📅 MONTH 4 — WEEK 15 (Fine-Tuning Transformers)

✅ DAY 1 — Transfer Learning + Fine-Tuning Basics (Super Beginner, Very Detailed)

> **Today's focus:** Understand *exactly* what "fine-tuning a transformer" means before we train anything.
> **Day 2 onwards:** We will start full training using Hugging Face Trainer.

---

0) First, What Are We Trying To Do in Week 15?

🎯 Weekly Goal (in 1 line)

You already know how to **use** pretrained models.

Now you will learn how to **train** them on your own dataset (like IMDB sentiment).

---

1) Key Words You Must Know (Super Simple)

✅ (A) Pretrained Model

Definition:

A model that has already learned language patterns from a *very large* dataset (books, Wikipedia, etc.).

Example:

DistilBERT already understands:

- grammar
- common words
- sentence structure
- context

📌 **Important:** A pretrained model is *not trained specifically* for your dataset (IMDB). It learned general language.

✅ (B) Downstream Task

Definition:

A smaller specific task we want the model to perform after pretraining.

Examples of downstream tasks:

- Sentiment classification (positive/negative)
- Fake news detection
- Spam detection
- Topic classification

✅ (C) Transfer Learning

Definition (easy):

Using a pretrained model's learned knowledge for your new task.

🧠 **Analogy:**

If you already know English, learning "movie review sentiment" is much easier than learning language from scratch.

📌 Transfer learning saves:

- time
- compute
- data requirement

🔹 Quick Notes (after topic)

- Pretrained model = already trained on huge text
- Downstream task = your small specific task
- Transfer learning = reuse pretrained knowledge

---

2) What is Fine-Tuning?

✅ Definition (beginner)

Fine-tuning means:

> Take a pretrained model and train it *a little more* on your dataset so it becomes very good at your task.

So the model goes from:

- **general language understanding**
  to
- **task-specific performance**

🧠 Analogy (very clear)

You know how to write English well.

Now you prepare for an exam where all questions are about movie reviews.

You won't relearn English, you'll **adapt** your knowledge to that exam pattern.

That is fine-tuning.

What exactly changes in the model?

During fine-tuning:

- the model's internal weights update slightly
- the classification head learns your labels

🔹 Quick Notes

- Fine-tuning = small extra training on your dataset
- Makes pretrained model "special" for your task

---

3) Fine-Tuning vs Feature Extraction (VERY IMPORTANT)

This is asked a lot in exams/interviews.

✅ Feature Extraction

Definition:

Use the pretrained model to get embeddings/features, but **do NOT update** its weights.

So:

- transformer is frozen ❄️
- only a small classifier is trained on top

📌 When used:

- small dataset
- low compute
- quick baseline

✅ Fine-Tuning

Definition:

The transformer weights are allowed to update.

So:

- transformer learns task-specific patterns
- usually better performance

Comparison Table

Point	Feature Extraction	Fine-Tuning
Transformer weights	Frozen	Updated
Training time	Low	Medium
Accuracy potential	Medium	Higher
Risk of overfitting	Lower	Higher

Used in Week 15	❌	✅

🔹 **Quick Notes**

- Feature extraction = embeddings only, frozen model
- Fine-tuning = update model weights, better accuracy

---

4) What Happens During Fine-Tuning? (Step-by-Step Pipeline)

Here is the full flow in simple words:

✅ Step 1: Input Text

Example:

> “This movie was amazing!”

✅ Step 2: Tokenizer converts text into numbers

The tokenizer creates:

- input_ids = token IDs
- attention_mask = 1 for real tokens, 0 for padding

✅ Step 3: Model Forward Pass

Model outputs:

- **logits** (raw class scores)

Example logits:

- Negative score: -1.2
- Positive score: 2.5

✅ Step 4: Loss Function checks how wrong prediction is

Loss compares:

- predicted output
- true label

If true label is positive, but model predicted negative strongly → high loss.

✅ Step 5: Backpropagation updates weights

Model adjusts weights to reduce loss next time.

✅ Step 6: Repeat for many batches (epochs)

This happens repeatedly until the model becomes good at your task.

🔹 Quick Notes

- Tokenizer → model → logits → loss → backprop → weight update
- This cycle is training

---

5) What are Logits? (Extremely Important)

✅ Definition

Logits are:

> raw numbers output by the model BEFORE converting to probabilities.

They are not between 0 and 1.

Example:

- logits = [-1.2, 2.5]

Model will choose the bigger one → class 1.

Why logits?

Because training is more stable using logits (math reason).

🔹 Quick Notes

- logits = raw class scores
- larger logit = more likely class

---

6) What is Loss? (Easy)

✅ Definition

Loss is:

a number showing how wrong the model is.

- High loss = very wrong
- Low loss = mostly correct

Real-life analogy

Loss is like **marks deducted**:

- more mistakes → more marks lost → higher loss

In classification we commonly use:

✅ **Cross-Entropy Loss** (Trainer handles this automatically)

🔹 Quick Notes

- Loss trains the model
- Lower loss means model is learning correctly

---

7) Loss vs Accuracy (Beginner Confusion Fix)

✅ Accuracy

How many predictions are correct?

✅ Loss

How wrong the model is (and how confident it is in wrong answers)

Example:

- Model predicts correct label but with low confidence
  → accuracy correct
  → loss may still be high

🔹 Quick Notes

- Training uses **loss**
- Evaluation uses **accuracy/F1**

---

8) What is Overfitting in NLP? (Very Important)

✅ Definition

Overfitting means:

Model performs very good on training data but performs bad on new data.

