AI Engineering Internship

15 Days · NumPy · scikit-learn · PyTorch · LangChain · RAG · FastAPI

NumPy/Pandas | scikit-learn | PyTorch | LangChain | FAISS/RAG | FastAPI | Railway

---

Core2Cloud • core2cloud.in

Practical internship — 1 hour/day — real projects — industry workflow

Contents

DAY 1 · 1 HOUR

AI Setup & Git

Install Python, VS Code, venv — and Git to version your experiments from day one.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

A data scientist at a Bengaluru startup trained a model for 6 hours. Accidentally deleted the script. No Git. **Six hours gone.**

AI work is messy — you try 10 things, keep 2. Git lets you experiment fearlessly. And `venv` means "it works on my machine" stops being an excuse when your model goes to production.

⚙ What You Install Today (5–25 min)

💡 THE 5 TOOLS

Python 3.12 — language + ecosystem for all AI/ML.
VS Code — editor with Jupyter notebook support built-in.
venv — isolated Python per project (no version conflicts).
pip — install numpy, pandas, scikit-learn, torch in one command.
Git — version every experiment so you can always go back.

💻 Live Demo — AI Project Scaffold (25–45 min)

```
>
# Step 1: Create AI project
mkdir my-ai-project
cd my-ai-project

# Step 2: Virtual environment
python -m venv venv
venv\Scripts\activate          # Windows

# Step 3: Install core AI stack
pip install numpy pandas matplotlib scikit-learn

# Step 4: Freeze dependencies
pip freeze > requirements.txt  # like package.json for Python

# Step 5: Git init
git init
git add .
git commit -m "initial AI project setup"
```

🧠 Your First NumPy Program

```
>
# explore.py — verify your stack works
import numpy as np
import pandas as pd

# NumPy: fast array math (the heart of all AI)
scores = np.array([85, 92, 78, 95, 88])
print(f"Mean: {scores.mean():.1f}, Std: {scores.std():.1f}" )

# Pandas: tabular data (CSV, databases, DataFrames)
df = pd.DataFrame({
    "name":  ["Priya", "Ravi", "Ananya"],
    "score": [85, 92, 78]
})
print(df)
```

📋 You Code It (45–55 min)

✍️ TASK

Create `stats.py` : make a list of 5 exam scores, print mean, max, min using NumPy. Then make a DataFrame with name + score columns and print it.

⚡ DAY 1 CHEAT SHEET

`python -m venv venv`	Create isolated Python env
`venv\Scripts\activate`	Activate on Windows
`pip install numpy pandas`	Install AI stack
`pip freeze > requirements.txt`	Save dependencies
`np.array([1,2,3])`	Create NumPy array
`arr.mean() / arr.std()`	Stats on array
`git init && git add .`	Start versioning
`git commit -m "msg"`	Save a snapshot

DAY 2 · 1 HOUR

NumPy & Arrays

Vectorized math, broadcasting, shapes — a 28×28 image is just a NumPy array of 784 numbers.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Every image your phone camera takes is a 3D NumPy array: **height × width × 3 (RGB)**. Instagram's filters? NumPy math on pixels. Face detection? NumPy arrays sliced and convolved.

Python loops over pixels would take **10 seconds** per photo. NumPy vectorization does it in **0.001 seconds**. That's why every AI framework is built on NumPy.

📈 Core Concepts (5–25 min)

💡 THE 3 NUMPY IDEAS

Shape — dimensions of the array: (28, 28) = 28 rows, 28 cols.
Vectorization — operations apply to ALL elements at once (no loop needed).
Broadcasting — arrays with different shapes can combine: (3,) + (3,1) works automatically.

💻 Live Demo (25–45 min)

```
>import numpy as np

# 1. Shapes — think of arrays as tensors
pixel_row = np.array([255, 128, 0, 64, 200])   # 1D: shape (5,)
image     = np.zeros((28, 28))                 # 2D: 28x28 grayscale
rgb_img   = np.zeros((28, 28, 3))              # 3D: 28x28 colour
print(image.shape, image.dtype)                 # (28, 28) float64

# 2. Vectorization — no loop, math on all at once
scores = np.array([45, 78, 92, 60, 55])
normalized = (scores - scores.min()) / (scores.max() - scores.min())
print(normalized)                               # all values in [0.0, 1.0]

# 3. Broadcasting — align shapes automatically
# Subtract mean from each row (centering data)
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
row_means = data.mean(axis=1, keepdims=True)   # shape (3,1)
centered  = data - row_means                   # broadcasts: (3,3) - (3,1)

# 4. Indexing and slicing
big = np.arange(100).reshape(10, 10)
patch = big[2:5, 3:7]                          # rows 2–4, cols 3–6
mask  = scores > 60                            # boolean mask
print(scores[mask])                             # [78 92]
```

> 🌎 **REAL-WORLD**
>
> When you load an image with PIL/OpenCV, it becomes a NumPy array. Resize = array slice + interpolation. Brightness = array multiply. Grayscale = weighted sum across axis 2.

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Create a (5, 4) random array of student test scores (0–100). Compute: row-wise mean (per student), column-wise mean (per subject), then normalize the whole matrix to [0, 1].

DAY 2 CHEAT SHEET

`np.array([...])`	Create array from list
`arr.shape / arr.dtype`	Dimensions and type
`np.zeros((m,n))`	m×n zero matrix
`np.random.rand(m,n)`	Random floats [0,1)
`arr.mean(axis=0)`	Column-wise mean
`arr.reshape(r, c)`	Change shape (same data)
`arr[mask]`	Boolean indexing
`arr1 @ arr2`	Matrix multiplication

DAY 3 · 1 HOUR

Pandas & EDA

DataFrames, CSV loading, groupby — understand your data *before* you build any model.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

A Swiggy data scientist built a delivery-time model that predicted negative times. The bug? One city's distances were stored in miles, all others in km. **The model learned nonsense.**

Exploratory Data Analysis (EDA) — looking at your data before modelling — would have caught it in 2 minutes. Bad data beats good models every time.

📈 Core Concepts (5–25 min)

💡 PANDAS IN 4 OPERATIONS

load — `pd.read_csv()` turns a file into a DataFrame.
explore — `df.describe()`, `df.info()`, `df.isnull().sum()`.
filter & select — `df[df["score"] > 80]`, `df[["name","score"]]`.
aggregate — `df.groupby("city")["sales"].mean()`.

