BUILD YOUR OWN AI

23 Complete Programs from Beginner to Advanced

A copy, run, understand, modify, and build guide for practical AI programming — including Hugging Face, transformers, embeddings, RAG, agents, APIs, Docker, evaluation, and production-style workflows.

FULL PROGRAMS • LARGE CODE • EXPLANATIONS • TESTS • CHALLENGES

KYAMBADDE GRRALD

FIRST EDITION • 2026 • UPDATED WITH A DEDICATED HUGGING FACE PROJECT

Copyright & Responsible-Use Notice

Copyright © 2026 Kyambadde Grrald. All rights reserved. No part of this book may be reproduced or distributed without permission, except for brief quotations used in reviews or other uses permitted by law.

Software notice: Python packages, APIs, model names, hardware requirements, and user interfaces change over time. Code in this book is educational and may need small adjustments to work with future releases. Record the versions you use and consult the current official documentation.

Responsible-use notice: Use AI systems, data, APIs, documents, images, microphones, and security tools only where you have permission. Security examples in this book are defensive and intended for authorized environments.

This book is educational material and is not a substitute for professional advice.

How to Use This Book

This edition is deliberately different from a theory-heavy AI textbook. Each project contains a complete runnable program, not a short fragment that leaves the main work for you.

The learning loop

1. Read the goal. 2. Create the folder. 3. Install the dependencies. 4. Copy the complete program. 5. Run it unchanged. 6. Read the output. 7. Trace the important functions. 8. Change one thing. 9. Test again.

Why complete programs matter

A small code fragment can demonstrate syntax, but a complete program teaches structure: imports, configuration, functions, error handling, input, processing, output, and the main execution flow. You will see how the pieces connect rather than memorizing isolated lines.

Code size

The code listings use a larger monospaced type than the previous edition and are allowed to continue across pages. Do not shrink the PDF when reading the code; use the PDF viewer's zoom if you are reading on screen.

Beta Reader

I would like to thank the following people for taking the time to read this book, carefully review its content, and provide valuable feedback:

— Faisal Sultan Alshamsi — for his expertise in managing security appliances, including Palo Alto Networks, FortiGate firewalls, and F5 BIG-IP LTM load balancers, as well as applying Access Control Lists (ACLs), creating security policies, blocking malicious IP addresses, and managing intrusion threats at the General Civil Aviation Authority (GCAA).

— Joseph Makasi Wasike — AVSEC Professional Manager, General Civil Aviation Authority (GCAA).

Thanks all for reading the book.

CONTENTS — 23 COMPLETE PROGRAMS

01 Build Your AI Workshop
02 Clean Real-World Data
03 Train Your First Classifier
04 Build a Spam Detector
05 Build a Price Predictor
06 Build a Recommendation Engine
07 Segment Customers
08 Train a Neural Network
09 Use a Transformer
10 Build Semantic Search
11 Build a Local RAG Assistant
12 Expose AI Through FastAPI
13 Build an AI Web App
14 Build a Gradio Interface
15 Build a Voice Assistant
16 Build a Safe AI Agent
17 Build an Evaluation Harness
18 Final Knowledge Assistant
19 Track AI Experiments
20 Dockerize an AI API
21 Secure an AI Project
22 Build Production-Style RAG
23 Build a Hugging Face Transformer Classifier

The projects progress from Python fundamentals to modern transformer, RAG, deployment, evaluation, security, and Hugging Face workflows.

PROJECT 01

Build Your AI Workshop

Goal: Create a clean Python environment and verify the tools you will use throughout the book.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

Python 3.10+; no extra package is required for the basic checks. Install NumPy, pandas, scikit-learn, and PyTorch to run the full doctor.

RUN

```
python 01_ai_workshop.py
```

PROJECT FLOW

Step	What happens
1	Checks the Python version
2	Imports each core package dynamically
3	Prints a clear status for every dependency
4	Counts how many components are ready

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 01 — COMPLETE PROGRAM

File: 01_ai_workshop.py Lines: 39

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 1
AI Workshop Doctor
Run: python 01_ai_workshop.py
"""
import importlib
import platform
import sys

PACKAGES = ["numpy", "pandas", "sklearn", "torch"]

def check_python():
    print("Python:", platform.python_version())
    if sys.version_info < (3, 10):
        print("WARNING: Python 3.10+ is recommended.")
    else:
        print("OK: Python version is supported.")

def check_package(name):
    try:
        module = importlib.import_module(name)
        version = getattr(module, "__version__", "installed")
        print(f"OK: {name} ({version})")
        return True
    except Exception as exc:
        print(f"MISSING: {name} -> {exc}")
        return False

def main():
    print("=" * 60)
    print("BUILD YOUR OWN AI — WORKSHOP DOCTOR")
    print("=" * 60)
    check_python()
    results = [check_package(name) for name in PACKAGES]
    print("\nReady:", f"{sum(results)}/{len(results)} core packages availab
    print("Next: create a virtual environment and install missing packages.

if __name__ == "__main__":
    main()
```

PROJECT 01 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	importlib, platform, sys
Functions	check_python, check_package, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Checks the Python version
2. Imports each core package dynamically
3. Prints a clear status for every dependency
4. Counts how many components are ready

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Break one dependency on purpose, diagnose the error, and repair the environment.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the

current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 02

Clean Real-World Data

Goal: Build a CSV cleaner that creates sample data, detects missing values, repairs types, and saves a clean dataset.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install pandas
```

RUN

```
python 02_csv_cleaner.py
```

PROJECT FLOW

Step	What happens
1	Creates a starter CSV only when one does not exist
2	Loads the CSV into a DataFrame
3	Converts messy values into numeric types
4	Fills missing values with explicit rules
5	Saves a clean CSV

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 02 — COMPLETE PROGRAM

File: 02_csv_cleaner.py Lines: 53

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 2
Self-contained CSV cleaner.
Run: python 02_csv_cleaner.py
"""
from pathlib import Path
import pandas as pd

INPUT = Path("customers_raw.csv")
OUTPUT = Path("customers_clean.csv")

SAMPLE = """name,age,country,spent
Amina,24,Uganda,120.50
John,,UAE,210
Sarah,31,,175.25
David,29,Uganda,
Musa,not_available,UAE,95
"""

def create_sample_if_missing():
    if not INPUT.exists():
        INPUT.write_text(SAMPLE, encoding="utf-8")
        print(f"Created sample data: {INPUT}")

def clean_data(df):
    df = df.copy()
    df["name"] = df["name"].fillna("Unknown").astype(str).str.strip()
    df["country"] = df["country"].fillna("Unknown").astype(str).str.strip()
    df["age"] = pd.to_numeric(df["age"], errors="coerce")
    df["spent"] = pd.to_numeric(df["spent"], errors="coerce")
    df["age"] = df["age"].fillna(df["age"].median())
    df["spent"] = df["spent"].fillna(0)
    df["age"] = df["age"].round().astype(int)
    df["spent"] = df["spent"].round(2)
    return df

def main():
    create_sample_if_missing()
    df = pd.read_csv(INPUT)
    print("\nRAW DATA")
```

```
    print(df.to_string(index=False))
    print("\nMISSING VALUES")
    print(df.isna().sum().to_string())

    clean = clean_data(df)
    clean.to_csv(OUTPUT, index=False)

    print("\nCLEAN DATA")
    print(clean.to_string(index=False))
    print(f"\nSaved: {OUTPUT.resolve()}")

if __name__ == "__main__":
    main()
```

PROJECT 02 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, pandas
Functions	create_sample_if_missing, clean_data, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Creates a starter CSV only when one does not exist
2. Loads the CSV into a DataFrame
3. Converts messy values into numeric types
4. Fills missing values with explicit rules
5. Saves a clean CSV

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a new column and a validation rule that rejects impossible ages.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 03

Train Your First Classifier

Goal: Train, evaluate, save, and reuse a Random Forest classifier.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn joblib
```

RUN

```
python 03_iris_classifier.py
```

PROJECT FLOW

Step	What happens
1	Loads a built-in labeled dataset
2	Splits data into training and test sets
3	Trains a Random Forest
4	Measures accuracy and prints a report
5	Saves the trained model
6	Runs one new prediction

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 03 — COMPLETE PROGRAM

File: 03_iris_classifier.py Lines: 55

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 3
Complete Iris classifier.
Run: python 03_iris_classifier.py
"""
from pathlib import Path
import joblib
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusic
from sklearn.model_selection import train_test_split

MODEL_FILE = Path("iris_model.joblib")

def train():
    data = load_iris()
    X_train, X_test, y_train, y_test = train_test_split(
        data.data, data.target, test_size=0.20, random_state=42,
        stratify=data.target
    )

    model = RandomForestClassifier(
        n_estimators=200, random_state=42
    )
    model.fit(X_train, y_train)

    predictions = model.predict(X_test)
    print("Accuracy:", round(accuracy_score(y_test, predictions), 4))
    print("\nClassification report:")
    print(classification_report(y_test, predictions,
                                target_names=data.target_names))
    print("Confusion matrix:")
    print(confusion_matrix(y_test, predictions))

    joblib.dump(
        {"model": model, "target_names": data.target_names.tolist()},
        MODEL_FILE
    )
    print(f"\nModel saved to {MODEL_FILE}")
    return model, data.target_names
```

```
def predict_one(model, target_names):
    sample = [[5.1, 3.5, 1.4, 0.2]]
    prediction = model.predict(sample)[0]
    probabilities = model.predict_proba(sample)[0]
    print("\nNew flower:", sample[0])
    print("Prediction:", target_names[prediction])
    print("Confidence:", round(float(probabilities[prediction]), 4))

def main():
    model, names = train()
    predict_one(model, names)

if __name__ == "__main__":
    main()
```

PROJECT 03 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, joblib, sklearn.datasets, sklearn.ensemble, sklearn.metrics, sklearn.model_selection
Functions	train, predict_one, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads a built-in labeled dataset
2. Splits data into training and test sets
3. Trains a Random Forest
4. Measures accuracy and prints a report
5. Saves the trained model
6. Runs one new prediction

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Change the forest size and compare accuracy and training time.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program

runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 04

Build a Spam Detector

Goal: Turn text into TF-IDF features, train a classifier, evaluate it, save it, and test new messages interactively.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn joblib
```

