability scores. TF-IDF captures the importance of words in context, while logistic regression learns the decision boundary between positive and negative sentiments. The pipeline ensures reproducibility and easy experimentation.

# File 3

# file: 3\_tune\_grid.py

from sklearn.model\_selection import GridSearchCV

from sklearn.pipeline import Pipeline

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

import numpy as np

texts = [

"excellent movie with great acting",

"terrible plot and awful pacing",

"loved every moment, fantastic!",

"boring and predictable",

"superb cinematography and direction",

"weak script and bad acting",

"what a masterpiece",

"not good at all",

"brilliant experience overall",

"do not recommend"

]

y = np.array([1,0,1,0,1,0,1,0,1,0])

pipe = Pipeline([

("tfidf", TfidfVectorizer()),

("clf", LogisticRegression(max\_iter=1000))

])

param\_grid = {

"tfidf\_\_ngram\_range": [(1,1),(1,2)],

"tfidf\_\_min\_df": [1,2],

"tfidf\_\_analyzer": ["word", "char\_wb"],

"clf\_\_C": [0.25, 1.0, 4.0]

}

search = GridSearchCV(pipe, param\_grid, cv=3, n\_jobs=-1, scoring="f1")

search.fit(texts, y)

print("Best params:", search.best\_params\_)

print("Best CV score (f1):", search.best\_score\_)

best\_model = search.best\_estimator\_

print("Sample prediction:", best\_model.predict(["not a great movie but had moments"]))

**Input:** Text dataset with positive and negative labels, and a set of hyperparameters to tune.

**Output:** Best hyperparameter combination, cross-validation F1 score, and a prediction for a sample text.

**Logic:**  
This code uses grid search to find the optimal hyperparameters for a text classification pipeline combining TF-IDF vectorization and logistic regression. The parameters tuned include n-gram range (single words or bigrams), minimum document frequency, analyzer type (word-level or character-level), and regularization strength C of logistic regression. GridSearchCV performs exhaustive search over all parameter combinations using cross-validation to evaluate performance (F1 score). The best combination is selected, and the trained model can then be used for prediction. This approach ensures that the model is not only trained well but also optimally configured for the dataset.

# File 4

# file: 4\_tfidf\_demo.py

from sklearn.feature\_extraction.text import TfidfVectorizer

import pandas as pd

docs = [

"machine learning is fun",

"deep learning advances machine intelligence",

"artificial intelligence and machine learning"

]

tfidf = TfidfVectorizer()

X = tfidf.fit\_transform(docs)

tfidf\_df = pd.DataFrame(X.toarray(), columns=tfidf.get\_feature\_names\_out())

print("Vocabulary:", tfidf.get\_feature\_names\_out())

print("\nTF-IDF Matrix:")

print(tfidf\_df.round(3))

**Input:** A small collection of text documents.

**Output:** Vocabulary of terms and the TF-IDF weighted matrix as a DataFrame.

**Logic:**  
This code demonstrates how TF-IDF transforms raw text into a numerical representation. TfidfVectorizer computes term frequency-inverse document frequency for each term across the corpus. This gives higher weight to words that are frequent in a document but rare across the corpus, highlighting important keywords. The resulting matrix is converted into a Pandas DataFrame for easy inspection. This representation can be fed into machine learning models for classification, clustering, or similarity computation.

# File 5

# file: 5\_spacy\_ner\_pos.py

import spacy

from pprint import pprint

nlp = spacy.load("en\_core\_web\_sm")

text = ("Apple is opening a new office in Bengaluru next quarter. "

"Tim Cook met Karnataka officials on September 3, 2025 to discuss expansion.")

doc = nlp(text)

print("\nNamed Entities (text, label):")

for ent in doc.ents:

print(f"{ent.text:<25} -> {ent.label\_}")

print("\nPart-of-Speech & Lemmas:")

for token in doc:

if not token.is\_space:

print(f"{token.text:<15} POS={token.pos\_:<5} Lemma={token.lemma\_}")

print("\nNoun chunks (base NPs):")

pprint([chunk.text for chunk in doc.noun\_chunks])