💻 Live Demo — Student Grade EDA (25–45 min)

```
>import pandas as pd
import numpy as np

# Create a sample dataset (in real life: pd.read_csv("grades.csv"))
df = pd.DataFrame({
    "name":    ["Priya", "Ravi", "Ananya", "Kiran", "Divya"],
    "city":    ["Chennai", "Mumbai", "Chennai", "Mumbai", "Bengaluru"],
    "math":    [88, 72, 95, 60, 85],
    "science": [76, 85, 90, 55, 92],
    "grade":   ["B", "B", "A", "C", "A"]
})

# 1. Quick overview
print(df.describe())          # mean, std, min, max per column
print(df.isnull().sum())    # missing values per column

# 2. Filter: students who passed both subjects
passed = df[(df["math"] ≥ 60) & (df["science"] ≥ 60)]
print(passed[["name", "grade"]])

# 3. Groupby: average score by city
city_avg = df.groupby("city")[["math", "science"]].mean().round(1)
print(city_avg)

# 4. Add derived column
df["avg"] = (df["math"] + df["science"]) / 2
df["result"] = df["avg"].apply(lambda x: "Pass" if x ≥ 60 else "Fail")
print(df[["name", "avg", "result"]])
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Add a 6th student with a missing `math` score (use `None`). Find how many nulls exist. Fill the null with the column mean. Then filter students with avg >= 75 and print their names.

DAY 3 CHEAT SHEET

`pd.read_csv("file.csv")`	Load CSV as DataFrame
`df.describe()`	Stats summary
`df.isnull().sum()`	Count missing per column
`df.fillna(df.mean())`	Fill nulls with mean
`df[df["col"] > 60]`	Filter rows
`df.groupby("col").mean()`	Aggregate by group
`df["new"] = df["a"] + df["b"]`	Derived column
`df.sort_values("col")`	Sort DataFrame

DAY 4 · 1 HOUR

Data Visualization

matplotlib + seaborn — a chart that tells the story in 1 second that a table takes 5 minutes to read.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

NASA engineers had data showing O-ring failures at low temperatures before the Challenger disaster. The data existed — but it was in a table. When someone finally plotted failures vs. temperature the night before launch, the pattern was unmistakable. **They didn't have time to act.**

A good chart surfaces what tables hide. In ML, visualization catches distribution shifts, outliers, and class imbalance before they ruin your model.

📈 Core Chart Types (5–25 min)

Chart	Use when	Example
`plt.hist()`	See distribution shape	Score distribution
`plt.scatter()`	Relationship between 2 variables	Height vs weight
`plt.bar()`	Compare categories	Sales by city
`plt.plot()`	Trends over time	Model loss vs epoch
`sns.heatmap()`	Correlation matrix	Feature correlations
`sns.boxplot()`	Distribution + outliers	Salary by role

💻 Live Demo — AI Training Dashboard (25–45 min)

```
>import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

# Simulate model training history
epochs = np.arange(1, 21)
train_loss = 1.0 * np.e ** (-0.15 * epochs) + np.random.normal(0, .02, 20)
val_loss   = 1.0 * np.e ** (-0.12 * epochs) + np.random.normal(0, .03, 20)

# 1. Training curves (most important AI chart)
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(epochs, train_loss, label="Train Loss", color="#7c3aed")
plt.plot(epochs, val_loss,   label="Val Loss",   color="#f59e0b", linestyle="--")
plt.xlabel("Epoch"); plt.ylabel("Loss")
plt.title("Training Curves"); plt.legend()

# 2. Feature correlation heatmap
df = pd.DataFrame(np.random.randn(50, 4),
                  columns=["age", "income", "score", "churn"])
plt.subplot(1, 2, 2)
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Feature Correlations")

plt.tight_layout()
plt.savefig("dashboard.png", dpi=150)
print("Saved dashboard.png")
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Plot a histogram of 500 random exam scores (normal distribution, mean=70, std=12). Add a vertical red line at mean. Title it "Score Distribution". Save as `scores.png`.

DAY 4 CHEAT SHEET

`plt.figure(figsize=(w,h))`	Set canvas size
`plt.subplot(r, c, i)`	Multi-chart layout
`plt.hist(data, bins=20)`	Histogram
`plt.scatter(x, y)`	Scatter plot
`plt.axvline(x=mean)`	Vertical reference line
`sns.heatmap(df.corr())`	Correlation heatmap
`plt.savefig("out.png")`	Save to file
`plt.tight_layout()`	Fix overlapping labels

DAY 5 · 1 HOUR

ML Fundamentals

scikit-learn, train/test split, linear regression — predict house prices from scratch.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

MagicBricks and 99acres don't have humans pricing every listing. A regression model looks at area, location, bedrooms, age — and outputs a price estimate in milliseconds.

The same math Gauss used in 1809 to predict asteroid orbits. Today it prices 10 million homes per day. **That's ML in production.**

📈 The ML Workflow (5–25 min)

💡 EVERY ML PROJECT HAS 5 STEPS

1. **Load data** — CSV, database, API.
2. **Split** — 80% train, 20% test. Never evaluate on training data.
3. **Train** — `model.fit(X_train, y_train)`.
4. **Evaluate** — RMSE, R^2 on the *test* set.
5. **Predict** — `model.predict(new_data)`.

⚠ OVERFITTING

A model that memorises training data but fails on new data. Like a student who copies answers for last year's exam but can't solve new problems. Always check test score vs train score.

💻 Live Demo — House Price Predictor (25–45 min)

```
>import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Synthetic house price dataset
np.random.seed(42)
n = 200
area    = np.random.randint(600, 3000, n)           # sq ft
rooms   = np.random.randint(1, 6, n)
age     = np.random.randint(0, 30, n)              # years
price   = 2000*area + 50000*rooms - 3000*age + np.random.normal(0, 80000, n)

df = pd.DataFrame({"area": area, "rooms": rooms,
                   "age": age,  "price": price})

# 1. Split
X = df[["area", "rooms", "age"]]
y = df["price"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 2. Train
model = LinearRegression()
model.fit(X_train, y_train)

# 3. Evaluate
y_pred = model.predict(X_test)
rmse   = mean_squared_error(y_test, y_pred) ** 0.5
r2     = r2_score(y_test, y_pred)
print(f"RMSE: ₹{rmse:,.0f}  R²: {r2:.3f}" )  # R² close to 1.0 is good

# 4. Predict a new house
new_house = [[1800, 3, 5]]           # 1800 sqft, 3 rooms, 5 yr old
est_price = model.predict(new_house)[0]
print(f"Estimated price: ₹{est_price:,.0f}" )
```

📋 You Code It (45–55 min)

✍️ **TASK**

Add a `location_score` feature (1–5) to the dataset. Re-train. Does R^2 improve? Print the model's coefficients with `model.coef_` to see which feature matters most.