Why it happens in transformers easily

Transformers are powerful.

Small dataset + too much training → model memorizes training data.

Signs of overfitting

- Training loss keeps decreasing ✅
- Validation loss stops decreasing or increases ❌

🔹 Quick Notes

- Overfitting is common
- Use validation set and early stopping

---

9) Metrics We Will Track in Week 15

✅ Accuracy

Correct predictions / total predictions

✅ F1-score

Balance of precision and recall

Why F1?

- Useful when data is imbalanced

🔹 Quick Notes

- We will track accuracy + F1 in training
- Loss tells learning, metrics tell performance

---

10) Hugging Face Trainer API (Concept Only Today)

Tomorrow we'll code it fully, but here is the idea:

✅ Definition

Trainer API is a tool that automatically handles:

- training loop
- batching
- optimizer
- evaluation
- saving best model

So you do **less manual PyTorch code**.

🧠 Analogy

Instead of cooking from scratch:

- Trainer is like a cooking machine
  You give:
- ingredients (dataset)

- recipe settings (epochs, lr)
  It cooks (trains) and gives output model.

🔹 **Quick Notes**

- Trainer automates training loop
- You still need correct data + settings

---

11) What You Will Build This Week (So You Know The Target)

✅ Fine-tune DistilBERT on IMDB (or your dataset)

✅ Track training loss and validation loss

✅ Evaluate accuracy and F1

✅ Save the fine-tuned model

✅ Reload model and run predictions

✅ Prepare model for app usage

✅ DAY 1 SUMMARY (Week 15)

Today you learned:

- Transfer learning in NLP
- Fine-tuning meaning and pipeline
- Fine-tuning vs feature extraction
- Logits, loss, metrics
- Overfitting signs
- What Trainer API does

🧾 Day 1 Quick Revision Sheet (Exam-ready)

- Transfer learning = reuse pretrained knowledge
- Fine-tuning = update pretrained weights on your dataset
- Feature extraction = freeze model, use embeddings
- Logits = raw class scores
- Loss trains model, metrics evaluate model
- Overfitting = good on train, bad on validation

✅ Mini Self-Test (Day 1 Practice Questions)

1. Define transfer learning in your own words.
2. What is fine-tuning?
3. Difference between fine-tuning and feature extraction?
4. What are logits?
5. Why do we track validation loss?
6. What is overfitting and one sign of it?

DAY 2 — Dataset Preparation & Tokenization for Training (VERY DETAILED, BEGINNER FRIENDLY)

> **Today is the MOST IMPORTANT technical foundation day of Week 15.**
> If dataset + tokenization is wrong → training will FAIL or give bad results.

🎯 Day 2 Goal:

By the end of today, you will **confidently prepare a dataset that is 100% ready for fine-tuning a transformer**.

You will understand:

- What a **training dataset** really means for transformers
- Train / validation / test splits (again, but deeper)
- What **labels** mean and how they are represented
- What **tokenization for training** is (not inference!)
- What max_length, padding, truncation really do
- How Hugging Face **Datasets + Tokenizer** work together
- How to prepare data for the **Trainer API**

- Common beginner mistakes and how to avoid them

Everything is explained **slowly, step-by-step, in very easy language**.

1️⃣ What Does "Dataset Preparation" Mean in Fine-Tuning?

✅ Definition (Very Easy)

Dataset preparation means:

> Converting raw text + labels into a numerical format that the transformer can learn from.

Transformer models **cannot** learn from:

- raw sentences
- text files
- CSV text directly

They only learn from:

- numbers (token IDs)
- attention masks
- labels (numbers)

🧠 Real-Life Analogy

Think of a student:

- Raw text = English paragraph
- Numbers = exam answer sheet (codes)
  The student (model) only understands **answer sheet format**, not raw paragraphs.

🔹 Quick Notes

- Dataset prep = converting text → numbers + labels
- This step decides training success

2️⃣ Dataset Splits (Train / Validation / Test) — FINAL CLARITY

You already saw this earlier, but now we go deeper.

✅ Train Set

- Used to **update model weights**
- Model learns from this data

✅ Validation Set

- Used to **check performance during training**
- Helps detect overfitting

✅ Test Set

- Used **after training**
- Final honest evaluation

🧠 Analogy

Data Type	Analogy
Train	Practice questions
Validation	Mock tests
Test	Final exam

🔹 Quick Notes

- Train = learning
- Validation = monitoring
- Test = final score

3️⃣ Choosing Dataset for Week 15

We will use **IMDB Sentiment Dataset** (default choice).

Why IMDB?

- Clean labels (0 = negative, 1 = positive)
- Large enough
- Common in NLP
- Perfect for beginners

Step 1: Install & Import Libraries

pip install datasets transformers torch

---

from datasets import load_dataset

Explanation

- datasets → Hugging Face library for datasets
- load_dataset() downloads datasets easily

4️⃣ Load the IMDB Dataset

dataset = load_dataset("imdb")

What this does

- Downloads IMDB dataset
- Automatically splits data

Inspect dataset

dataset

Output (conceptually):

DatasetDict({

```
  train: Dataset(...)
  test: Dataset(...)
})
```

🔹 Important Observation

IMDB has:

- train split
- test split
  No validation split by default.

5️⃣ Creating a Validation Split (VERY IMPORTANT)

We **must** create a validation set.