**Input:** A raw text containing names, dates, and locations.

**Output:** Named entities with labels, POS tags with lemmas, and noun chunks.

**Logic:**  
This code uses spaCy to analyze text linguistically. Named Entity Recognition (NER) identifies real-world entities such as companies, locations, and dates. Part-of-speech tagging assigns grammatical roles to each token and computes lemmas to reduce words to their base form. Noun chunks extract base noun phrases that can represent key concepts or subjects. Together, these analyses allow structured information extraction and provide rich features for downstream NLP tasks like relation extraction or information retrieval.

# File 6

# file: 6\_nb\_classify.py

from sklearn.model\_selection import train\_test\_split

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import classification\_report

texts = [

"I love this movie",

"This film was awful",

"Amazing performance and great story",

"Boring and too long",

"Fantastic acting",

"Terrible direction"

]

labels = [1, 0, 1, 0, 1, 0]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(texts, labels, test\_size=0.3, random\_state=42)

vectorizer = CountVectorizer()

X\_train\_bow = vectorizer.fit\_transform(X\_train)

X\_test\_bow = vectorizer.transform(X\_test)

clf = MultinomialNB()

clf.fit(X\_train\_bow, y\_train)

y\_pred = clf.predict(X\_test\_bow)

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

**Input:** A small sentiment-labeled text dataset.

**Output:** Classification report including precision, recall, and F1-score.

**Logic:**  
This code illustrates a basic Naive Bayes classification pipeline. The raw text is transformed into a bag-of-words representation, capturing the frequency of each word in the dataset. MultinomialNB uses these counts to compute the probability of each class given the words in a document. The dataset is split into training and test sets to evaluate model performance. Naive Bayes is particularly effective for text data because it assumes word independence and handles high-dimensional sparse features efficiently.

# File 7

# file: 7\_similarity\_demo.py

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

docs = [

"I love machine learning and NLP",

"NLP and machine learning are amazing",

"Cooking recipes are fun to try",

]

vectorizer = TfidfVectorizer()

X = vectorizer.fit\_transform(docs)

sim\_matrix = cosine\_similarity(X)

print("Cosine Similarity Matrix:\n", sim\_matrix)

**Input:** A small collection of text documents.

**Output:** Cosine similarity matrix showing pairwise similarity scores between documents.

**Logic:**  
The code converts text into TF-IDF vectors to quantify the importance of words. Cosine similarity measures the angle between vectors in high-dimensional space, giving a score between 0 and 1. A higher score indicates more similar content. This method is commonly used in information retrieval, recommendation systems, and clustering to find semantically similar documents.

# File 8

# file: 8\_topic\_modeling.py

from gensim import corpora, models

docs = [

"I love deep learning and natural language processing",

"Artificial intelligence is the future",

"Cooking and baking are my hobbies",

"I enjoy trying new recipes in the kitchen",

"Machine learning and AI are closely related"

]

texts = [doc.lower().split() for doc in docs]

dictionary = corpora.Dictionary(texts)

corpus = [dictionary.doc2bow(text) for text in texts]

lda\_model = models.LdaModel(corpus, num\_topics=2, id2word=dictionary, passes=10)

for idx, topic in lda\_model.print\_topics(-1):

print(f"Topic {idx}: {topic}")

**Input:** A set of text documents.

**Output:** Topics discovered by the LDA model with associated keywords.

**Logic:**  
This code performs topic modeling using Latent Dirichlet Allocation (LDA). Each document is tokenized, and a dictionary mapping terms to IDs is created. The corpus is represented as a bag-of-words for LDA. LDA assumes documents are mixtures of topics, and topics are distributions over words. The model learns these distributions iteratively (passes=10). The output topics summarize the main themes in the dataset, useful for document clustering, content summarization, or exploratory text analysis.