⚡ DAY 5 CHEAT SHEET

`train_test_split(X, y, test_size=0.2)`	80/20 split
`model.fit(X_train, y_train)`	Train the model
`model.predict(X_test)`	Generate predictions
`r2_score(y_test, y_pred)`	R^2: 1.0 = perfect
`mean_squared_error(...)**0.5`	RMSE: error in same units
`model.coef_`	Feature weights
`model.intercept_`	Bias term
`StandardScaler().fit_transform(X)`	Normalize features

DAY 6 · 1 HOUR

Classification

Decision Tree, Random Forest, confusion matrix — build a spam filter that learns by example.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

Gmail's spam filter processes 15 billion emails per day. It doesn't use a list of bad words — it uses a Random Forest (and more). It learns from millions of examples: *this email = spam, this = not*.

Classification is ML's answer to yes/no questions: spam or not, fraud or not, cancer or not. The stakes are real — a false negative (missed fraud) costs money; a false positive (blocked real email) costs trust.

📈 Key Metrics (5–25 min)

Metric	Formula	When it matters
Accuracy	(TP+TN) / total	Balanced classes
Precision	TP / (TP+FP)	Cost of false alarms (spam filter)
Recall	TP / (TP+FN)	Cost of misses (cancer detection)
F1	2×P×R / (P+R)	Imbalanced classes

🧠 **WHY RANDOM FOREST?**

One decision tree overfits. Random Forest grows 100 trees on random data subsets, then votes. Like asking 100 doctors — majority rules. Much harder to fool. This is **ensemble learning**.

💻 Live Demo — Spam Filter (25–45 min)

```
>import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

# Features: [word_count, exclamation_marks, links, caps_ratio, has_unsubscribe]
np.random.seed(0)
n_spam = 300
n_ham  = 700

spam_X = np.column_stack([
    np.random.randint(20, 200, n_spam),    # word_count
    np.random.randint(3, 20, n_spam),     # exclamation_marks
    np.random.randint(2, 15, n_spam),     # links
    np.random.uniform(.3, .9, n_spam),    # caps_ratio
    np.random.randint(0, 2, n_spam)       # has_unsubscribe
])
ham_X = np.column_stack([
    np.random.randint(50, 500, n_ham),
    np.random.randint(0, 3, n_ham),
    np.random.randint(0, 3, n_ham),
    np.random.uniform(.0, .2, n_ham),
    np.zeros(n_ham, dtype=int)
])

X = np.vstack([spam_X, ham_X])
y = np.array([1]*n_spam + [0]*n_ham)   # 1=spam, 0=ham

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=["ham", "spam"]))
print("Confusion matrix:")
print(confusion_matrix(y_test, y_pred))

# Feature importance: which signal matters most?
features = ["word_count", "exclamations", "links", "caps_ratio", "unsubscribe"]
```

```
for f, imp in zip(features, model.feature_importances_):
    print(f"  {f:20s}: {imp:.3f}" )
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Change `n_estimators` to 10, 50, 200. Print F1 score for each. Does more trees always help? Also try a single `DecisionTreeClassifier` — how does its accuracy compare to the forest?

⚡ DAY 6 CHEAT SHEET

`RandomForestClassifier(n_estimators=100)`	100-tree forest
`model.fit(X_train, y_train)`	Train classifier
`model.predict(X_test)`	Class predictions
`model.predict_proba(X)`	Probability per class
`classification_report(y, yhat)`	Precision/recall/F1
`confusion_matrix(y, yhat)`	TP/TN/FP/FN grid
`model.feature_importances_`	Which features matter
`cross_val_score(model, X, y)`	K-fold CV score

DAY 7 · 1 HOUR

Neural Networks

Perceptron, layers, activation functions — teach a network to recognise handwritten digits.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

India Post processes 100 million letters per year. Sorting by hand-reading ZIP codes = slow and error-prone. A neural network trained on MNIST-style digit images reads PINs with 99.7% accuracy — faster than any human.

Neural networks are the engine behind voice assistants, cancer detection, Google Translate, and ChatGPT. Today you build one from scratch.

📈 How a Neural Network Learns (5–25 min)

💡 **FORWARD → LOSS → BACKWARD → UPDATE**

Forward pass — input flows through layers, each neuron: `output = activation(weights · input + bias)`.
Loss — how wrong is the prediction? Cross-entropy for classification.
Backward pass — chain rule computes gradient of loss w.r.t. every weight.
Optimizer — SGD/Adam nudges weights in the direction that reduces loss.

💻 Live Demo — MNIST Digit Classifier (25–45 min)

```
>import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1. Load MNIST (28x28 grayscale images, 10 digit classes)
transform = transforms.Compose([transforms.ToTensor()])
train_ds   = datasets.MNIST("./data", train=True,  download=True,
transform=transform)
test_ds    = datasets.MNIST("./data", train=False, download=True,
transform=transform)
train_dl   = DataLoader(train_ds, batch_size=64, shuffle=True)

# 2. Define network: 784 → 256 → 128 → 10
class DigitNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),                # 28x28 → 784
            nn.Linear(784, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 10)           # 10 digit classes
        )
    def forward(self, x):
        return self.net(x)

model     = DigitNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# 3. Train for 3 epochs
for epoch in range(3):
    total_loss = 0
    for images, labels in train_dl:
        optimizer.zero_grad()
        loss = criterion(model(images), labels)
        loss.backward()
        optimizer.step()
```

```
        total_loss += loss.item()
    print(f"Epoch {epoch+1} loss: {total_loss/len(train_dl):.4f}" )
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Add a `Dropout(0.3)` layer after each ReLU. Run for 5 epochs. Does validation loss diverge less? Add an evaluation loop that prints test accuracy after training.

⚡ DAY 7 CHEAT SHEET

`nn.Linear(in, out)`	Fully-connected layer
`nn.ReLU()`	Activation (max(0,x))
`nn.Sequential(...)`	Stack layers in order
`loss.backward()`	Compute gradients
`optimizer.step()`	Update weights
`optimizer.zero_grad()`	Clear old gradients first
`nn.Dropout(0.3)`	Randomly zero 30% neurons
`model.eval()`	Switch off dropout/BN

DAY 8 · 1 HOUR

NLP Basics

Tokenization, TF-IDF, sentiment analysis — turn raw text into numbers a model can learn from.