Why?

Trainer needs:

- train dataset
- validation dataset
  To monitor loss and avoid overfitting.

Step: Split training data into train + validation

```
dataset = dataset["train"].train_test_split(test_size=0.1)
```

Line-by-line explanation

- dataset["train"] → original training data
- .train_test_split() → splits dataset
- test_size=0.1 → 10% becomes validation

Result

Now:

- dataset["train"] → training data
- dataset["test"] → validation data (yes, name is “test” here, but we use it as validation)

Rename for clarity (optional but good)

train_dataset = dataset["train"]
val_dataset = dataset["test"]

🔹 Quick Notes

- Validation split is mandatory
- 90/10 split is common

6️⃣ Understanding Dataset Columns (CRITICAL)

Let’s inspect one sample.

train_dataset[0]

Output example:

{
 'text': "This movie was absolutely fantastic...",
 'label': 1
}

Meaning

- text → input sentence
- label → target class
    - 0 = Negative
    - 1 = Positive

🔹 Quick Notes

- Labels must be numeric
- Trainer expects column name "label"

7️⃣ What is Tokenization for Training?

❗ Important Difference

Tokenization for **training** ≠ tokenization for **inference**

During training:

- All inputs must have **same length**
- Must include labels
- Must be batched

Tokenizer produces:

- input_ids
- attention_mask

These are **required** for training.

8️⃣ Load Tokenizer (Must Match Model!)

Choose model

We will fine-tune **DistilBERT**.

from transformers import AutoTokenizer

---

model_name = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)

Explanation

- Loads tokenizer used during pretraining
- Tokenizer + model must match

🔹 Beginner Warning

❌ Wrong tokenizer = broken training

✅ Always use tokenizer from same model name

9️⃣ Define Tokenization Function (MOST IMPORTANT CODE)

We must define a function that:

- takes raw text
- returns tokenized output

```
def tokenize_function(example):
    return tokenizer(
        example["text"],
        truncation=True,
        padding="max_length",
        max_length=128
    )
```

Line-by-Line Explanation (VERY IMPORTANT)

example["text"]

- Gets raw sentence

---

truncation=True

If sentence is longer than 128 tokens:

- extra tokens are cut

Why?

- Transformers have max length
- Prevents memory issues

---

padding="max_length"

If sentence is shorter than 128 tokens:

- pad with zeros

Why?

- All inputs must be same length for batching

---

max_length=128

- Maximum tokens per sentence
- 128 is standard for sentiment tasks
- Faster and enough context

🔹 Quick Notes

- truncation avoids crashes
- padding ensures uniform shape
- 128 tokens = good balance

🔟 Apply Tokenization to Entire Dataset

This is where Hugging Face shines.

```
train_dataset = train_dataset.map(tokenize_function, batched=True)
val_dataset = val_dataset.map(tokenize_function, batched=True)
```

Explanation

- .map() applies function to every example
- batched=True → faster processing

After mapping, each example now has:

- input_ids
- attention_mask
- label

🔹 Quick Notes

- .map() = mass transformation
- Always tokenize entire dataset

1️⃣1️⃣ Remove Unnecessary Columns (Cleaner Dataset)

Trainer doesn’t need raw text anymore.

```
train_dataset = train_dataset.remove_columns(["text"])
val_dataset = val_dataset.remove_columns(["text"])
```

Why remove?

- Saves memory
- Trainer focuses only on tensors

1️⃣2️⃣ Set Dataset Format for PyTorch

Trainer uses PyTorch tensors.

```
train_dataset.set_format("torch")
val_dataset.set_format("torch")
```

What this does

- Converts lists → PyTorch tensors
- Makes dataset Trainer-ready

🔹 Quick Notes

- Trainer expects torch tensors
- This step is mandatory

1️⃣3️⃣ Final Dataset Check (DO THIS!)

train_dataset[0]

Expected output keys:

input_ids
attention_mask
label

If you see this → ✅ PERFECT

1️⃣4️⃣ Common Beginner Mistakes (VERY IMPORTANT)

❌ Forgetting validation split

❌ Using wrong tokenizer

❌ Forgetting padding

❌ Using very large max_length (slow + overfitting)

❌ Removing label column accidentally

✅ Golden Rule

If dataset format is correct → Trainer will work smoothly.

1️⃣5️⃣ What We Achieved Today

Your dataset is now:

- split correctly
- tokenized correctly
- converted to tensors
- ready for training

✅ DAY 2 SUMMARY (Week 15)

Today you learned:

✔ Dataset preparation for fine-tuning

✔ Train/validation split creation

✔ Tokenization for training

✔ Importance of padding & truncation

✔ Hugging Face .map() usage

✔ Trainer-ready dataset format

🧾 Day 2 Quick Revision Sheet (Exam-Ready)

- Dataset prep converts text → numbers
- Tokenizer produces input_ids and attention_mask
- padding ensures equal length
- truncation avoids overflow
- Validation set monitors overfitting
- Trainer needs PyTorch tensors

DAY 3 — Training a Transformer using Hugging Face Trainer API (VERY DETAILED, BEGINNER-FRIENDLY)

Today is the BIG DAY 🎯
You will **actually fine-tune DistilBERT** on your dataset.
Yesterday you prepared the dataset.
Today you will **train the model end-to-end**.

I will explain **every concept**, **every parameter**, and **every line of code** as if you are seeing this **for the first time**.