RUN

```
python 04_spam_detector.py
```

PROJECT FLOW

Step	What happens
1	Stores labeled training examples
2	Converts messages into TF-IDF features
3	Trains logistic regression
4	Evaluates the held-out test set
5	Saves the complete pipeline
6	Accepts interactive predictions

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 04 — COMPLETE PROGRAM

File: 04_spam_detector.py Lines: 75

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 4
Complete spam detector with a self-contained starter dataset.
Run: python 04_spam_detector.py
"""
from pathlib import Path
import joblib
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

MODEL_FILE = Path("spam_detector.joblib")

MESSAGES = [
    ("Win a free phone now, claim your prize!", "spam"),
    ("Congratulations, you have won a cash reward", "spam"),
    ("Click here to receive your bonus today", "spam"),
    ("Urgent offer, claim your free gift", "spam"),
    ("You were selected for a special promotion", "spam"),
    ("Limited time prize waiting for you", "spam"),
    ("Free voucher available, click now", "spam"),
    ("You have been chosen for a reward", "spam"),
    ("Claim your money immediately", "spam"),
    ("Exclusive deal, act now to win", "spam"),
    ("Can you send me the meeting notes?", "normal"),
    ("I will arrive at the office at nine", "normal"),
    ("Please call me when you are free", "normal"),
    ("The project report is attached", "normal"),
    ("Let's meet for coffee tomorrow", "normal"),
    ("Your appointment is confirmed for Friday", "normal"),
    ("I finished the task and pushed the changes", "normal"),
    ("Can you review this document?", "normal"),
    ("Dinner is ready when you get home", "normal"),
    ("The package arrived this afternoon", "normal"),
    ("Please confirm the delivery address", "normal"),
    ("I sent the invoice this morning", "normal"),
    ("The training session starts at ten", "normal"),
    ("Are we still meeting today?", "normal"),
```

```
]

def build_model():
    return Pipeline([
        ("tfidf", TfidfVectorizer(lowercase=True, ngram_range=(1, 2))),
        ("classifier", LogisticRegression(max_iter=2000))
    ])

def main():
    texts = [x[0] for x in MESSAGES]
    labels = [x[1] for x in MESSAGES]

    X_train, X_test, y_train, y_test = train_test_split(
        texts, labels, test_size=0.25, random_state=42, stratify=labels
    )

    model = build_model()
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    print(classification_report(y_test, predictions))
    joblib.dump(model, MODEL_FILE)
    print("Saved:", MODEL_FILE)

    print("\nInteractive test. Press Enter on an empty line to quit.")
    while True:
        message = input("Message: ").strip()
        if not message:
            break
        label = model.predict([message])[0]
        probability = max(model.predict_proba([message])[0])
        print(f"Prediction: {label} | confidence={probability:.3f}")

if __name__ == "__main__":
    main()
```

PROJECT 04 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, joblib, sklearn.feature_extraction.text, sklearn.linear_model, sklearn.metrics, sklearn.model_selection, sklearn.pipeline
Functions	build_model, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Stores labeled training examples
2. Converts messages into TF-IDF features
3. Trains logistic regression
4. Evaluates the held-out test set
5. Saves the complete pipeline
6. Accepts interactive predictions

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add at least 20 new labeled messages and measure false positives.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program

runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 05

Build a Price Predictor

Goal: Train a regression model on generated housing-style data and interpret MAE and R^2.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install numpy scikit-learn joblib
```

RUN

```
python 05_price_predictor.py
```

PROJECT FLOW

Step	What happens
1	Generates a controlled synthetic dataset
2	Splits features and target values
3	Trains a regression forest
4	Measures MAE and R^2
5	Saves the model
6	Predicts a new example

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 05 — COMPLETE PROGRAM

File: 05_price_predictor.py Lines: 57

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 5
Self-contained house-price regression demo.
Run: python 05_price_predictor.py
"""
import joblib
import numpy as np
from pathlib import Path
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split

MODEL_FILE = Path("price_model.joblib")

def make_dataset():
    rng = np.random.default_rng(42)
    area = rng.integers(40, 300, 300)
    bedrooms = rng.integers(1, 6, 300)
    age = rng.integers(0, 35, 300)
    distance = rng.uniform(0.5, 25, 300)
    noise = rng.normal(0, 18000, 300)

    price = (
        35000
        + area * 1800
        + bedrooms * 25000
        - age * 2200
        - distance * 3500
        + noise
    )

    X = np.column_stack([area, bedrooms, age, distance])
    return X, price

def main():
    X, y = make_dataset()
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )
```

```
    model = RandomForestRegressor(
        n_estimators=300, random_state=42
    )
    model.fit(X_train, y_train)

    predictions = model.predict(X_test)
    print("MAE:", round(mean_absolute_error(y_test, predictions), 2))
    print("R2 :", round(r2_score(y_test, predictions), 4))

    joblib.dump(model, MODEL_FILE)
    example = [[120, 3, 5, 4.0]]
    print("\nExample house:", example[0])
    print("Predicted price:", round(float(model.predict(example)[0]), 2))
    print("Saved:", MODEL_FILE)

if __name__ == "__main__":
    main()
```

PROJECT 05 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	joblib, numpy, pathlib, sklearn.ensemble, sklearn.metrics, sklearn.model_selection
Functions	make_dataset, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Generates a controlled synthetic dataset
2. Splits features and target values
3. Trains a regression forest
4. Measures MAE and R^2
5. Saves the model
6. Predicts a new example

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a feature and see whether the test error changes.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program

runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 06

Build a Recommendation Engine

Goal: Recommend similar items using TF-IDF and cosine similarity.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn
```

RUN

```
python 06_recommender.py
```

PROJECT FLOW

Step	What happens
1	Stores items with descriptive text
2	Vectorizes descriptions with TF-IDF
3	Computes cosine similarity
4	Ranks neighbors
5	Returns the requested number of recommendations

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 06 — COMPLETE PROGRAM

File: 06_recommender.py Lines: 54

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 6
Complete content-based recommender.
Run: python 06_recommender.py
"""
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

ITEMS = [
    ("Spanish Latte", "espresso milk sweet creamy coffee"),
    ("Cappuccino", "espresso milk foam balanced coffee"),
    ("Flat White", "espresso silky milk strong coffee"),
    ("Caramel Latte", "espresso milk caramel sweet creamy"),
    ("Pistachio Latte", "espresso milk pistachio nutty sweet"),
    ("Dark Mocha", "espresso chocolate cocoa rich sweet"),
    ("Americano", "espresso water bold black coffee"),
    ("Cold Drip", "cold coffee slow brewed smooth"),
]

def recommend(title, count=3):
    names = [x[0] for x in ITEMS]
    descriptions = [x[1] for x in ITEMS]
    matrix = TfidfVectorizer().fit_transform(descriptions)
    scores = cosine_similarity(matrix)

    if title not in names:
        raise ValueError(f"Unknown item. Choose from: {', '.join(names)}")

    index = names.index(title)
    ranked = scores[index].argsort()[::-1]
    results = []
    for i in ranked:
        if i != index:
            results.append((names[i], float(scores[index, i])))
        if len(results) == count:
            break
    return results

def main():
    print("Available items:")
```

```
    for name, _ in ITEMS:
        print(" -", name)

    title = input("\nWhat do you like? ").strip()
    try:
        results = recommend(title)
        print("\nRecommendations:")
        for name, score in results:
            print(f"{name:20} similarity={score:.3f}")
    except ValueError as exc:
        print("Error:", exc)

if __name__ == "__main__":
    main()
```

PROJECT 06 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	sklearn.feature_extraction.text, sklearn.metrics.pairwise
Functions	recommend, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Stores items with descriptive text
2. Vectorizes descriptions with TF-IDF
3. Computes cosine similarity
4. Ranks neighbors
5. Returns the requested number of recommendations

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add 10 items and a category filter.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 07

Segment Customers

Goal: Use KMeans and silhouette scores to discover groups without labels.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install numpy pandas scikit-learn
```

RUN

```
python 07_customer_segments.py
```

PROJECT FLOW

Step	What happens
1	Creates synthetic customer behavior
2	Standardizes the features
3	Tries several cluster counts
4	Scores clusters with silhouette score
5	Keeps the selected model
6	Prints segment averages

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 07 — COMPLETE PROGRAM

File: 07_customer_segments.py Lines: 45

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 7
Complete customer segmentation with KMeans.
Run: python 07_customer_segments.py
"""
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score

def make_customers():
    rng = np.random.default_rng(7)
    rows = []
    for _ in range(300):
        visits = rng.integers(1, 25)
        spend = max(10, rng.normal(100 + visits * 35, 60))
        recency = rng.integers(1, 90)
        rows.append([visits, spend, recency])
    return pd.DataFrame(rows, columns=["visits", "spend", "recency"])

def main():
    df = make_customers()
    features = df[["visits", "spend", "recency"]]

    scaler = StandardScaler()
    X = scaler.fit_transform(features)

    best = None
    for k in range(2, 7):
        model = KMeans(n_clusters=k, random_state=42, n_init=10)
        labels = model.fit_predict(X)
        score = silhouette_score(X, labels)
        print(f"k={k} silhouette={score:.3f}")
        if best is None or score > best[0]:
            best = (score, model, labels)

    score, model, labels = best
    df["segment"] = labels
    print("\nSelected k:", model.n_clusters)
```