0–5 min	5–25 min	25–45 min	45–55 min	55–60 min
Real Hook	Core Concept	Live Coding	You Code It	Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

Flipkart processes 2 million product reviews per day. A human reading all of them? Impossible. An NLP pipeline classifies each review as positive/negative/neutral in real time, feeds the insight to sellers automatically.

Every chatbot, email classifier, autocomplete, and LLM is built on the same foundational NLP primitives you'll learn today.

📈 NLP Pipeline (5–25 min)

💡 RAW TEXT → NUMBERS

Tokenize — split text into words/tokens.
Clean — lowercase, remove punctuation, stop words.
Vectorize — TF-IDF, Bag-of-Words, or embeddings → numeric matrix.
Model — feed matrix to classifier (Naive Bayes, Logistic Regression, NN).

💻 Live Demo — Review Sentiment Classifier (25–45 min)

```
>import re
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Sample product reviews
reviews = [
    "Excellent product, very happy with the quality!",
    "Worst purchase ever. Broke in 2 days.",
    "Good value for money, works as expected.",
    "Terrible quality, do not buy this.",
    "Amazing! Fast delivery and perfect condition.",
    "Disappointed. Not as described in listing.",
    "Superb build quality, highly recommend!",
    "Stopped working after a week. Waste of money.",
    "Average product. Nothing special about it.",
    "Brilliant! Exactly what I needed. Very satisfied.",
]
labels = [1, 0, 1, 0, 1, 0, 1, 0, 0, 1]   # 1=positive, 0=negative

# Clean text
def clean(text):
    text = text.lower()
    text = re.sub(r"[^a-z\s]", "" " , text)
    return text

clean_reviews = [clean(r) for r in reviews]

# TF-IDF: how important is a word to this doc vs the corpus?
vectorizer = TfidfVectorizer(stop_words="english", max_features=500)
X = vectorizer.fit_transform(clean_reviews)

X_tr, X_te, y_tr, y_te = train_test_split(X, labels, test_size=0.3, random_state=42)
clf = LogisticRegression()
clf.fit(X_tr, y_tr)
print(classification_report(y_te, clf.predict(X_te)))

# Predict new review
new = ["Really bad experience, total waste of money!"]
```

```
vec = vectorizer.transform(new)
print(f"Sentiment: {'Positive' if clf.predict(vec)[0]==1 else 'Negative'}" )
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Add 5 more reviews to the dataset (mix of positive/negative). Print the top 10 TF-IDF features by weight using `vectorizer.get_feature_names_out()` and `clf.coef_`. Which words most indicate spam?

⚡ DAY 8 CHEAT SHEET

`TfidfVectorizer(stop_words="english")`	Text → TF-IDF matrix
`vectorizer.fit_transform(texts)`	Learn vocab + transform
`vectorizer.transform(new_texts)`	Transform new texts
`vectorizer.get_feature_names_out()`	List of vocabulary words
`re.sub(r"[^a-z]", " ", text)`	Remove non-alpha chars
`text.lower()`	Normalize case
`clf.coef_[0]`	Word importance weights
`np.argsort(coef)[-10:]`	Top 10 features

DAY 9 · 1 HOUR

LLM & Prompt Engineering

Groq API, system prompts, temperature — talk to an LLM the way engineers do, not users.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

Two people ask ChatGPT to write a Python function. One gets a 10-line vague answer. The other gets a production-ready function with docstring, type hints, and edge case handling. **Same model. Different prompt.**

Prompt engineering is the new skill gap. Companies pay ₹15–25 LPA for engineers who know how to instruct LLMs reliably. Today you learn the difference.

📈 Prompt Engineering Principles (5–25 min)

Technique	What it does	Example
System prompt	Sets personality & constraints	"You are a senior Python developer..."
Few-shot	Show examples before the task	Input/output pairs
Chain-of-thought	Ask for reasoning steps	"Think step by step"
Temperature 0	Deterministic output	Code, JSON, facts
Temperature 0.7+	Creative output	Stories, brainstorming

💻 Live Demo — Code Review Bot (25–45 min)

```
>from groq import Groq
import os

client = Groq(api_key=os.getenv("GROQ_API_KEY"))

# System prompt defines the LLM's role and behaviour
SYSTEM = "You are a senior Python code reviewer at a top tech company.
Review code for: correctness, PEP8 style, edge cases, performance.
Be specific. Rate each issue: Critical / Warning / Suggestion.
Keep response under 200 words."

code_to_review = "
def get_user(users, id):
    for u in users:
        if u["id"] == id:
            return u
    return None
"

response = client.chat.completions.create(
    model="llama3-8b-8192",
    temperature=0,                          # deterministic for code review
    messages=[
        {"role": "system", "content": SYSTEM},
        {"role": "user",   "content": f"Review this Python code:\n{code_to_review}" }
    ]
)
print(response.choices[0].message.content)

# Few-shot: teach the model the exact output format
few_shot_msgs = [
    {"role": "user",      "content": "Classify: Free iPhone click here!!!"},
    {"role": "assistant", "content": "SPAM"},
    {"role": "user",      "content": "Classify: Meeting at 3pm tomorrow"},
    {"role": "assistant", "content": "HAM"},
    {"role": "user",      "content": "Classify: Congratulations you won a prize!"},
]
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Write a prompt that converts a plain English description into a Python function signature with type hints. Test it on: "a function that takes a list of names and returns them sorted with duplicates removed". Try temperature 0 vs 0.9 — observe the difference.

⚡ DAY 9 CHEAT SHEET

`Groq(api_key=os.getenv(...))`	Create Groq client
`client.chat.completions.create()`	Call the LLM
`messages=[{role, content}]`	Conversation turns
`role: "system"`	Set personality/constraints
`temperature=0`	Deterministic output
`temperature=0.7`	Creative output
`response.choices[0].message.content`	Get text response
`max_tokens=200`	Limit response length

DAY 10 · 1 HOUR

LangChain

Chains, prompt templates, output parsers — build a structured Q&A bot in 30 lines.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

A raw LLM API call gives you text. But real AI products need: structured JSON output, conversation memory, tool calls, retries on failure, streaming responses.

LangChain is the plumbing that connects an LLM to the rest of your application. Like Django for web — you could build everything from scratch, but why?