0️⃣ What We Are Doing Today (In One Sentence)

👉 We will **fine-tune DistilBERT** on a sentiment dataset using the **Trainer API**, instead of writing a manual PyTorch training loop.

1️⃣ What is the Hugging Face Trainer API?

✅ Definition (Very Easy)

The **Trainer API** is a tool that:

> Automatically handles the entire training process for transformer models.

It does:

- batching
- forward pass
- loss calculation
- backpropagation
- optimizer step
- evaluation
- logging
- saving models

👉 **You do NOT need to write the training loop manually.**

🧠 Real-Life Analogy

Instead of:

- manually cooking (cut vegetables, control fire, stir food)

Trainer is like:

- a **fully automatic cooking machine**
  You give:
- ingredients (dataset)
- recipe (training arguments)
  It cooks (trains) perfectly.

🔹 Quick Notes

- Trainer = automation for training
- Less code, fewer bugs
- Industry-standard approach

2️⃣ What Does Trainer Need? (VERY IMPORTANT)

Trainer needs **5 things**:

1. A **model**
2. A **tokenized training dataset**
3. A **tokenized validation dataset**
4. **Training arguments** (epochs, batch size, etc.)
5. (Optional) **Metrics function**

We already prepared datasets in **Day 2**.

3️⃣ Step 1 — Import Required Libraries

```
import torch
from transformers import (
    AutoModelForSequenceClassification,
    Trainer,
    TrainingArguments
)
```

Line-by-line explanation

- AutoModelForSequenceClassification
  → Transformer model **with classification head**
- Trainer
  → Handles training loop
- TrainingArguments
  → Stores all training settings

4️⃣ Step 2 — Load the Model (Very Important)

We load **DistilBERT** for classification.

```
model_name = "distilbert-base-uncased"

model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=2
)
```

Line-by-line Explanation

from_pretrained(model_name)

- Loads pretrained DistilBERT weights
- These weights already understand language

num_labels=2

- We are doing **binary classification**
- 0 → Negative
- 1 → Positive

👉 This tells the model:

> "You must predict 2 classes."

🔹 Quick Notes

- Always match num_labels with dataset labels
- Wrong number → training error

5️⃣ Step 3 — What is TrainingArguments?

✅ Definition

TrainingArguments is a configuration object that tells Trainer:

> HOW to train the model.

It controls:

- learning rate
- batch size
- epochs
- evaluation frequency
- logging
- saving models

6️⃣ Step 4 — Define TrainingArguments (MOST IMPORTANT BLOCK)

```
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir="./logs",
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss"
)
```

Line-by-Line Explanation (VERY DETAILED)

output_dir="./results"

- Folder where:
    - checkpoints
    - models
    - logs are saved

---

evaluation_strategy="epoch"

- Evaluate model **after every epoch**
- Helps monitor validation loss

---

save_strategy="epoch"

- Save model **after every epoch**

- Useful for recovery

---

learning_rate=2e-5

- Controls how much weights change
- Small value → stable learning
- Standard value for transformers

🧠 Analogy:

- Big learning rate = jumping too fast → fall
- Small learning rate = careful steps → success

---

per_device_train_batch_size=16

- Number of samples processed together (training)
- 16 is safe for most GPUs / Colab

---

per_device_eval_batch_size=16

- Batch size during validation

---

num_train_epochs=3

- Full passes over training data
- Transformers usually need **2–4 epochs**

---

weight_decay=0.01

- Prevents overfitting
- Penalizes very large weights

---

logging_dir="./logs"

- Where training logs are stored

---

load_best_model_at_end=True

- After training:
    - best model (lowest validation loss) is loaded automatically

---

metric_for_best_model="eval_loss"

- Best model is chosen based on **lowest validation loss**

🔹 Quick Notes

- Learning rate + epochs = most sensitive settings
- 3 epochs + 2e-5 is a safe default

7️⃣ Step 5 — Define Evaluation Metrics (Accuracy + F1)

Trainer trains using **loss**, but we want **accuracy & F1**.

Import metrics

from sklearn.metrics import accuracy_score, f1_score

Define metric function

```
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = torch.argmax(torch.tensor(logits), dim=1)

    acc = accuracy_score(labels, predictions)
    f1 = f1_score(labels, predictions)

    return {
        "accuracy": acc,
        "f1": f1
    }
```

Line-by-Line Explanation

eval_pred

- Contains:

- model predictions (logits)
- true labels

---

torch.argmax(logits, dim=1)

- Picks class with highest score
- Converts logits → predicted class

---

accuracy_score(...)

- Measures correctness

---

f1_score(...)