```
    print("\nSegment averages:")
    print(df.groupby("segment")[["visits", "spend", "recency"]].mean().rour

if __name__ == "__main__":
    main()
```

PROJECT 07 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	numpy, pandas, sklearn.cluster, sklearn.preprocessing, sklearn.metrics
Functions	make_customers, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Creates synthetic customer behavior
2. Standardizes the features
3. Tries several cluster counts
4. Scores clusters with silhouette score
5. Keeps the selected model
6. Prints segment averages

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Name each segment based on its average behavior and justify the name.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 08

Train a Neural Network

Goal: Build a complete PyTorch training loop for digit classification, including device selection and model saving.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install torch scikit-learn
```

RUN

```
python 08_pytorch_mlp.py
```

PROJECT FLOW

Step	What happens
1	Loads the built-in digits dataset
2	Standardizes numeric inputs
3	Builds a PyTorch neural network
4	Runs mini-batch gradient descent
5	Evaluates on held-out data
6	Saves model weights

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 08 — COMPLETE PROGRAM

File: 08_pytorch_mlp.py Lines: 78

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 8
Complete PyTorch classifier using sklearn's built-in digits dataset.
Run: python 08_pytorch_mlp.py
"""
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def prepare_data():
    digits = load_digits()
    X = digits.data.astype("float32")
    y = digits.target.astype("int64")

    X = StandardScaler().fit_transform(X).astype("float32")
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    train_ds = TensorDataset(torch.tensor(X_train), torch.tensor(y_train))
    test_ds = TensorDataset(torch.tensor(X_test), torch.tensor(y_test))
    return DataLoader(train_ds, batch_size=64, shuffle=True), test_ds

class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10),
        )

    def forward(self, x):
        return self.net(x)
```

```
def main():
    train_loader, test_ds = prepare_data()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = Network().to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(10):
        model.train()
        total_loss = 0.0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            optimizer.zero_grad()
            logits = model(X_batch)
            loss = loss_fn(logits, y_batch)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        model.eval()
        with torch.no_grad():
            X_test = test_ds.tensors[0].to(device)
            y_test = test_ds.tensors[1].to(device)
            predictions = model(X_test).argmax(dim=1)
            accuracy = (predictions == y_test).float().mean().item()

        print(
            f"epoch={epoch+1:02d} "
            f"loss={total_loss/len(train_loader):.4f} "
            f"test_accuracy={accuracy:.4f}"
        )

    torch.save(model.state_dict(), "digits_mlp.pt")
    print("Saved: digits_mlp.pt")

if __name__ == "__main__":
    main()
```

PROJECT 08 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	torch, torch, torch.utils.data, sklearn.datasets, sklearn.model_selection, sklearn.preprocessi
Functions	prepare_data, main
Classes	Network
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads the built-in digits dataset
2. Standardizes numeric inputs
3. Builds a PyTorch neural network
4. Runs mini-batch gradient descent
5. Evaluates on held-out data
6. Saves model weights

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add early stopping and compare two learning rates.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program

runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 09

Use a Transformer

Goal: Run a pretrained transformer for sentiment analysis and understand labels, scores, and model downloads.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install transformers torch
```

RUN

```
python 09_transformer_sentiment.py
```

PROJECT FLOW

Step	What happens
1	Loads a pretrained transformer pipeline
2	Feeds sample text to the classifier
3	Prints the predicted label and score
4	Switches to interactive testing

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 09 — COMPLETE PROGRAM

File: 09_transformer_sentiment.py Lines: 35

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 9
Complete transformer sentiment analyzer.
Install: pip install transformers torch
Run: python 09_transformer_sentiment.py
"""
from transformers import pipeline

MODEL = "distilbert-base-uncased-finetuned-sst-2-english"

def main():
    print("Loading model:", MODEL)
    classifier = pipeline("sentiment-analysis", model=MODEL)

    examples = [
        "The service was excellent and the product arrived early.",
        "The application crashed and the support response was slow.",
        "The coffee was okay, but the waiting time was long.",
    ]

    for text in examples:
        result = classifier(text, truncation=True)[0]
        print("\nText:", text)
        print("Label:", result["label"])
        print("Score:", round(result["score"], 4))

    print("\nInteractive mode. Empty input exits.")
    while True:
        text = input("Text: ").strip()
        if not text:
            break
        print(classifier(text, truncation=True)[0])

if __name__ == "__main__":
    main()
```

PROJECT 09 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	transformers
Functions	main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads a pretrained transformer pipeline
2. Feeds sample text to the classifier
3. Prints the predicted label and score
4. Switches to interactive testing

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Create a 20-row test file and inspect where the model disagrees with you.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the

current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 10

Build Semantic Search

Goal: Convert documents and queries into embeddings and retrieve results by semantic similarity.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install sentence-transformers
```

RUN

```
python 10_semantic_search.py
```

PROJECT FLOW

Step	What happens
1	Loads document text
2	Encodes documents into embeddings
3	Encodes the query using the same model
4	Ranks documents by vector similarity
5	Prints the strongest matches

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 10 — COMPLETE PROGRAM

File: 10_semantic_search.py Lines: 47

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 10
Complete semantic-search program using sentence-transformers.
Install: pip install sentence-transformers
Run: python 10_semantic_search.py
"""
from sentence_transformers import SentenceTransformer, util

MODEL = "all-MiniLM-L6-v2"

DOCUMENTS = [
    "Python is a programming language used for data science and AI.",
    "Machine learning learns patterns from examples instead of fixed rules.
    "Neural networks contain layers of trainable parameters.",
    "Transformers are widely used for modern language applications.",
    "Embeddings represent text as vectors that can be compared by similarit
    "RAG retrieves relevant information before generating an answer.",
    "FastAPI can expose Python functions through HTTP endpoints.",
    "Docker packages applications and their dependencies into containers.",
]

def main():
    print("Loading embedding model...")
    model = SentenceTransformer(MODEL)
    document_vectors = model.encode(
        DOCUMENTS, normalize_embeddings=True
    )

    print("\nSemantic search. Empty input exits.")
    while True:
        query = input("Query: ").strip()
        if not query:
            break

        query_vector = model.encode(
            query, normalize_embeddings=True
        )
        scores = util.dot_score(query_vector, document_vectors)[0]
        top_ids = scores.argsort(descending=True)[:3]
```

```
        print("\nTop results:")
        for idx in top_ids:
            i = int(idx)
            print(f"{float(scores[i]):.3f}  {DOCUMENTS[i]}")

if __name__ == "__main__":
    main()
```

PROJECT 10 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	sentence_transformers
Functions	main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads document text
2. Encodes documents into embeddings
3. Encodes the query using the same model
4. Ranks documents by vector similarity
5. Prints the strongest matches

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Replace the documents with your own notes and test multilingual queries.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 11

Build a Local RAG Assistant

Goal: Retrieve relevant local text with TF-IDF and optionally pass the evidence to a local Ollama model.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

pip install scikit-learn requests; install Ollama separately if you want generation.

RUN

```
python 11_rag_local_ollama.py
```

PROJECT FLOW

Step	What happens
1	Reads local text documents
2	Builds TF-IDF retrieval vectors
3	Retrieves the most relevant evidence
4	Builds a grounded prompt
5	Calls a local model when configured
6	Falls back to showing evidence when generation is unavailable

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 11 — COMPLETE PROGRAM

File: 11_rag_local_ollama.py Lines: 95

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 11
Complete local RAG assistant using TF-IDF retrieval + Ollama generation.
No paid API is required.

Setup:
1. Install Ollama and pull a local model using its current official docs.
2. Put .txt files in the documents/ folder.
3. Run: python 11_rag_local_ollama.py

The program still shows retrieved sources if the local model is unavailable
"""
from pathlib import Path
import requests
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

DOCS_DIR = Path("documents")
OLLAMA_URL = "http://localhost:11434/api/generate"
MODEL = "YOUR_LOCAL_MODEL"

def load_documents():
    DOCS_DIR.mkdir(exist_ok=True)
    files = sorted(DOCS_DIR.glob("*.txt"))
    if not files:
        sample = DOCS_DIR / "ai_notes.txt"
        sample.write_text(
            "AI systems learn patterns from data. "
            "Machine learning includes classification and regression. "
            "RAG retrieves source text before generation. "
            "Evaluation is necessary because model outputs can be wrong.",
            encoding="utf-8"
        )
        files = [sample]

    return [(p.name, p.read_text(encoding="utf-8", errors="ignore"))
            for p in files]

def retrieve(query, documents, top_k=3):
    texts = [text for _, text in documents]
```

```
    vectorizer = TfidfVectorizer(stop_words="english")
    matrix = vectorizer.fit_transform(texts + [query])
    scores = cosine_similarity(matrix[-1], matrix[:-1])[0]
    order = scores.argsort()[::-1][:top_k]
    return [(documents[i][0], documents[i][1], float(scores[i]))
            for i in order]

def generate_with_ollama(question, results):
    context = "\n\n".join(
        f"[SOURCE: {name}]\n{text}" for name, text, _ in results
    )
    prompt = (
        "Answer using only the supplied sources. "
        "If the sources do not contain the answer, say you do not know.\n\r
        f"SOURCES:\n{context}\n\nQUESTION:\n{question}"
    )
    response = requests.post(
        OLLAMA_URL,
        json={"model": MODEL, "prompt": prompt, "stream": False},
        timeout=120
    )
    response.raise_for_status()
    return response.json()["response"].strip()

def main():
    documents = load_documents()
    print(f"Loaded {len(documents)} document(s).")

    while True:
        question = input("\nQuestion (Enter to quit): ").strip()
        if not question:
            break

        results = retrieve(question, documents)
        print("\nRetrieved sources:")
        for name, text, score in results:
            print(f"- {name} | similarity={score:.3f}")

        if MODEL == "YOUR_LOCAL_MODEL":
            print("\nSet MODEL to a model installed on your machine "
                  "to enable generation.")
            print("\nRetrieved context:")
            for name, text, _ in results:
                print(f"\n[{name}]\n{text}")
            continue

        try:
            answer = generate_with_ollama(question, results)
            print("\nAI answer:\n", answer)
```

```
        except requests.RequestException as exc:
            print("\nLocal model request failed:", exc)
            print("The retrieval step still completed successfully.")

if __name__ == "__main__":
    main()
```