📈 LangChain Building Blocks (5–25 min)

💡 THE 4 CORE PRIMITIVES

PromptTemplate — parameterized prompt with `{variables}`.
LLM / ChatModel — the model call (ChatGroq, ChatOpenAI, etc.).
OutputParser — coerce raw text into Python objects (str, JSON, Pydantic).
Chain (|) — pipe them together: `prompt | llm | parser`.

💻 Live Demo — Course Q&A Bot (25–45 min)

```
>from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.messages import HumanMessage, SystemMessage
import os

llm = ChatGroq(
    model="llama3-8b-8192",
    api_key=os.getenv("GROQ_API_KEY"),
    temperature=0.3
)

# 1. Simple chain: prompt | llm | parser
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an AI tutor for Core2Cloud internship students.
         Answer clearly in 3 sentences max. Always give a code example."),
    ("human",  "{question}")
])
parser = StrOutputParser()
chain  = prompt | llm | parser

answer = chain.invoke({"question": "What is gradient descent?"})
print(answer)

# 2. Structured output with Pydantic
from pydantic import BaseModel
from langchain_core.output_parsers import JsonOutputParser

class ConceptCard(BaseModel):
    concept:     str
    one_line:    str
    analogy:     str
    code_example: str

parser2 = JsonOutputParser(pydantic_object=ConceptCard)
prompt2 = ChatPromptTemplate.from_template(
    "Explain the ML concept: {concept}\n{format_instructions}"
)
chain2 = prompt2 | llm | parser2
card = chain2.invoke({
    "concept": "Overfitting",
    "format_instructions": parser2.get_format_instructions()
```

```
})
print(card)
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Build a `ConversationChain` that remembers context. Ask it: "My name is Priya", then in the next turn ask "What is my name?" — it should recall. Use `ChatMessageHistory` from LangChain.

⚡ DAY 10 CHEAT SHEET

`ChatPromptTemplate.from_messages()`	Multi-role prompt
`prompt | llm | parser`	LCEL chain (pipe syntax)
`chain.invoke({"var": val})`	Run chain with inputs
`chain.stream({...})`	Stream tokens as they arrive
`StrOutputParser()`	Raw text output
`JsonOutputParser(pydantic_object=M)`	Typed JSON output
`format_instructions`	Tell LLM how to format
`ChatMessageHistory()`	Conversation memory

DAY 11 · 1 HOUR

Vector DB & RAG

FAISS, embeddings, retrieval-augmented generation — give the LLM your own documents.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

ChatGPT's knowledge cuts off at a date. It doesn't know your company's internal docs, last week's product update, or your student handbook.

RAG (Retrieval-Augmented Generation) fixes this: embed your documents, store in a vector DB, retrieve the relevant chunks at query time, inject them into the prompt. The LLM answers with YOUR data. Every enterprise AI chatbot works this way.

📈 How RAG Works (5–25 min)

💡 4-STEP RAG PIPELINE

1. **Chunk** — split documents into 500-token chunks with overlap.
2. **Embed** — convert each chunk to a 768-dim vector (semantic meaning as numbers).
3. **Store** — put vectors in FAISS (fast approximate nearest-neighbour index).
4. **Retrieve & Generate** — embed query, find top-k similar chunks, send to LLM with question.

💻 Live Demo — Course Notes Q&A (25–45 min)

```
>from langchain_community.vectorstores import FAISS
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate
import os

# Sample knowledge base (your company docs, textbooks, etc.)
docs = [
    "Linear regression finds the best-fit line through data points
    by minimising the sum of squared errors (least squares method).",
    "Random Forest is an ensemble of decision trees. Each tree trains
    on a random subset of data and features. Final prediction = majority vote.",
    "Overfitting occurs when a model memorises training data but
    generalises poorly to unseen data. Fix with regularisation or more data.",
    "Gradient descent updates model weights by stepping in the
    direction of steepest loss decrease. Learning rate controls step size.",
    "Neural networks stack layers of neurons. Each layer learns
    increasingly abstract features. Deep = many layers.",
]

# 1. Chunk documents
splitter = RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=50)
chunks   = splitter.create_documents(docs)

# 2. Embed and store in FAISS
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
vectorstore = FAISS.from_documents(chunks, embeddings)

# 3. Build retriever + LLM
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})
llm = ChatGroq(model="llama3-8b-8192", api_key=os.getenv("GROQ_API_KEY"),
temperature=0)

# 4. RAG chain
prompt = ChatPromptTemplate.from_template("
Answer using ONLY the context below. If unsure, say "I don't know."

Context: {context}
Question: {question}
")
```

```python
def rag_answer(question):
    context_docs = retriever.invoke(question)
    context = "\n\n".join(d.page_content for d in context_docs)
    chain = prompt | llm
    return chain.invoke({"context": context, "question": question}).content

print(rag_answer("How does random forest prevent overfitting?"))
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Add 3 more documents about PyTorch, transformers, and LangChain. Ask a question that requires combining two docs. Change `k=2` to `k=3` — does the answer improve? Print which chunks were retrieved.

⚡ DAY 11 CHEAT SHEET

`HuggingFaceEmbeddings(model_name=...)`	Free local embeddings
`FAISS.from_documents(chunks, emb)`	Build vector index
`vectorstore.as_retriever(k=3)`	Top-3 similar chunks
`vectorstore.save_local("db")`	Persist index to disk
`FAISS.load_local("db", emb)`	Reload saved index
`RecursiveCharacterTextSplitter`	Smart chunking with overlap
`retriever.invoke(question)`	Get relevant docs
`chunk_overlap=50`	Overlap prevents context loss

DAY 12 · 1 HOUR

AI Agents

Tool use, ReAct loop — an agent that searches, calculates, and makes decisions autonomously.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

GitHub Copilot doesn't just autocomplete — it can read your entire repo, run tests, write a fix, and open a PR. **That's an AI agent.**

Agents are LLMs that can USE TOOLS: search the web, run code, read files, call APIs, query databases. They decide which tool to call, call it, observe the result, and repeat until the task is done. ReAct = Reason + Act.

📈 The ReAct Loop (5–25 min)

💡 THOUGHT → ACTION → OBSERVATION → REPEAT

Thought — LLM reasons about what to do next.
Action — LLM calls a tool (calculator, search, Python REPL).
Observation — tool returns a result.
Final Answer — LLM synthesises all observations into a response.