- Balances precision + recall

🔹 Quick Notes

- Metrics are for **evaluation only**
- They do NOT affect training weights

8️⃣ Step 6 — Create the Trainer Object

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics
)
```

Explanation

- model → DistilBERT classifier
- args → training configuration
- train_dataset → tokenized training data
- eval_dataset → validation data
- compute_metrics → accuracy & F1

👉 Trainer now knows **everything** it needs.

9️⃣ Step 7 — Start Training (ONE LINE!)

trainer.train()

🎉 **Training begins here** 🎉

What Happens Internally?

Trainer:

1. Takes a batch
2. Runs forward pass
3. Computes loss
4. Backpropagation
5. Updates weights
6. Evaluates after epoch
7. Saves model

All automatically.

🔍 What You Will See During Training

Typical logs:

Epoch 1/3
Training Loss: 0.42
Validation Loss: 0.35
Accuracy: 0.85
F1: 0.84

How to Read This

- Training loss ↓ → model learning
- Validation loss ↓ → generalizing well
- Accuracy ↑ → predictions improving
- F1 ↑ → balanced performance

🔟 Common Beginner Mistakes (READ CAREFULLY)

❌ Too many epochs → overfitting

❌ Large learning rate → unstable training

❌ No validation dataset → blind training

❌ Forgetting num_labels → model crash

❌ Using wrong dataset format → Trainer error

1️⃣1️⃣ What We Achieved Today

✅ You have:

- Fine-tuned DistilBERT
- Used Trainer API correctly
- Tracked training & validation loss
- Computed accuracy & F1
- Learned professional training workflow

✅ DAY 3 SUMMARY (Week 15)

Today you learned:

✔ What Trainer API is

✔ How to configure training properly

✔ How to fine-tune DistilBERT

✔ How loss & metrics work during training

✔ How to read training logs

🧾 Day 3 Quick Revision Sheet

- Trainer automates training loop
- TrainingArguments control learning behavior
- Loss trains the model

- Accuracy & F1 evaluate performance
- Small learning rate is essential
- Validation loss monitors overfitting

DAY 4 — Evaluating the Fine-Tuned Model (Accuracy, F1, Confusion Matrix, Error Analysis)

Today is about answering ONE big question:
❓ *"Is my fine-tuned model actually good?"*

You trained the model on **Day 3**.

Now you must **evaluate it properly like a real ML engineer**.

Everything below is explained **from zero**, **in very easy language**, with **step-by-step code explanations**.

🎯 Day 4 Goal (Very Clear)

By the end of today, you will be able to:

- Evaluate a **fine-tuned transformer** correctly
- Understand **evaluation vs training**
- Compute **accuracy, precision, recall, F1**
- Build and understand a **confusion matrix**
- Perform **error analysis** (most important for reports)
- Write a **strong evaluation section** in your project

1️⃣ What Does "Model Evaluation" Mean?

✅ Definition (Very Easy)

Model evaluation means:

> Checking how well your trained model performs on data it has **never seen before**.

Why evaluation is necessary

During training:

- model sees training data many times
- performance may look very good

But real-world data is:

- unseen
- messy
- different

So we must test on **new data**.

🧠 Real-Life Analogy

- Practicing answers at home ≠ real exam
- Evaluation = final exam

🔹 Quick Notes

- Training performance ≠ real performance
- Evaluation tells the truth

2️⃣ Evaluation Data: What Should We Use?

❗ Very Important Rule

❌ NEVER evaluate on training data

✅ ALWAYS evaluate on **validation or test data**

In Week 15:

- Training data → model learns
- Validation data → monitor during training
- Test data → final evaluation (if available)

Today we'll evaluate using:

👉 **Validation dataset** (from Day 2)

3️⃣ Ways to Evaluate a Fine-Tuned Model

We will do **3 levels of evaluation**:

1. **Built-in Trainer evaluation**
2. **Manual predictions + metrics**
3. **Error analysis (human-level inspection)**

4️⃣ Method 1: Trainer's Built-in Evaluation (Easiest)

Trainer already knows:

- model
- validation dataset
- metrics function

So evaluation is **one line**.

Step 1: Run Evaluation

```
eval_results = trainer.evaluate()
print(eval_results)
```

What this does

- Runs model on validation dataset
- Computes:
  - validation loss
  - accuracy
  - F1-score

Example Output (Simplified)

```
{
 'eval_loss': 0.32,
 'eval_accuracy': 0.86,
 'eval_f1': 0.85,
 'eval_runtime': 45.3
}
```

How to Understand This

- eval_loss ↓ → good generalization
- accuracy = 0.86 → 86% correct predictions
- f1 = 0.85 → balanced precision & recall

🔹 Quick Notes

- Trainer evaluation is fast
- Good for quick performance check

5️⃣ Why Built-in Evaluation Is Not Enough

Trainer metrics are:

- numbers only

But for **projects, exams, interviews**, you must also:

- understand **where the model fails**
- show **confusion matrix**
- show **real examples of errors**

That's what we do next.

6️⃣ Method 2: Manual Predictions on Validation Data

We will:

- get predictions

- compare with true labels
- compute metrics manually

This gives **full control and understanding**.

Step 1: Get Predictions from Trainer

predictions = trainer.predict(val_dataset)

What is inside

predictions

?

predictions.predictions # logits
predictions.label_ids   # true labels

🔹 Important

- predictions.predictions = logits (not probabilities)
- predictions.label_ids = true labels

Step 2: Convert Logits to Class Labels

import numpy as np

logits = predictions.predictions
y_true = predictions.label_ids

y_pred = np.argmax(logits, axis=1)

Line-by-line Explanation

logits

- raw model outputs
- shape: [num_samples, num_classes]

np.argmax(logits, axis=1)

- picks class with highest score

- converts logits → predicted labels

🔹 Quick Notes

- argmax gives final class
- probabilities are not needed for metrics

7️⃣ Compute Evaluation Metrics Manually

Now we calculate metrics ourselves.

Step 1: Import metrics

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

---

Step 2: Calculate metrics