PROJECT 11 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, requests, sklearn.feature_extraction.text, sklearn.metrics.pairwise
Functions	load_documents, retrieve, generate_with_ollama, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Reads local text documents
2. Builds TF-IDF retrieval vectors
3. Retrieves the most relevant evidence
4. Builds a grounded prompt
5. Calls a local model when configured
6. Falls back to showing evidence when generation is unavailable

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add five documents and require every generated answer to cite a source.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 12

Expose AI Through FastAPI

Goal: Turn a trained model into an HTTP API with validation, health checks, and JSON responses.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install fastapi uvicorn scikit-learn joblib
```

RUN

```
python 04_spam_detector.py then uvicorn
12_fastapi_ai_service:app --reload
```

PROJECT FLOW

Step	What happens
1	Loads the trained spam model at startup
2	Defines validated request and response schemas
3	Exposes a health endpoint
4	Validates input text
5	Returns the model prediction as JSON

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 12 — COMPLETE PROGRAM

File: 12_fastapi_ai_service.py Lines: 59

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 12
Complete FastAPI service around a spam model.

Setup:
1. pip install fastapi uvicorn scikit-learn joblib
2. Run 04_spam_detector.py once to create spam_detector.joblib.
3. Run:
   uvicorn 12_fastapi_ai_service:app --reload

Test:
POST http://127.0.0.1:8000/predict
JSON: {"text":"Congratulations, you won a prize!"}
"""
from pathlib import Path
import joblib
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

MODEL_FILE = Path("spam_detector.joblib")
app = FastAPI(title="Practical AI API", version="1.0.0")

class PredictRequest(BaseModel):
    text: str

class PredictResponse(BaseModel):
    label: str
    confidence: float

def load_model():
    if not MODEL_FILE.exists():
        raise FileNotFoundError(
            "spam_detector.joblib not found. Run 04_spam_detector.py first.
        )
    return joblib.load(MODEL_FILE)

model = None

@app.on_event("startup")
def startup():
```

```
    global model
    model = load_model()

@app.get("/health")
def health():
    return {"status": "ok", "model_loaded": model is not None}

@app.post("/predict", response_model=PredictResponse)
def predict(request: PredictRequest):
    text = request.text.strip()
    if not text:
        raise HTTPException(status_code=400, detail="text cannot be empty")
    probabilities = model.predict_proba([text])[0]
    index = int(probabilities.argmax())
    label = model.classes_[index]
    return PredictResponse(
        label=str(label),
        confidence=float(probabilities[index])
    )
```

PROJECT 12 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, joblib, fastapi, pydantic
Functions	load_model, startup, health, predict
Classes	PredictRequest, PredictResponse
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads the trained spam model at startup
2. Defines validated request and response schemas
3. Exposes a health endpoint
4. Validates input text
5. Returns the model prediction as JSON

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add an API endpoint that returns the model classes.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 13

Build an AI Web App

Goal: Put a trained model behind a simple Streamlit interface.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install streamlit scikit-learn joblib
```

RUN

```
python 04_spam_detector.py then streamlit run
13_streamlit_ai_app.py
```

PROJECT FLOW

Step	What happens
1	Loads and caches the trained model
2	Creates a browser interface with Streamlit
3	Accepts a message
4	Runs the model
5	Displays prediction and class probabilities

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 13 — COMPLETE PROGRAM

File: 13_streamlit_ai_app.py Lines: 48

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 13
Complete Streamlit web app around the spam detector.

Setup:
1. Run 04_spam_detector.py once.
2. pip install streamlit scikit-learn joblib
3. streamlit run 13_streamlit_ai_app.py
"""
from pathlib import Path
import joblib
import streamlit as st

MODEL_FILE = Path("spam_detector.joblib")

st.set_page_config(page_title="Practical AI Detector", page_icon="")
st.title(" Practical AI Message Detector")
st.write("Enter a message and let the trained classifier predict its categc

@st.cache_resource
def load_model():
    if not MODEL_FILE.exists():
        st.error("Model not found. Run 04_spam_detector.py first.")
        st.stop()
    return joblib.load(MODEL_FILE)

model = load_model()
text = st.text_area(
    "Message",
    placeholder="Example: Please confirm tomorrow's meeting."
)

if st.button("Analyze", type="primary"):
    if not text.strip():
        st.warning("Enter a message first.")
    else:
        probabilities = model.predict_proba([text])[0]
        index = int(probabilities.argmax())
        label = model.classes_[index]
        confidence = float(probabilities[index])
```

```
st.subheader("Result")
st.metric("Prediction", str(label))
st.metric("Confidence", f"{confidence:.1%}")

st.subheader("Class probabilities")
for name, probability in zip(model.classes_, probabilities):
    st.write(f"**{name}** — {probability:.1%}")
```

PROJECT 13 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, joblib, streamlit
Functions	load_model
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads and caches the trained model
2. Creates a browser interface with Streamlit
3. Accepts a message
4. Runs the model
5. Displays prediction and class probabilities

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a history panel that shows the last five predictions.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 14

Build a Gradio Interface

Goal: Wrap an AI function in a clean browser interface and expose probabilities.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install gradio scikit-learn joblib
```

RUN

```
python 04_spam_detector.py then python 14_gradio_ai_app.py
```

PROJECT FLOW

Step	What happens
1	Loads the trained model
2	Defines one clean prediction function
3	Builds a Gradio interface
4	Adds examples for quick testing
5	Launches a local browser UI

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 14 — COMPLETE PROGRAM

File: 14_gradio_ai_app.py Lines: 48

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 14
Complete Gradio interface around the spam detector.

Setup:
1. Run 04_spam_detector.py once.
2. pip install gradio scikit-learn joblib
3. python 14_gradio_ai_app.py
"""
from pathlib import Path
import joblib
import gradio as gr

MODEL_FILE = Path("spam_detector.joblib")

if not MODEL_FILE.exists():
    raise SystemExit("Run 04_spam_detector.py first.")

model = joblib.load(MODEL_FILE)

def classify(text):
    if not text or not text.strip():
        return {"error": 1.0}

    probabilities = model.predict_proba([text])[0]
    return {
        str(name): float(probability)
        for name, probability in zip(model.classes_, probabilities)
    }

demo = gr.Interface(
    fn=classify,
    inputs=gr.Textbox(
        lines=5,
        label="Message",
        placeholder="Type a message..."
    ),
    outputs=gr.Label(label="Prediction"),
    title="Practical AI Message Detector",
    description="A complete ML model exposed through a browser interface.",
```

```
    examples=[
        ["Congratulations! Claim your free reward now."],
        ["Can we meet after work today?"],
    ],
)

if __name__ == "__main__":
    demo.launch()
```

PROJECT 14 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, joblib, gradio
Functions	classify
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads the trained model
2. Defines one clean prediction function
3. Builds a Gradio interface
4. Adds examples for quick testing
5. Launches a local browser UI

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a second input mode and a custom example list.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 15

Build a Voice Assistant

Goal: Capture speech, recognize it, route it to safe commands, and speak the response.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install SpeechRecognition pyttsx3 PyAudio
```

RUN

```
python 15_voice_assistant.py
```

PROJECT FLOW

Step	What happens
1	Initializes speech recognition and text-to-speech
2	Captures microphone audio
3	Converts speech to text
4	Routes recognized phrases to safe commands
5	Speaks the response
6	Handles common microphone and recognition failures

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 15 — COMPLETE PROGRAM

File: 15_voice_assistant.py Lines: 69

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 15
Complete voice assistant using local text rules.

Setup:
pip install SpeechRecognition pyttsx3 PyAudio
Run: python 15_voice_assistant.py

For microphone installation issues, consult the package documentation
for your operating system.
"""
from datetime import datetime
import sys
import speech_recognition as sr
import pyttsx3

def speak(text, engine):
    print("AI:", text)
    engine.say(text)
    engine.runAndWait()

def answer(text):
    lower = text.lower()
    if "time" in lower:
        return f"The time is {datetime.now().strftime('%H:%M')}."
    if "date" in lower:
        return f"Today is {datetime.now().strftime('%A, %d %B %Y')}."
    if "hello" in lower or "hi" in lower:
        return "Hello! What would you like to do?"
    if "your name" in lower:
        return "I am your local voice assistant."
    if "help" in lower:
        return "You can ask for the time, date, or say goodbye."
    return "I heard you, but I do not have a tool for that request yet."

def main():
    recognizer = sr.Recognizer()
    engine = pyttsx3.init()
    speak("Voice assistant ready. Say goodbye to exit.", engine)
```

```
    while True:
        try:
            with sr.Microphone() as source:
                print("\nListening...")
                recognizer.adjust_for_ambient_noise(source, duration=0.5)
                audio = recognizer.listen(source, timeout=5, phrase_time_li

            text = recognizer.recognize_google(audio)
            print("You:", text)

            if "goodbye" in text.lower() or "exit" in text.lower():
                speak("Goodbye!", engine)
                break

            speak(answer(text), engine)

        except sr.WaitTimeoutError:
            print("No speech detected.")
        except sr.UnknownValueError:
            speak("I could not understand that. Please try again.", engine)
        except sr.RequestError as exc:
            print("Speech service error:", exc)
            break
        except KeyboardInterrupt:
            print("\nStopped.")
            break

if __name__ == "__main__":
    main()
```

PROJECT 15 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	datetime, sys, speech_recognition, pyttsx3
Functions	speak, answer, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Initializes speech recognition and text-to-speech
2. Captures microphone audio
3. Converts speech to text
4. Routes recognized phrases to safe commands
5. Speaks the response
6. Handles common microphone and recognition failures