💻 Live Demo — Research Agent (25–45 min)

```
>from langchain_groq import ChatGroq
from langchain.agents import create_react_agent, AgentExecutor
from langchain_core.tools import tool
from langchain import hub
import os, math

llm = ChatGroq(model="llama3-8b-8192", api_key=os.getenv("GROQ_API_KEY"),
temperature=0)

# Define tools the agent can use
@tool
def calculator(expression: str) -> str:
    "Evaluate a mathematical expression. Input: a Python math expression string."
    try:
        result = eval(expression, {"__builtins__": {}}, vars(math))
        return f"Result: {result}"
    except Exception as e:
        return f"Error: {e}"

@tool
def unit_converter(value_and_units: str) -> str:
    "Convert units. Input format: "100 km to miles" or "5 kg to pounds"."
    parts = value_and_units.split()
    if len(parts) < 4:
        return "Invalid format. Use: "100 km to miles""
    val, from_u, _, to_u = parts[0], parts[1].lower(), parts[2], parts[3].lower()
    conversions = {("km","miles"): 0.621371, ("kg","pounds"): 2.20462,
                   ("celsius","fahrenheit"): None}
    if (from_u, to_u) in conversions:
        factor = conversions[(from_u, to_u)]
        if factor:
            return f"{float(val) * factor:.4f} {to_u}"
        else:
            return f"{float(val) * 9/5 + 32:.1f} fahrenheit"
    return "Conversion not supported"

tools = [calculator, unit_converter]

# Load standard ReAct prompt from LangChain hub
react_prompt = hub.pull("hwchase17/react")
agent = create_react_agent(llm, tools, react_prompt)
executor = AgentExecutor(agent=agent, tools=tools, verbose=True, max_iterations=5)
```

```
result = executor.invoke({
    "input": "If I run 42.2 km (a marathon) and burn 2800 calories,
            how many calories per mile is that?"
})
print(result["output"])
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Add a `word_counter` tool that counts words in a string. Ask the agent: "Count the words in this sentence and tell me the square root of that number: The quick brown fox jumps over the lazy dog."

⚡ DAY 12 CHEAT SHEET

`@tool`	Decorator to register a tool
`create_react_agent(llm, tools, prompt)`	Build ReAct agent
`AgentExecutor(agent, tools)`	Run loop with tools
`executor.invoke({"input": q})`	Ask the agent
`verbose=True`	See Thought/Action/Obs
`max_iterations=5`	Prevent infinite loops
`hub.pull("hwchase17/react")`	Standard ReAct prompt
`tool docstring`	LLM reads this to know when to use the tool

DAY 13 · 1 HOUR

FastAPI for AI

Serve your ML model as a REST API — from Jupyter notebook to production endpoint.

0–5 min Real Hook	5–25 min Core Concept	25–45 min Live Coding	45–55 min You Code It	55–60 min Cheat Sheet

🔥 **REAL-WORLD HOOK (0–5 MIN)**

A model that lives in a Jupyter notebook helps only you. A model behind a FastAPI endpoint can be called by a mobile app in Chennai, a web dashboard in Mumbai, and a batch job in Bengaluru — all at the same time.

Every ML model in production runs behind an API. FastAPI is Python's fastest, with automatic docs, type validation via Pydantic, and async support. Netflix, Uber, and Microsoft use it.

📈 FastAPI Core Concepts (5–25 min)

💡 **WHY FASTAPI OVER FLASK?**

Auto docs — Swagger UI at `/docs`, Redoc at `/redoc`. Zero config.
Pydantic validation — wrong input type = 422 error with a clear message, not a crash.
Async — handle thousands of concurrent LLM calls without blocking.
Speed — on par with Node.js, 2× faster than Flask for I/O-heavy work.

💻 Live Demo — Sentiment Analysis API (25–45 min)

```
>
```

```
# main.py — run with: uvicorn main:app --reload
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
import pickle, os

app = FastAPI(title="Sentiment API", version="1.0")

# Request / Response schemas (Pydantic validates automatically)
class ReviewRequest(BaseModel):
    text: str
    language: str = "en"

class SentimentResponse(BaseModel):
    sentiment: str         # "positive" | "negative"
    confidence: float
    word_count: int

# Load model at startup (not per request)
vectorizer: TfidfVectorizer = None
model: LogisticRegression   = None

@app.on_event("startup")
async def load_model():
    global vectorizer, model
    if os.path.exists("model.pkl"):
        with open("model.pkl", "rb") as f:
            vectorizer, model = pickle.load(f)
    print("Model loaded")

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

@app.post("/predict", response_model=SentimentResponse)
async def predict(req: ReviewRequest):
    if not model:
        raise HTTPException(status_code=503, detail="Model not loaded")
    vec = vectorizer.transform([req.text])
    pred  = model.predict(vec)[0]
    proba = model.predict_proba(vec)[0].max()
```

```
    return SentimentResponse(
        sentiment  = "positive" if pred == 1 else "negative",
        confidence = round(float(proba), 3),
        word_count = len(req.text.split())
    )
```

📋 You Code It (45–55 min)

> ✍️ **TASK**
>
> Add a `POST /batch` endpoint that accepts a list of reviews and returns a list of `SentimentResponse`. Add a `GET /stats` endpoint that returns total predictions made (use a module-level counter).

⚡ DAY 13 CHEAT SHEET

`app = FastAPI(title="...")`	Create app
`@app.post("/path")`	POST endpoint
`@app.get("/path")`	GET endpoint
`class X(BaseModel)`	Pydantic schema
`response_model=MyModel`	Type-validate output
`HTTPException(status_code=4xx)`	Return error responses
`@app.on_event("startup")`	Run code once at boot
`uvicorn main:app --reload`	Dev server with hot-reload

DAY 14 · 1 HOUR

Deployment

Railway, env vars, Docker basics — your AI API live on the internet in one session.

0–5 min	5–25 min	25–45 min	45–55 min	55–60 min
Real Hook	Core Concept	Live Coding	You Code It	Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

A senior developer once said: "A model that isn't deployed doesn't exist." You can have the world's best AI — if it's sitting on your laptop, it helps nobody.

Deployment used to need a DevOps team. Railway changed that: connect your GitHub repo, set env vars, deploy in 3 minutes. Used by thousands of startups in India.

📈 The Deployment Checklist (5–25 min)

⚠ NEVER COMMIT SECRETS

Your `GROQ_API_KEY` must NEVER go into Git. Use `.env` locally + Railway's env vars panel in production. Add `.env` to `.gitignore` before your first commit.

💻 Live Demo — Deploy FastAPI to Railway (25–45 min)