```
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
```

What Each Metric Means (Again, Very Simple)

- **Accuracy** → overall correctness
- **Precision** → how reliable positive predictions are
- **Recall** → how many positives were correctly found
- **F1-score** → balance of precision and recall

🔹 Quick Notes

- Accuracy alone is not enough

- F1 is very important for NLP tasks

8️⃣ Confusion Matrix (MOST IMPORTANT VISUAL TOOL)

✅ Definition

A **confusion matrix** shows:

> How many predictions were correct and what types of mistakes happened.

Step 1: Create Confusion Matrix

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_true, y_pred)
print(cm)
```

Example Output

```
[[430  70]
 [ 60 440]]
```

How to Read This (VERY IMPORTANT)

Actual ↓ / Predicted →	Negative	Positive
Negative	430 (TN)	70 (FP)
Positive	60 (FN)	440 (TP)

Meaning

- 430 correct negatives

- 440 correct positives
- 70 false positives
- 60 false negatives

🔹 Quick Notes

- FP & FN tell where model is weak
- Confusion matrix = error breakdown

9️⃣ Visualize Confusion Matrix (Optional but Good)

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```

Explanation

- Heatmap shows mistakes clearly
- Dark cells = more predictions

🔟 Method 3: Error Analysis (MOST IMPORTANT FOR REPORTS)

✅ Definition

Error analysis means:

> Looking at wrong predictions and understanding *why* the model failed.

This is what separates:

- average projects ❌
- strong ML projects ✅

Step 1: Find Wrong Predictions

wrong_indices = np.where(y_true != y_pred)[0]

Explanation

- Finds indices where model prediction ≠ true label

Step 2: Inspect Some Wrong Examples

```
for idx in wrong_indices[:5]:
    text = val_dataset[idx]["input_ids"]
    print("Index:", idx)
    print("True label:", y_true[idx])
    print("Predicted label:", y_pred[idx])
    print("-" * 50)
```

⚠️ We removed raw text earlier, so ideally you should keep a copy of raw validation text separately for analysis.

Best Practice (IMPORTANT)

Before removing "text" column, save:

raw_texts = val_dataset_original["text"]

Then during error analysis:

print(raw_texts[idx])

1️⃣1️⃣ Common Reasons NLP Models Make Mistakes

From real-world experience:

❌ Sarcasm

“Yeah, great movie… I almost slept.”

❌ Mixed sentiment

“Good acting but terrible story.”

❌ Negation

“Not good at all.”

❌ Long text

Important sentiment appears at the end.

❌ Domain mismatch

Model trained on movies, used on product reviews.

🔹 Quick Notes

- Errors are expected
- Understanding them is what matters

1️⃣2️⃣ How to Write Evaluation Section in Your Report

Use this structure:

1. Dataset used for evaluation
2. Number of samples
3. Accuracy, Precision, Recall, F1
4. Confusion matrix explanation
5. 3–5 error examples + reasons

Sample Report Sentence

“The fine-tuned DistilBERT model achieved an accuracy of 86% and an F1-score of 0.85 on the validation dataset. The confusion matrix shows balanced performance across both classes, with slightly more errors in mixed-sentiment reviews. Error analysis revealed challenges with sarcasm and long reviews.”

✅ DAY 4 SUMMARY (Week 15)

Today you learned:

✔ How to evaluate a fine-tuned transformer

✔ Trainer-based evaluation

✔ Manual metric computation

✔ Confusion matrix creation and interpretation

✔ Error analysis techniques

✔ How to explain evaluation in reports

🧾 Day 4 Quick Revision Sheet

- Evaluation uses unseen data
- Trainer provides quick metrics
- Manual evaluation gives deeper understanding
- Confusion matrix shows error types
- Error analysis explains model weaknesses

DAY 5 — Overfitting Control + Training Efficiency (Early Stopping, Regularization, Best Practices)

Today is about training "smartly".
Many beginners train a model and think:
"More epochs = better accuracy."
❌ This often makes the model worse due to **overfitting**.

🎯 Day 5 Goal:

By the end of today, you will understand:

- What **overfitting** is (deep clarity)
- How to detect overfitting using **training vs validation loss**
- What **early stopping** means and how to use it

- How to choose **epochs, learning rate, batch size**
- What **regularization** means in NLP (simple)
- Techniques that reduce overfitting in transformer fine-tuning
- Practical “efficiency tips” to train faster without losing quality

---

1️⃣ What is Overfitting? (Deep Beginner Explanation)

✅ Definition (Very Easy)

Overfitting means:

> The model memorizes the training data instead of learning general rules.

So:

- training performance becomes very high ✅
- but new/unseen data performance becomes low ❌

🧠 Real-Life Analogy

A student memorizes last year’s exam answers word-by-word.

If the same questions come → student scores high.

If new questions come → student fails.

That is overfitting.

🔹 Quick Notes

- Overfitting = memorization
- Good train results, bad validation results

2️⃣ How to Detect Overfitting (Most Important Skill)

We detect overfitting by comparing:

✅ **Training loss**

✅ **Validation loss**

What Should Ideally Happen?

- Training loss decreases steadily ✅
- Validation loss decreases steadily ✅

Overfitting Pattern

- Training loss keeps decreasing ✅
- Validation loss stops decreasing or increases ❌

Example Table (Understand This)

Epoch	Train Loss	Val Loss	Meaning
1	0.60	0.52	good learning
2	0.45	0.40	improving
3	0.30	0.42	❌ overfitting starts
4	0.20	0.55	❌ strong overfitting

🔹 Quick Notes

- Val loss is the real truth
- If val loss increases → stop training

3️⃣ Why Transformers Overfit Easily?