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add one new safe command and handle its failure case.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program

runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 16

Build a Safe AI Agent

Goal: Create a tool-using agent with a calculator and document search while keeping the tool surface deliberately restricted.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn
```

RUN

```
python 16_safe_agent.py
```

PROJECT FLOW

Step	What happens
1	Defines a very small approved tool set
2	Parses arithmetic with Python's AST
3	Searches local documents with TF-IDF
4	Chooses a tool from a narrow rule set
5	Returns evidence and tool results

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 16 — COMPLETE PROGRAM

File: 16_safe_agent.py Lines: 84

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 16
Complete safe tool-calling style agent.
It can use only two local tools: calculator and document search.

Run: python 16_safe_agent.py
"""
from pathlib import Path
import ast
import operator as op
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

DOCUMENTS = {
    "python.txt": "Python is used for scripting, automation, data science, and AI.",
    "ml.txt": "Machine learning learns patterns from data. Classification predicts categories.",
    "rag.txt": "RAG retrieves relevant source text before a language model generates an answer.",
}

ALLOWED_OPERATORS = {
    ast.Add: op.add,
    ast.Sub: op.sub,
    ast.Mult: op.mul,
    ast.Div: op.truediv,
    ast.USub: op.neg,
}

def safe_calculator(expression):
    tree = ast.parse(expression, mode="eval")

    def evaluate(node):
        if isinstance(node, ast.Constant) and isinstance(node.value, (int, float)):
            return node.value
        if isinstance(node, ast.UnaryOp) and type(node.op) in ALLOWED_OPERATORS:
            return ALLOWED_OPERATORS[type(node.op)](evaluate(node.operand))
        if isinstance(node, ast.BinOp) and type(node.op) in ALLOWED_OPERATORS:
            return ALLOWED_OPERATORS[type(node.op)](
                evaluate(node.left), evaluate(node.right)
            )
        raise ValueError("Only numbers and + - * / are allowed.")
```

```
    return evaluate(tree.body)

def search_documents(query, top_k=2):
    names = list(DOCUMENTS)
    texts = list(DOCUMENTS.values())
    vectorizer = TfidfVectorizer()
    matrix = vectorizer.fit_transform(texts + [query])
    scores = cosine_similarity(matrix[-1], matrix[:-1])[0]
    order = scores.argsort()[::-1][:top_k]
    return [(names[i], texts[i], float(scores[i])) for i in order]

def agent(question):
    lower = question.lower()

    if lower.startswith("calculate "):
        expression = question[len("calculate "):].strip()
        try:
            return f"Calculator result: {safe_calculator(expression)}"
        except Exception as exc:
            return f"Calculator error: {exc}"

    results = search_documents(question)
    best_name, best_text, score = results[0]

    return (
        f"Best source: {best_name} (similarity {score:.3f})\n"
        f"Evidence: {best_text}\n\n"
        "This demo deliberately uses only approved local tools. "
        "A production agent should validate every tool call and permission.
    )

def main():
    print("Safe AI Agent")
    print("Try: calculate 12 * (3 + 2)")
    print("Or ask: What is RAG?")
    while True:
        question = input("\nYou: ").strip()
        if not question:
            break
        print("\nAgent:", agent(question))

if __name__ == "__main__":
    main()
```

PROJECT 16 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, ast, operator, sklearn.feature_extraction.text, sklearn.metrics.pairwise
Functions	safe_calculator, search_documents, agent, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Defines a very small approved tool set
2. Parses arithmetic with Python's AST
3. Searches local documents with TF-IDF
4. Chooses a tool from a narrow rule set
5. Returns evidence and tool results

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a third read-only tool and validate every argument before execution.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 17

Build an Evaluation Harness

Goal: Run repeatable tests against an AI model and save expected versus actual results.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn joblib
```

RUN

```
python 04_spam_detector.py then python 17_evaluation_harness.py
```

PROJECT FLOW

Step	What happens
1	Loads a previously trained classifier
2	Defines repeatable test cases
3	Runs predictions
4	Calculates accuracy and a report
5	Writes expected versus actual results to CSV

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 17 — COMPLETE PROGRAM

File: 17_evaluation_harness.py Lines: 47

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 17
Complete evaluation harness for a simple classifier.

Run: python 17_evaluation_harness.py
"""
import csv
from pathlib import Path
import joblib
from sklearn.metrics import accuracy_score, classification_report

MODEL_FILE = Path("spam_detector.joblib")
RESULT_FILE = Path("evaluation_results.csv")

TESTS = [
    ("Win a free prize now", "spam"),
    ("Claim your reward immediately", "spam"),
    ("Can we meet at 3 PM?", "normal"),
    ("Please send the report", "normal"),
    ("Exclusive bonus waiting for you", "spam"),
    ("The package was delivered", "normal"),
    ("Click now to receive money", "spam"),
    ("I will call you after work", "normal"),
]

def main():
    if not MODEL_FILE.exists():
        raise SystemExit("Run 04_spam_detector.py first.")

    model = joblib.load(MODEL_FILE)
    texts = [x[0] for x in TESTS]
    expected = [x[1] for x in TESTS]
    actual = model.predict(texts)

    print("Accuracy:", accuracy_score(expected, actual))
    print(classification_report(expected, actual))

    with RESULT_FILE.open("w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["input", "expected", "actual", "pass"])
```

```
            for text, exp, got in zip(texts, expected, actual):
                writer.writerow([text, exp, got, exp == got])

    print("Saved:", RESULT_FILE)

if __name__ == "__main__":
    main()
```

PROJECT 17 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	csv, pathlib, joblib, sklearn.metrics
Functions	main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads a previously trained classifier
2. Defines repeatable test cases
3. Runs predictions
4. Calculates accuracy and a report
5. Writes expected versus actual results to CSV

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add 20 edge cases and make a change that improves one metric without breaking another.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 18

Final Knowledge Assistant

Goal: Combine retrieval, source citations, a safe calculator, local files, and an interactive interface into one complete capstone.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn
```

RUN

```
python 18_capstone_assistant.py
```

PROJECT FLOW

Step	What happens
1	Creates a knowledge folder when needed
2	Loads local text files
3	Supports a safe calculator
4	Retrieves relevant documents
5	Returns evidence with source names
6	Provides a complete interactive loop

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 18 — COMPLETE PROGRAM

File: 18_capstone_assistant.py Lines: 116

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 18
Complete final capstone: local knowledge assistant.

Features:
- Reads .txt documents from knowledge/
- TF-IDF retrieval
- Source citations
- Safe calculator tool
- Interactive command line
- No paid API required

Run: python 18_capstone_assistant.py
"""
from pathlib import Path
import ast
import operator as op
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

KNOWLEDGE = Path("knowledge")
OPERATORS = {
    ast.Add: op.add,
    ast.Sub: op.sub,
    ast.Mult: op.mul,
    ast.Div: op.truediv,
}

def load_knowledge():
    KNOWLEDGE.mkdir(exist_ok=True)
    if not list(KNOWLEDGE.glob("*.txt")):
        (KNOWLEDGE / "ai_basics.txt").write_text(
            "Artificial intelligence includes systems that perform tasks "
"associated with intelligent behavior. Machine learning learns "
"patterns from examples. Retrieval augmented generation retrieves "
"supporting information before generation. AI evaluation measures "
            "whether a system behaves as intended.",
            encoding="utf-8"
        )
```

```
    documents = []
    for path in sorted(KNOWLEDGE.glob("*.txt")):
        text = path.read_text(encoding="utf-8", errors="ignore").strip()
        if text:
            documents.append((path.name, text))

    if not documents:
        raise RuntimeError("Add at least one .txt file to knowledge/.")
    return documents

def calculate(expression):
    tree = ast.parse(expression, mode="eval")

    def visit(node):
    if isinstance(node, ast.Constant) and isinstance(node.value, (int, float)):
            return node.value
        if isinstance(node, ast.BinOp) and type(node.op) in OPERATORS:
     return OPERATORS[type(node.op)](visit(node.left), visit(node.right))
    if isinstance(node, ast.UnaryOp) and isinstance(node.op, ast.USub):
            return -visit(node.operand)
        raise ValueError("Unsupported expression.")

    return visit(tree.body)

def retrieve(query, documents, top_k=3):
    texts = [text for _, text in documents]
    vectorizer = TfidfVectorizer(stop_words="english")
    matrix = vectorizer.fit_transform(texts + [query])
    scores = cosine_similarity(matrix[-1], matrix[:-1])[0]
    ids = scores.argsort()[::-1][:min(top_k, len(documents))]
  return [(documents[i][0], documents[i][1], float(scores[i])) for i in ids]

def answer(question, documents):
    stripped = question.strip()

    if stripped.lower().startswith("calculate "):
        expression = stripped[10:].strip()
        try:
            return f"Calculator: {calculate(expression)}"
        except Exception as exc:
            return f"Calculator error: {exc}"

    results = retrieve(stripped, documents)
    best_score = results[0][2]

    if best_score == 0:
    return "I could not find relevant information in the knowledge base."

    lines = ["Evidence found in your knowledge base:"]
```

```
    for name, text, score in results:
        lines.append(f"\n[{name} | similarity={score:.3f}]\n{text}")
    lines.append(
        "\nNote: this capstone is retrieval-based. "
        "For generative answers, connect the retrieved context to a "
        "language model only after adding evaluation and access controls."
    )
    return "\n".join(lines)

def main():
    documents = load_knowledge()
    print("=" * 70)
    print("BUILD YOUR OWN AI — FINAL CAPSTONE")
    print("=" * 70)
    print(f"Loaded {len(documents)} knowledge file(s).")
    print("Commands: calculate <expression>, or ask a knowledge question.")
    print("Press Enter on an empty line to quit.")

    while True:
        question = input("\nYou: ").strip()
        if not question:
            print("Goodbye.")
            break
        print("\nAssistant:\n" + answer(question, documents))

if __name__ == "__main__":
    main()
```