```
># File structure for deployment
ai-sentiment-api/
├── main.py           # FastAPI app
├── requirements.txt  # pip freeze > requirements.txt
├── Procfile          # tells Railway how to start
├── .env              # local secrets (NEVER commit)
└── .gitignore        # must contain .env
```

```
># Procfile — one line, no file extension
web: uvicorn main:app --host 0.0.0.0 --port $PORT
```

```
># requirements.txt
fastapi==0.111.0
uvicorn==0.30.1
scikit-learn==1.5.0
groq==0.9.0
langchain==0.2.5
langchain-groq==0.1.6
```

```
># .gitignore
.env
__pycache__/
*.pyc
*.pkl
venv/
.DS_Store
```

```
># Step-by-step Railway deploy

# 1. Push to GitHub
git init
git add .
git commit -m "feat: add sentiment API"
git remote add origin https://github.com/yourname/ai-sentiment-api.git
git push -u origin main

# 2. Go to railway.app → New Project → Deploy from GitHub
# 3. Set environment variables in Railway dashboard:
#    GROQ_API_KEY = your_key_here
# 4. Railway auto-detects Procfile and deploys!

# 5. Test your live API
curl -X POST https://your-app.railway.app/predict      -H "Content-Type:
application/json"      -d "{"text": "Amazing product, very happy!"}"
```

> 💡 **DOCKER (OPTIONAL BUT POWERFUL)**
>
> A `Dockerfile` packages your app + Python + dependencies into a container that runs identically anywhere — dev laptop, Railway, AWS, GCP. Railway can auto-detect a `Dockerfile` if you provide one.

```
>
```

```dockerfile
# Dockerfile
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

📋 You Code It (45–55 min)

✍️ TASK

Add a `GET /` root endpoint returning `{"message": "AI API is running", "version": "1.0"}`. Deploy to Railway. Share the live URL with your cohort.

⚡ DAY 14 CHEAT SHEET

`Procfile`	Start command for Railway
`--host 0.0.0.0 --port $PORT`	Listen on Railway's assigned port
`os.getenv("GROQ_API_KEY")`	Read secret from env
`.gitignore: .env`	Never commit secrets
`pip freeze > requirements.txt`	Lock dependencies
`FROM python:3.12-slim`	Minimal Docker base image
`docker build -t app .`	Build image locally
`docker run -p 8000:8000 app`	Run container locally

DAY 15 · 1 HOUR

Capstone & Industry Sim

Full AI pipeline + pytest + git-flow — ship like a real AI engineer on day one.

0–5 min	5–25 min	25–45 min	45–55 min	55–60 min
Real Hook	Core Concept	Live Coding	You Code It	Cheat Sheet

🔥 REAL-WORLD HOOK (0–5 MIN)

At a Bengaluru AI startup, a new hire's first task was: add a confidence threshold to the sentiment API so low-confidence predictions return a "needs review" flag. They had to write tests, open a PR, pass CI, get review, merge.

Not write code. **Ship code.** Git-flow, PR, tests, review, merge — today you do it all.

🛠 Capstone Project: SmartReview API (5–25 min)

🧠 WHAT YOU BUILD

A production-ready sentiment analysis API with:
• Confidence threshold — low-confidence = "uncertain" label
• Language detection — reject non-English input
• Batch endpoint — analyse up to 50 reviews at once
• pytest suite — 6 tests covering happy path + edge cases
• GitHub PR — feature branch → PR → merge

💻 Full Capstone Code (25–45 min)

```
>
```

```python
# smart_review/api.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
from typing import List
import pickle, re

app = FastAPI(title="SmartReview API", version="1.0")
CONFIDENCE_THRESHOLD = 0.70          # below = uncertain

class ReviewIn(BaseModel):
    text: str = Field( ... , min_length=3, max_length=1000)

class ReviewOut(BaseModel):
    sentiment:  str    # positive | negative | uncertain
    confidence: float
    word_count: int

def analyse(text: str, vectorizer, model) -> ReviewOut:
    vec   = vectorizer.transform([text])
    pred  = model.predict(vec)[0]
    proba = model.predict_proba(vec)[0].max()
    if proba < CONFIDENCE_THRESHOLD:
        label = "uncertain"
    else:
        label = "positive" if pred == 1 else "negative"
    return ReviewOut(sentiment=label, confidence=round(float(proba), 3),
                     word_count=len(text.split()))

@app.post("/predict", response_model=ReviewOut)
def predict(req: ReviewIn):
    return analyse(req.text, vectorizer, model)

@app.post("/batch", response_model=List[ReviewOut])
def batch_predict(reviews: List[ReviewIn]):
    if len(reviews) > 50:
        raise HTTPException(400, "Max 50 reviews per batch")
    return [analyse(r.text, vectorizer, model) for r in reviews]
```

```
>
```

```python
# tests/test_api.py
from fastapi.testclient import TestClient
from smart_review.api import app

client = TestClient(app)

def test_positive_review():
    r = client.post("/predict", json={"text": "Excellent! Highly recommend this product."})
    assert r.status_code == 200
    assert r.json()["sentiment"] in ["positive", "uncertain"]

def test_negative_review():
    r = client.post("/predict", json={"text": "Terrible, waste of money, very bad."})
    assert r.status_code == 200

def test_text_too_short():
    r = client.post("/predict", json={"text": "ok"})
    assert r.status_code == 422           # Pydantic min_length=3

def test_batch_limit():
    big_batch = [{"text": "great product"}] * 51
    r = client.post("/batch", json=big_batch)
    assert r.status_code == 400

def test_confidence_field():
    r = client.post("/predict", json={"text": "Amazing quality, super fast delivery!"})
    data = r.json()
    assert 0.0 ≤ data["confidence"] ≤ 1.0

def test_word_count():
    text = "This is a five word review"
    r = client.post("/predict", json={"text": text})
    assert r.json()["word_count"] == 6
```

🚀 Git-Flow — Ship It Like a Pro (25–45 min)

```
>
# Industry standard: never commit directly to main

git checkout -b feature/confidence-threshold

# ... write code and tests ...

git add .
git commit -m "feat: add confidence threshold to sentiment API"

git push origin feature/confidence-threshold

# GitHub: Create Pull Request
# PR title: "feat: add confidence threshold"
# Description: What changed, why, how to test
# Request review from a classmate

# After approval: merge to main
git checkout main
git pull
git branch -d feature/confidence-threshold
```

> 🌎 **WHAT YOU'VE BUILT IN 15 DAYS**
>
> NumPy • Pandas • Matplotlib • scikit-learn • PyTorch • NLP pipeline • Groq LLM API • LangChain • RAG with FAISS • AI Agents • FastAPI • Railway deploy • pytest • Git-flow.
>
> That's the full modern AI engineering stack. Put every project on GitHub. You have a portfolio. You're ready.