PROJECT 18 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, ast, operator, sklearn.feature_extraction.text, sklearn.metrics.pairwise
Functions	load_knowledge, calculate, retrieve, answer, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Creates a knowledge folder when needed
2. Loads local text files
3. Supports a safe calculator
4. Retrieves relevant documents
5. Returns evidence with source names
6. Provides a complete interactive loop

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a web UI only after the command-line version passes your evaluation tests.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 19

Track AI Experiments

Goal: Use MLflow to compare multiple model configurations and record parameters and metrics.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install mlflow scikit-learn
```

RUN

```
python 19_mlflow_experiment.py; then mlflow ui
```

PROJECT FLOW

Step	What happens
1	Creates an MLflow experiment
2	Trains several model configurations
3	Logs parameters
4	Logs evaluation metrics
5	Makes runs comparable through the MLflow UI

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 19 — COMPLETE PROGRAM

File: 19_mlflow_experiment.py Lines: 45

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 19
Complete MLflow experiment tracker.
Install: pip install mlflow scikit-learn
Run: python 19_mlflow_experiment.py
"""
import mlflow
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

def run_experiment(n_estimators):
    data = load_iris()
    X_train, X_test, y_train, y_test = train_test_split(
        data.data, data.target, test_size=0.2,
        random_state=42, stratify=data.target
    )

    with mlflow.start_run():
        model = RandomForestClassifier(
            n_estimators=n_estimators, random_state=42
        )
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)

        mlflow.log_param("n_estimators", n_estimators)
        mlflow.log_metric("accuracy", accuracy)

        print(
            f"n_estimators={n_estimators} "
            f"accuracy={accuracy:.4f}"
        )

def main():
    mlflow.set_experiment("iris-complete-project")
    for value in [50, 100, 200]:
        run_experiment(value)
```

```
    print("\nStart the MLflow UI with:")
    print("mlflow ui")

if __name__ == "__main__":
    main()
```

PROJECT 19 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	mlflow, sklearn.datasets, sklearn.ensemble, sklearn.metrics, sklearn.model_selection
Functions	run_experiment, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Creates an MLflow experiment
2. Trains several model configurations
3. Logs parameters
4. Logs evaluation metrics
5. Makes runs comparable through the MLflow UI

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add a second metric and identify which parameter change caused it.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 20

Dockerize an AI API

Goal: Package a complete FastAPI service in a Docker image so the environment is reproducible.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

pip install fastapi uvicorn pydantic; Docker Desktop or Docker Engine for container steps.

RUN

```
uvicorn 20_docker_fastapi_project:app --host 0.0.0.0 --port 8000; then docker build -t practical-ai-api .
```

PROJECT FLOW

Step	What happens
1	Defines a small FastAPI application
2	Validates incoming JSON
3	Exposes health and analysis endpoints
4	Returns structured responses
5	Packages the application in a Docker image

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 20 — COMPLETE PROGRAM

File: 20_docker_fastapi_project.py Lines: 53

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 20
Complete Docker-ready FastAPI application.

Files used by this project:
- 20_docker_fastapi_project.py
- requirements_docker.txt
- Dockerfile

Run locally:
pip install -r requirements_docker.txt
uvicorn 20_docker_fastapi_project:app --host 0.0.0.0 --port 8000

Build:
docker build -t practical-ai-api .
docker run --rm -p 8000:8000 practical-ai-api
"""
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(
    title="Practical AI Starter API",
    description="A small production-style API with validation and health ch
    version="1.0.0",
)

class TextRequest(BaseModel):
    text: str

class TextResponse(BaseModel):
    characters: int
    words: int
    message: str

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/analyze", response_model=TextResponse)
def analyze(request: TextRequest):
```

```
    text = request.text.strip()
    if not text:
        raise HTTPException(
            status_code=400,
            detail="text cannot be empty"
        )

    words = len(text.split())
    return TextResponse(
        characters=len(text),
        words=words,
        message="Input received successfully."
    )
```

PROJECT 20 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	fastapi, pydantic
Functions	health, analyze
Classes	TextRequest, TextResponse
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Defines a small FastAPI application
2. Validates incoming JSON
3. Exposes health and analysis endpoints
4. Returns structured responses
5. Packages the application in a Docker image

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Add an automated health check to the Docker deployment.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 21

Secure an AI Project

Goal: Run a defensive checker that searches a local project for obvious secrets and missing safety files.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

No third-party package required.

RUN

```
python 21_ai_security_checker.py
```

PROJECT FLOW

Step	What happens
1	Walks through a local project
2	Scans text files for simple credential patterns
3	Checks for a useful .gitignore
4	Prints defensive findings
5	Clearly states that it is only a starter checker

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 21 — COMPLETE PROGRAM

File: 21_ai_security_checker.py Lines: 67

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 21
Complete defensive AI configuration checker.
It checks a local project for common safety mistakes.

Run: python 21_ai_security_checker.py
"""
from pathlib import Path
import re

ROOT = Path(".")
SECRET_PATTERNS = [
    re.compile(r"sk-[A-Za-z0-9_-]{20,}"),
    re.compile(r"api[_-]?key\s*=\s*['\"][^'\"]+['\"]", re.I),
    re.compile(r"password\s*=\s*['\"][^'\"]+['\"]", re.I),
]

def scan_file(path):
    findings = []
    try:
        text = path.read_text(encoding="utf-8", errors="ignore")
    except Exception:
        return findings

    for number, line in enumerate(text.splitlines(), start=1):
        for pattern in SECRET_PATTERNS:
            if pattern.search(line):
                findings.append(
                    f"{path}:{number}: possible secret or credential"
                )
    return findings

def check_gitignore():
    path = ROOT / ".gitignore"
    if not path.exists():
        return ["Missing .gitignore"]
    text = path.read_text(encoding="utf-8", errors="ignore")
    required = [".env", "__pycache__", "*.pyc"]
    return [
        f".gitignore should include {item}"
```

```
        for item in required
        if item not in text
    ]

def main():
    findings = []
    for path in ROOT.rglob("*"):
        if path.is_file() and ".git" not in path.parts:
            findings.extend(scan_file(path))

    findings.extend(check_gitignore())

    print("=" * 60)
    print("DEFENSIVE AI PROJECT SECURITY CHECK")
    print("=" * 60)

    if findings:
        print("\nFindings:")
        for item in findings:
            print("-", item)
    else:
        print("\nNo obvious issues found by these simple checks.")
```

print("\nImportant: this is a starter checker, not a complete security audit.")

```
if __name__ == "__main__":
    main()
```

PROJECT 21 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	pathlib, re
Functions	scan_file, check_gitignore, main
Classes	No classes
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Walks through a local project
2. Scans text files for simple credential patterns
3. Checks for a useful .gitignore
4. Prints defensive findings
5. Clearly states that it is only a starter checker

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Extend the checker to flag debug mode and unsafe file permissions in your own test project.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before

immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

PROJECT 22

Build Production-Style RAG

Goal: Implement chunking, document IDs, tenant filtering, retrieval, and source-aware output as a foundation for a production system.

WHAT YOU WILL BUILD

A complete program that can be copied into a file, installed, executed, tested, and then extended. The code is intentionally self-contained where practical.

INSTALL

```
pip install scikit-learn
```

RUN

```
python 22_production_rag.py
```

PROJECT FLOW

Step	What happens
1	Loads documents for a tenant
2	Splits long text into overlapping chunks
3	Assigns document IDs and metadata
4	Builds a TF-IDF retrieval index
5	Filters by tenant before retrieval
6	Returns source-aware evidence

Beginner rule: Do not change the code before you have run the original version successfully. First prove the baseline works; then experiment.

PROJECT 22 — COMPLETE PROGRAM

File: 22_production_rag.py Lines: 132

COPY THIS WHOLE PROGRAM. The listing below is the complete source for this project; nothing essential is intentionally omitted.

```
"""
BUILD YOUR OWN AI — Project 22
Complete production-style RAG skeleton with:
- document IDs
- tenant filtering
- chunking
- TF-IDF retrieval
- citations
- evaluation hooks

Run: python 22_production_rag.py
"""
from dataclasses import dataclass
from pathlib import Path
from typing import List
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

@dataclass
class Chunk:
    document_id: str
    tenant_id: str
    source: str
    text: str

def chunk_text(text, size=500, overlap=80):
    text = " ".join(text.split())
    chunks = []
    start = 0

    while start < len(text):
        end = min(len(text), start + size)
        chunks.append(text[start:end])
        if end == len(text):
            break
        start = end - overlap

    return chunks

def load_chunks(root, tenant_id):
```

```
    root = Path(root)
    chunks: List[Chunk] = []

    for path in sorted(root.glob("*.txt")):
        text = path.read_text(encoding="utf-8", errors="ignore")
        for number, piece in enumerate(chunk_text(text), start=1):
            chunks.append(
                Chunk(
                    document_id=f"{path.stem}-{number}",
                    tenant_id=tenant_id,
                    source=path.name,
                    text=piece,
                )
            )
    return chunks

class Retriever:
    def __init__(self, chunks):
        self.chunks = chunks
        self.vectorizer = TfidfVectorizer(stop_words="english")
        self.matrix = self.vectorizer.fit_transform(
            [chunk.text for chunk in chunks]
        )

    def search(self, query, tenant_id, top_k=3):
        allowed = [
            i for i, chunk in enumerate(self.chunks)
            if chunk.tenant_id == tenant_id
        ]
        if not allowed:
            return []

        query_vector = self.vectorizer.transform([query])
        scores = cosine_similarity(
            query_vector, self.matrix[allowed]
        )[0]

        ranked = sorted(
            zip(allowed, scores),
            key=lambda item: item[1],
            reverse=True
        )[:top_k]

        return [
            (self.chunks[i], float(score))
            for i, score in ranked
        ]

def answer(query, results):
```

```
    if not results:
        return "No authorized documents were found."

    if all(score == 0 for _, score in results):
        return "No relevant evidence was found."

    lines = ["Answer from authorized evidence:"]
    for chunk, score in results:
        lines.append(
            f"\n[{chunk.source} | {chunk.document_id} | {score:.3f}]\n"
            f"{chunk.text}"
        )
    return "\n".join(lines)

def main():
    root = Path("tenant_docs")
    root.mkdir(exist_ok=True)

    sample = root / "guide.txt"
    if not sample.exists():
        sample.write_text(
            "RAG combines retrieval and generation. "
            "A secure system filters documents before retrieval results "
            "are passed to a model. Evaluation should test both relevance "
            "and access control.",
            encoding="utf-8"
        )

    chunks = load_chunks(root, tenant_id="tenant-a")
    if not chunks:
        raise SystemExit("Add .txt files to tenant_docs/.")

    retriever = Retriever(chunks)

    while True:
        query = input("\nQuestion (Enter to quit): ").strip()
        if not query:
            break
        results = retriever.search(query, tenant_id="tenant-a")
        print(answer(query, results))

if __name__ == "__main__":
    main()
```

PROJECT 22 — UNDERSTAND THE PROGRAM

HOW THE CODE WAS BUILT

The program is assembled in layers rather than as one giant block. Start at the top of the file and follow the data from input to processing to output. The following map shows the major pieces detected in the complete source.