⚡ DAY 15 CHEAT SHEET — FULL STACK SUMMARY

`numpy / pandas`	Data wrangling
`matplotlib / seaborn`	Visualization
`sklearn`	Classical ML
`torch / nn.Sequential`	Deep learning
`Groq / LangChain`	LLM integration
`FAISS + embeddings`	RAG pipeline
`FastAPI + Pydantic`	Serve model as API
`Railway + Procfile`	Deploy to production

AI Engineering — Real-World Projects

15 portfolio-ready AI projects spanning ML, LLMs, RAG, agents, and production deployment.

HOW TO USE THIS LIST

Pick at least one from each level. The advanced projects (RAG, agents, multi-modal) are exactly what product companies like Sarvam AI, Krutrim, Zepto, and CRED are hiring for. Deploy every project on Railway with a live demo URL.

▲ BEGINNER — BUILD CONFIDENCE (1–2 WEEKS EACH)

PROJECT 01

Spam Email Classifier

Train a Random Forest on a labelled email dataset. Serve predictions via FastAPI. Deploy to Railway. Accuracy >95% target.

scikit-learn | FastAPI | Railway | TF-IDF | + Email dataset (Kaggle)

BEGINNER — 1 week

PROJECT 02

Student Performance Predictor

Regression model on study-hours, attendance, past scores → predict final grade. Interactive prediction form with Chart.js.

scikit-learn | Pandas | FastAPI | NumPy | + Streamlit dashboard

BEGINNER — 1 week

PROJECT 03

Resume Screener

Parse PDF resumes, extract skills with TF-IDF/regex, score against a job description, rank top-5 candidates automatically.

NLP | TF-IDF | scikit-learn | FastAPI

+ PDF parsing (pdfplumber)

BEGINNER

🕐 1–2 weeks

▶ INTERMEDIATE — INDUSTRY-READY (2–3 WEEKS EACH)

PROJECT 04

Customer Support Chatbot

RAG over company FAQ docs. User asks a question, FAISS retrieves relevant chunks, LLM answers with citations. React chat UI.

LangChain | FAISS | Groq | RAG

+ React chat frontend

INTERMEDIATE

🕐 2 weeks

PROJECT 05

Fake News Detector

Fine-tune or embed article text, binary classifier (real/fake). Explain which phrases triggered the flag. FastAPI + browser extension.

NLP | PyTorch | FastAPI | Embeddings

+ Browser extension

INTERMEDIATE

🕐 2–3 weeks

PROJECT 06

Movie Recommender System

Collaborative filtering + content-based hybrid. User rates 5 movies → get 10 recommendations. Cold-start handled with genre preference.

scikit-learn | Pandas | NumPy | FastAPI

+ Matrix factorisation

INTERMEDIATE

🕐 2 weeks

PROJECT 07

YouTube Video Summariser

Input a YouTube URL, fetch transcript via API, chunk and summarise with LLM, output bullet-point summary + 5 key timestamps.

LangChain | Groq | FastAPI | RAG

+ youtube-transcript-api

INTERMEDIATE

🕐 2 weeks

PROJECT 08

Code Review Bot

Paste Python code, LLM reviews for bugs/style/performance, returns structured JSON: issue type, line number, suggestion, severity.

Groq | LangChain | Pydantic | FastAPI

+ GitHub API integration

INTERMEDIATE

🕐 2 weeks

PROJECT 09

Price Prediction Dashboard

Scrape or use dataset (house/car/gold). Train regression + feature importance. Interactive sliders → live prediction. Deployed API.

scikit-learn | Pandas | FastAPI | NumPy

+ Streamlit / Plotly Dash

INTERMEDIATE

🕐 2–3 weeks

★ ADVANCED — PORTFOLIO SHOWCASERS (3–4 WEEKS EACH)

PROJECT 10

Document Intelligence API

Upload PDF (invoice, contract, report). Extract key fields with LLM, answer questions about the document, export structured JSON.

LangChain | FAISS | Groq | RAG | FastAPI

+ OCR (Tesseract)

ADVANCED

🕐 3 weeks

PROJECT 11

SQL Query Generator

Describe what data you want in plain English → LLM generates SQL → runs on your database → returns result table. Handles joins, aggregates.

LangChain | Groq | FastAPI | Pydantic

+ LangChain SQL chain

ADVANCED

🕐 2–3 weeks

PROJECT 12

Crop Disease Detector

CNN trained on PlantVillage dataset. Upload leaf photo → identify disease + treatment advice from LLM. Mobile-friendly FastAPI.

PyTorch | CNN | FastAPI | LangChain

+ Image augmentation

ADVANCED

🕐 3–4 weeks

PROJECT 13

Personal Finance Advisor

RAG over RBI guidelines + income tax rules. Chat interface: ask about 80C deductions, home loan tax, NPS benefits. Never hallucinates rules.

LangChain | FAISS | Groq | RAG | FastAPI

+ PDF rule ingestion

ADVANCED

🕐 3 weeks

PROJECT 14

AI-Powered Study Planner Agent

Input syllabus PDF + exam date. Agent breaks topics into daily plan, generates quiz questions per topic, tracks progress, adjusts schedule.

LangChain Agents | Groq | FAISS | FastAPI | + Calendar API integration

ADVANCED

🕐 3–4 weeks

PROJECT 15

Multi-modal Product Cataloguer

Upload product image + raw description. AI extracts structured fields (name, category, specs, price tier), generates SEO title and tags auto.

PyTorch | LangChain | Groq | FastAPI | Pydantic | + Vision model (CLIP/LLaVA)

ADVANCED

🕐 4 weeks

🏆 Skills You Practice Across All Projects

SCIKIT-LEARN ML Projects 1,2,3,6,9	**FASTAPI SERVING** All projects	**LANGCHAIN + GROQ** Projects 4,5,7,8,10,11,13,14,15
FAISS / RAG Projects 4,10,13,14	**PYTORCH / CNN** Projects 5,12,15	**PYDANTIC SCHEMAS** All API projects
RAILWAY DEPLOYMENT All projects	**PROMPT ENGINEERING** Projects 4,7,8,10,11,13,14	