Part	Purpose
Imports	dataclasses, pathlib, typing, sklearn.feature_extraction.text, sklearn.metrics.pairwise
Functions	chunk_text, load_chunks, answer, main
Classes	Chunk, Retriever
Main flow	The main() function, or the module-level application flow, connects the pieces.

READ THE PROGRAM IN THIS ORDER

1. Loads documents for a tenant
2. Splits long text into overlapping chunks
3. Assigns document IDs and metadata
4. Builds a TF-IDF retrieval index
5. Filters by tenant before retrieval
6. Returns source-aware evidence

TEST IT

Run the program exactly as printed. Then test at least three cases: a normal input, an edge case, and an invalid or empty input where the program supports it. Write down the output. This turns running code into an experiment.

WHAT TO CHANGE NEXT

Create two tenants and prove that each tenant can retrieve only its own documents.

COMMON FAILURE POINTS

If Python says a module is missing, check that the virtual environment is active and install the package shown in the INSTALL section. If the program runs but the result is poor, inspect the data and evaluation before immediately changing the model. If a model download fails, check the current model/library documentation and your network connection.

Knowledge checkpoint: Explain the program to another beginner without reading the code. If you can describe the input, transformation, model/tool, output, and failure cases, you understand the architecture—not just the syntax.

Appendix A — Master Installation Guide

• Create a project folder and a virtual environment before installing packages.

• Windows PowerShell: python -m venv .venv then .\.venv\Scripts\Activate.ps1

• Linux/macOS/WSL: python3 -m venv .venv then source .venv/bin/activate

• Upgrade packaging tools: python -m pip install --upgrade pip

• Install only the requirements for the project you are running.

• Keep a requirements.txt file for every project.

• Do not install every package in the book just to run Project 1.

Practical habit: keep this checklist beside your project and update it as your software stack changes.

Appendix B — How to Read Python AI Code

• Imports tell you which tools the program depends on.

• Constants at the top make settings easy to change.

• Functions turn repeated work into named steps.

• Classes group data and behavior when a project becomes larger.

• main() is a common entry point that makes the execution flow easy to find.

• if __name__ == '__main__': prevents main() from running when the file is imported.

• Models normally follow a train/evaluate/predict or load/predict pattern.

• APIs normally follow input validation -> processing -> structured response.

• RAG normally follows documents -> chunks -> retrieval -> evidence -> generation.

Practical habit: keep this checklist beside your project and update it as your software stack changes.

Appendix C — Debugging Checklist

- Read the complete error message, including the last line.
- Confirm the virtual environment is active.
- Print the values entering the failing function.
- Check data types and missing values.
- Reduce the input to the smallest example that still fails.
- Change one thing at a time.
- Keep the last working version in Git.
- Do not paste API keys into source files or screenshots.
- After fixing an error, add a test so it does not silently return.

Practical habit: keep this checklist beside your project and update it as your software stack changes.

Appendix D — Beginner-to-Advanced Roadmap

• Stage 1: Python, files, functions, and data cleaning.

• Stage 2: classification, regression, recommendations, and clustering.

• Stage 3: neural networks and PyTorch training loops.

• Stage 4: transformers, embeddings, and semantic search.

• Stage 5: RAG, chatbots, interfaces, and voice.

• Stage 6: tools, agents, evaluation, experiment tracking, and APIs.

• Stage 7: Docker, security, access control, and production-style retrieval.

• Stage 8: build your own product and replace the example data with a real authorized use case.

Practical habit: keep this checklist beside your project and update it as your software stack changes.

Appendix E — Final Capstone Checklist

- ☐ The program works from a clean environment.
- ☐ Every dependency is documented.
- ☐ The README explains installation and usage.
- ☐ The model, prompt, or retrieval configuration is recorded.
- ☐ There is a repeatable evaluation set.
- ☐ Inputs are validated.
- ☐ Errors are handled clearly.
- ☐ Secrets are kept outside source code.
- ☐ Data is used with permission.
- ☐ Logs do not expose sensitive information.
- ☐ Tool permissions follow least privilege.
- ☐ The project has known limitations documented.
- ☐ Another person can run it using the README.
- ☐ You can explain every major component.

Practical habit: keep this checklist beside your project and update it as your software stack changes.

About the Author

Kyambadde Grrald is a technology enthusiast and practical learner interested in software, AI, cybersecurity, and real-world digital projects. This book was created for readers who want to move from watching tutorials to building complete programs.

The teaching approach is simple: build a complete version first, understand the important pieces, test the result, and then improve it. The source-code companion included with this edition is organized so readers can copy each project into its own folder.

Build it. Understand it. Improve it. Ship it.

PROJECT 23 — HUGGING FACE

Build a transformer application by connecting the Hugging Face Hub, tokenizer, model, inference, probabilities, and local model storage.

What you will build

A complete sentiment classifier that loads a pretrained sequence-classification model from the Hugging Face Hub, tokenizes text with AutoTokenizer, runs inference with AutoModelForSequenceClassification, converts logits to probabilities, prints the model labels, and optionally saves the model locally. This is intentionally more transparent than hiding everything behind a single pipeline call.

Install

```
python -m venv .venv
# Windows PowerShell
.venv\Scripts\Activate.ps1
# Linux/macOS
source .venv/bin/activate
pip install -r requirements.txt
```

Run

```
python 23_hugging_face_sentiment.py
```

The first run may download model files. Model size, hardware needs, and download time depend on the selected checkpoint.

> Hugging Face is not a single model. The Hub is a platform for sharing model repositories, datasets, and related artifacts. The Transformers library supplies reusable model and tokenizer classes and higher-level pipelines.

Project flow

Step	What happens
1	Choose a model ID from the Hugging Face Hub.
2	AutoTokenizer loads the vocabulary and tokenization configuration.
3	AutoModelForSequenceClassification loads model weights and configuration.
4	Text becomes tensors through tokenization.
5	The model produces logits.
6	Softmax converts logits into class probabilities.
7	The best class and confidence are printed.
8	Optional save_pretrained stores reusable local files.

PROJECT 23 — COMPLETE PROGRAM (PART 1)

Copy the complete source from the companion ZIP. The pages below show the full program without omitting the core logic.

```
"""
BUILD YOUR OWN AI — Project 23
Hugging Face transformer sentiment classifier.

This project demonstrates the Hugging Face Hub -> tokenizer -> model ->
inference pipeline without hiding the important steps behind a helper.

Install:
    pip install -r requirements.txt

Run:
    python 23_hugging_face_sentiment.py

Optional model override:
    set HF_MODEL_ID=distilbert-base-uncased-finetuned-sst-2-english
    python 23_hugging_face_sentiment.py
"""

import os
from pathlib import Path

import torch
from transformers import AutoModelForSequenceClassification, AutoTokenizer

DEFAULT_MODEL = "distilbert-base-uncased-finetuned-sst-2-english"
MODEL_ID = os.getenv("HF_MODEL_ID", DEFAULT_MODEL)
SAVE_DIR = Path("saved_hugging_face_model")

EXAMPLES = [
    "The service was excellent and the product arrived early.",
    "The application crashed and the support response was slow.",
    "The coffee was okay, but the waiting time was long.",
]


def load_model():
    """Download the tokenizer and model from the Hugging Face Hub."""
    print("Loading model from Hugging Face Hub:", MODEL_ID)
    tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
    model = AutoModelForSequenceClassification.from_pretrained(MODEL_ID)
    model.eval()
    return tokenizer, model
```

Read this part carefully: the model ID is a repository identifier, not a file path. from_pretrained() can retrieve model/tokenizer files from the Hub and can also load a previously saved local directory.

PROJECT 23 — COMPLETE PROGRAM (PART 2)

Copy the complete source from the companion ZIP. The pages below show the full program without omitting the core logic.

```
def classify(text, tokenizer, model):
    """Tokenize one text, run the model, and return label probabilities."""
    inputs = tokenizer(
        text,
        return_tensors="pt",
        truncation=True,
        padding=True,
    )

    with torch.no_grad():
        outputs = model(**inputs)
        probabilities = torch.softmax(outputs.logits, dim=-1)[0]

    best_id = int(torch.argmax(probabilities).item())
    label = model.config.id2label.get(best_id, str(best_id))
    confidence = float(probabilities[best_id].item())

    return label, confidence, probabilities


def print_prediction(text, tokenizer, model):
    label, confidence, probabilities = classify(text, tokenizer, model)
    print("\nText:", text)
    print(f"Prediction: {label}")
    print(f"Confidence: {confidence:.4f}")

    print("All class probabilities:")
    for index, probability in enumerate(probabilities):
        name = model.config.id2label.get(index, str(index))
        print(f"  {name}: {float(probability):.4f}")


def save_model(tokenizer, model):
    """Save model files locally so they can be reused without downloading again."""
    SAVE_DIR.mkdir(parents=True, exist_ok=True)
    tokenizer.save_pretrained(SAVE_DIR)
    model.save_pretrained(SAVE_DIR)
    print("\nSaved local model to:", SAVE_DIR.resolve())


def main():
    print("=" * 72)
    print("BUILD YOUR OWN AI — PROJECT 23: HUGGING FACE")
    print("=" * 72)

    tokenizer, model = load_model()
```

Why logits first? The neural network returns raw scores. Softmax converts those scores into a probability distribution. The highest probability is used only as the demo prediction; production systems should also define thresholds, evaluation data, and failure handling.

PROJECT 23 — COMPLETE PROGRAM (PART 3)

Copy the complete source from the companion ZIP. The pages below show the full program without omitting the core logic.

```
    print("\nBuilt-in tests")
    for text in EXAMPLES:
        print_prediction(text, tokenizer, model)

    answer = input("\nSave tokenizer and model locally? [y/N]: ").strip().lower()
    if answer == "y":
        save_model(tokenizer, model)

    print("\nInteractive mode. Press Enter on an empty line to exit.")
    while True:
        text = input("Text: ").strip()
        if not text:
            break
        print_prediction(text, tokenizer, model)


if __name__ == "__main__":
    main()
```

Reproducibility: save the model and tokenizer together. Record the model ID, library versions, hardware, and evaluation results so another developer can reproduce the experiment.

UNDERSTAND THE HUGGING FACE STACK

The fastest way to become productive with Hugging Face is to understand which layer is doing the work.

1. Hugging Face Hub

The Hub hosts model and dataset repositories. A model repository can include weights, configuration, tokenizer files, documentation, and a model card. Treat the model card as part of the engineering process: inspect intended use, limitations, training information, evaluation, and license before selecting a checkpoint.

2. Transformers

Transformers provides model architectures, tokenizers, processors, configuration objects, and task-oriented pipelines. Auto classes let your code select an appropriate implementation from a pretrained checkpoint.

3. Tokenizer

A tokenizer converts human-readable text into model inputs such as token IDs and attention masks. The tokenizer must match the model family/checkpoint. Do not swap a tokenizer randomly and expect reliable results.

4. Model

The model transforms tokenized inputs into outputs. For classification, the output commonly contains logits for each class. The checkpoint configuration maps class IDs to human-readable labels when that metadata is available.

5. Pipeline

A pipeline is a convenient high-level interface that can combine preprocessing, model inference, and post-processing. Project 9 already uses a pipeline. Project 23 deliberately exposes the lower-level steps so you can see what the helper is doing.

6. Datasets

The Hugging Face Datasets ecosystem provides tools for loading, preparing, processing, and sharing datasets. A real training project should document its data source, split strategy, preprocessing, license, and evaluation method.

PIPELINE VS. EXPLICIT MODEL CODE

Both styles are useful. Use the simplest interface that still gives you the control your project needs.

Approach	Best for	Trade-off
pipeline()	Fast prototypes and learning task workflows.	Hides some implementation details.
AutoTokenizer + AutoModel...	Learning, debugging, custom preprocessing, controlled inference.	More code and more decisions.
Fine-tuning	Adapting a pretrained model to a documented task/data set.	Requires careful data, evaluation, compute, and licensing review.
Hub sharing	Versioned collaboration and model/dataset distribution.	Requires good documentation and access control.

Pipeline example

```
from transformers import pipeline

classifier = pipeline(
    "sentiment-analysis",
    model="distilbert-base-uncased-finetuned-sst-2-english",
)
print(classifier("The product arrived early."))
```

The helper can automatically load the model and tokenizer required for the task. The exact model and task must still be checked for suitability, license, language coverage, and evaluation quality.

Explicit example

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
model = AutoModelForSequenceClassification.from_pretrained(MODEL_ID)
inputs = tokenizer(text, return_tensors="pt", truncation=True)
outputs = model(**inputs)
```

REAL-WORLD HUGGING FACE WORKFLOW

A model demo is only the first step. For a useful application, move through a repeatable workflow.

1. Define the task — Write the input, output, acceptable errors, latency target, and success metric before choosing a model.

2. Search the Hub — Compare task, language, model size, license, intended use, limitations, and evaluation information.

3. Prototype — Run a small representative test set. Do not judge a model from one impressive example.

4. Build an evaluation set — Create labeled examples that reflect real user inputs, including difficult and ambiguous cases.

5. Measure — Track accuracy or task-specific metrics plus latency, memory use, and failure categories.

6. Package — Save the model/tokenizer or reference a pinned checkpoint. Record software versions and configuration.

7. Deploy — Expose the model through an application or API. Add input validation, timeouts, logging, monitoring, and access control.

8. Review — Monitor drift, model changes, data quality, licenses, and security. Re-evaluate after updates.

Example applications include customer-message classification, support-ticket routing, review sentiment analysis, document triage, search enrichment, and internal knowledge tools. The same engineering loop applies even when the model changes.

TESTING, TROUBLESHOOTING, AND RESPONSIBLE USE

Common problem: model download fails

Check the model ID, internet access, local cache permissions, and whether the model is gated. Some repositories require authentication or accepting terms. Do not put access tokens directly into source code. Use the Hugging Face CLI or environment-based credentials according to the current official documentation.

Common problem: out-of-memory errors

Try a smaller checkpoint, shorter input, CPU inference, or an appropriate lower-memory model. Large models may require specialized hardware and optimized inference techniques.

Common problem: poor predictions

Do not treat the confidence score as proof that a prediction is correct. Build a labeled evaluation set and inspect false positives, false negatives, class imbalance, and examples that differ from the model's training distribution.

Common problem: labels are confusing

Inspect model.config.id2label and the model card. Class IDs and labels are checkpoint-specific. Never assume label 0 always means the same thing across models.

Responsible model selection

Check the model card, dataset information when available, license, intended use, limitations, and evaluation. Avoid deploying a model into high-impact decisions without appropriate domain review, testing, governance, and human oversight.

Security rule

Never expose secrets, private documents, customer data, or authentication tokens to a model or public Hub repository without explicit authorization and appropriate controls. Treat uploaded model and dataset files as software/data artifacts that require review.

CHALLENGES — TAKE PROJECT 23 FURTHER

Complete these in order. Each challenge turns a working example into a stronger engineering project.

1. Add a command-line argument that accepts a model ID instead of using an environment variable.

2. Add batch classification so a CSV column can be processed in one run.

3. Write predictions and probabilities to a new CSV file.

4. Add a small evaluation CSV with expected labels and calculate accuracy.

5. Add a configurable maximum input length and document the trade-off.

6. Add structured logging for model load time and inference time.

7. Build a FastAPI endpoint around the classifier from Project 12.

8. Containerize the API using the Docker workflow from Project 20.

9. Add a model configuration file containing model ID, task name, and version metadata.

10. Create a model card for a model you fine-tune or publish, including intended use, limitations, data, and evaluation.

Extension idea: combine Project 23 with Project 10 semantic search and Project 22 production-style RAG. Retrieve evidence with embeddings, then use a Hugging Face model for a controlled downstream task. Evaluate retrieval and model output separately.

OFFICIAL REFERENCES AND FURTHER READING

The links below are starting points, not a substitute for checking the current documentation for the exact library version you install.

Hugging Face Transformers — Pipelines
https://huggingface.co/docs/transformers/main/en/main_classes/pipelines

Hugging Face Transformers — Auto Classes
https://huggingface.co/docs/transformers/model_doc/auto

Hugging Face Hub — Models
https://huggingface.co/docs/hub/models

Hugging Face Hub — Model Cards
https://huggingface.co/docs/hub/model-cards

Hugging Face Datasets — Documentation
https://huggingface.co/docs/datasets/main/index

Hugging Face Hub — Datasets
https://huggingface.co/docs/hub/datasets

PyTorch — Documentation
https://pytorch.org/docs/stable/

FastAPI — Tutorial
https://fastapi.tiangolo.com/tutorial/

Docker — Get Started
https://docs.docker.com/get-started/

MLflow — Tracking
https://mlflow.org/docs/latest/ml/tracking

Amazon KDP — Paperback Manuscript Formatting
https://kdp.amazon.com/en_US/help/topic/G202145400

Amazon KDP — Paperback Cover Requirements
https://kdp.amazon.com/en_US/help/topic/G201953020

Amazon KDP — Trim Size, Bleed, and Margins
https://kdp.amazon.com/en_US/help/topic/GVBQ3CMEQW3W2VL6

Reference note: documentation URLs are provided for readers to verify current installation commands, API behavior, model availability, licenses, and KDP production requirements.

YOUR NEXT BUILDING ROADMAP

You now have a path from a blank Python folder to a multi-component AI system. Use the projects as building blocks rather than treating them as isolated tutorials.

Foundation — Projects 01–05: environments, data cleaning, supervised learning, evaluation, and model persistence.

Applied ML — Projects 06–08: recommendation, clustering, and neural networks.

Modern NLP — Projects 09–10 and 23: transformers, embeddings, semantic search, and Hugging Face workflows.

Knowledge systems — Projects 11, 18, and 22: retrieval, local knowledge, citations, tenancy, chunking, and evaluation hooks.

Applications — Projects 12–15: APIs, web interfaces, Gradio, and voice interaction.

Agents and quality — Projects 16–17: constrained tools, evaluation, and repeatable tests.

Operations — Projects 19–20: experiment tracking and containers.

Security — Project 21: defensive AI security checks and safer project design.

Capstone mindset — Combine components only after each component works independently. Measure the system at each boundary.

The most important habit in practical AI is not collecting libraries. It is building small, testable systems, measuring what they do, documenting assumptions, and improving them deliberately.

BUILD IT. UNDERSTAND IT. TEST IT. IMPROVE IT. SHIP IT.