Transformers are:

- very powerful
- have millions of parameters
- can memorize small datasets quickly

Also NLP datasets sometimes have:

- repeated patterns
- weak labeling quality
- small size

🔹 Quick Notes

- Transformers learn fast
- Overfitting happens quickly if training too long

4️⃣ What is Early Stopping?

✅ Definition (Very Simple)

Early stopping means:

> Stop training when validation performance stops improving.

Instead of forcing 10 epochs, we stop at best point.

🧠 Analogy

If you keep studying the same notes too much:

- you start memorizing exact lines
  Better to stop and practice new questions.

🔹 **Quick Notes**

- Early stopping saves time
- Prevents overfitting

5️⃣ Early Stopping in Hugging Face Trainer (Step-by-Step)

Trainer supports early stopping using callbacks.

Step 1: Import callback

from transformers import EarlyStoppingCallback

Explanation

- Callback = extra feature added to Trainer
- EarlyStoppingCallback checks validation metric

Step 2: Add evaluation + saving settings (must be enabled)

Early stopping needs evaluation happening regularly.

```
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    num_train_epochs=10,   # we keep big, early stopping will stop earlier
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16
)
```

Line-by-line explanation of new parts

metric_for_best_model="eval_loss"

- early stopping monitors validation loss

greater_is_better=False

- because lower loss is better

num_train_epochs=10

- we set a high value
- early stopping will stop at best epoch automatically

Step 3: Add callback to Trainer

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=2)]
)
```

Explanation line-by-line

callbacks=[...]

Adds callback features to training.

EarlyStoppingCallback(early_stopping_patience=2)

Means:

- If validation loss does not improve for **2 evaluations**
  → stop training

Example:

- epoch 3 no improvement
- epoch 4 no improvement
  → stop

🔹 Quick Notes

- Patience = how many times to wait before stopping
- 2 is common for beginners

6️⃣ Regularization (Easy Explanation)

✅ Definition

Regularization means:

> Techniques that stop the model from memorizing.

Types you will use in transformers

✅ Weight decay

✅ Dropout (already inside models)

✅ Fewer epochs

✅ Smaller model (DistilBERT instead of BERT)

6.1 Weight Decay

✅ Definition

Weight decay:

> Punishes very large weights to reduce memorization.

Add in training args

weight_decay=0.01

🔹 Quick Notes

- Weight decay helps generalization
- Values like 0.01 are common

7️⃣ Choosing Hyperparameters Smartly (Most Beginner Confusion)

✅ Learning Rate (Most Sensitive)

- Too high → training unstable
- Too low → training slow

Best safe values:

- 2e-5
- 3e-5
- 5e-5

Quick analogy

Learning rate is like:

- speed of writing notes
  Too fast → messy, wrong
  Too slow → wastes time

✅ Epochs

Most fine-tuning works in:

- 2 to 4 epochs

If you train 10 epochs without early stopping:

- high chance of overfitting

✅ Batch Size

- 8, 16, 32

Bigger batch:

- faster training
- but needs more memory

🔹 **Quick Notes**

- Learning rate is most important knob
- Use 2–4 epochs usually
- Use early stopping if unsure

8️⃣ Training Efficiency Tips (Save Time + GPU)

✅ Tip 1: Use

fp16=True

if GPU supports

Definition

fp16 = mixed precision training

- faster
- less memory

```
training_args = TrainingArguments(
    ...,
    fp16=True
)
```

✅ Tip 2: Use smaller max_length

128 is good for IMDB.

256 doubles compute time often.

✅ Tip 3: Use DistilBERT not BERT for first fine-tune

- faster
- easier
- still strong performance

✅ Tip 4: Evaluate less frequently if training is slow

If dataset is huge:

- evaluate every epoch (good)
- or every few steps (advanced)

🔹 Quick Notes

- fp16 saves memory and speeds training
- max_length controls speed directly

9️⃣ Track Training Loss and Validation Loss Properly

Trainer logs loss automatically.

You can also get log history:

```
logs = trainer.state.log_history
logs[:5]
```

This stores:

- step
- training loss
- evaluation loss
- metrics

🔹 Quick Notes

- Log history is useful for plotting later

🔟 Common Beginner Mistakes (Important)

❌ Training too long

✅ Use early stopping or fewer epochs

❌ Using high learning rate like 1e-3

✅ Use 2e-5 to 5e-5

❌ Increasing max_length unnecessarily

✅ Keep max_length 128 unless needed

❌ Evaluating on training set

✅ Use validation/test set only

✅ DAY 5 SUMMARY (Week 15)

Today you learned:

✔ Overfitting meaning and detection

✔ Training loss vs validation loss patterns

✔ Early stopping concept and code

✔ Regularization (weight decay, dropout, fewer epochs)

✔ Hyperparameter selection best practices

✔ Training efficiency tips (fp16, max_length, smaller models)

🧾 Day 5 Quick Revision Sheet (Exam-ready)

- Overfitting: good train, bad validation
- Detect using loss curves

- Early stopping stops training when val loss doesn't improve
- Use small learning rate (2e-5)
- Use 2–4 epochs normally
- Weight decay helps reduce memorization
- fp16 speeds training on GPU

DAY 6 — Saving, Reloading & Using Your Fine-Tuned Model (Deployment-Ready Basics)

> **Today is the moment your model becomes "real".**
> Until now, your model lived only inside a training notebook.
> Today you will **save it, reload it, and use it like a real application model**.

Everything is explained **slowly**, **clearly**, and **from zero**, exactly for a **first-time beginner**.

🎯 Day 6 Goal (Very Clear)

By the end of today, you will be able to:

✔ Save a fine-tuned transformer model

✔ Save its tokenizer correctly

✔ Reload the model later (even after restart)

✔ Run inference on **new custom text**

✔ Use the model with **pipeline**

✔ Understand what files get saved and why

✔ Prepare your model for real apps (chatbot / API / UI)

1️⃣ Why Saving the Model is Necessary

❓ Beginner Question

"Why can't I just keep the trained model in memory?"

✅ Answer

Because:

- Notebook can close
- Kernel can restart
- Training is expensive
- You don't want to retrain again

So we **save** the trained knowledge.

🧠 Real-Life Analogy

You study for an exam → you don't throw notes away.

You **save notes** so you can revise anytime.

🔹 Quick Notes

- Training = expensive
- Saving = reuse anytime
- Required for deployment

2️⃣ What Exactly Needs to be Saved?

To use a fine-tuned transformer later, you must save:

1. **Model weights**
2. **Model configuration**
3. **Tokenizer**

👉 If tokenizer is missing → model input breaks ❌

🔹 Golden Rule

Always save model + tokenizer together

3️⃣ How Hugging Face Saves Models (Concept)

Hugging Face uses a standard format:

When you save, it creates:

- pytorch_model.bin → learned weights
- config.json → model architecture & labels
- tokenizer.json / vocab files → tokenization rules

🔹 **Quick Notes**

- These files are portable
- Can be loaded anywhere (Colab, server, laptop)

4️⃣ Saving the Fine-Tuned Model (Step-by-Step)

We assume:

- You already trained using Trainer
- Model is now fine-tuned

Step 1: Define save directory

save_directory = "./fine_tuned_model"

Explanation

- Folder where model files will be stored
- Can be any name you want

Step 2: Save model using Trainer

trainer.save_model(save_directory)

What this does

- Saves model weights
- Saves config
- Uses Hugging Face format

🔹 Quick Notes

- This saves the **best model** (if load_best_model_at_end=True)
- Very important for real use

Step 3: Save the tokenizer

tokenizer.save_pretrained(save_directory)

Why this is mandatory

- Model expects inputs tokenized the same way
- Different tokenizer = wrong token IDs = bad results

🔹 Quick Notes

- Never skip tokenizer saving
- Model + tokenizer are a pair

5️⃣ Check Saved Files (Understand This)

After saving, your folder will look like:

```
fine_tuned_model/
│
├── pytorch_model.bin
├── config.json
├── tokenizer.json
├── tokenizer_config.json
```

├─ vocab.txt

Meaning of Each File (Easy)

File	Purpose
pytorch_model.bin	Learned weights
config.json	Model settings + labels
tokenizer.json	Tokenization rules
vocab.txt	Vocabulary
tokenizer_config.json	Tokenizer metadata

🔹 **Quick Notes**

- These files are enough to reload model
- Can be uploaded to Hugging Face Hub later

6️⃣ Reloading the Fine-Tuned Model (Very Important)

Now imagine:

- You close notebook
- Restart kernel
- Come back next day

You must reload the model.

Step 1: Import required classes

from transformers import AutoModelForSequenceClassification, AutoTokenizer

---

Step 2: Load tokenizer from saved folder

loaded_tokenizer = AutoTokenizer.from_pretrained(save_directory)

Explanation

- Reads tokenizer files
- Restores exact tokenization rules

Step 3: Load model from saved folder

loaded_model = AutoModelForSequenceClassification.from_pretrained(save_directory)

Explanation

- Loads fine-tuned weights
- Loads classification head
- Loads label mappings

🔹 Quick Notes

- No retraining happens here
- This is instant loading

7️⃣ Running Inference Manually (Step-by-Step)

Now we test the reloaded model on **new text**.

Step 1: Put model in evaluation mode

loaded_model.eval()

Why?

- Disables dropout
- Makes predictions stable

Step 2: Prepare input text

text = "This movie was absolutely fantastic!"

Step 3: Tokenize input

```
inputs = loaded_tokenizer(
    text,
    return_tensors="pt",
    truncation=True,
    padding=True
)
```

Line-by-line explanation

- Converts text → input IDs
- return_tensors="pt" → PyTorch format
- truncation → avoid long overflow
- padding → proper shape

Step 4: Run model forward pass

```
import torch

with torch.no_grad():
    outputs = loaded_model(**inputs)
```

Explanation

- torch.no_grad() → inference mode (no training)
- outputs.logits → raw scores

Step 5: Convert logits to prediction

```
logits = outputs.logits
predicted_class = torch.argmax(logits, dim=1).item()
```

Explanation

- Picks class with highest score
- Converts tensor → Python number

Step 6: Convert class ID to label

```
label = loaded_model.config.id2label[predicted_class]
print("Prediction:", label)
```

🔹 **Quick Notes**

- id2label comes from config.json
- Ensures human-readable output

8️⃣ Running Inference using Pipeline (Easiest Way)

Instead of manual steps, Hugging Face provides **pipeline**.

Step 1: Create pipeline with your model

```
from transformers import pipeline

sentiment_pipeline = pipeline(
    "sentiment-analysis",
    model=loaded_model,
    tokenizer=loaded_tokenizer
)
```

Step 2: Test on new sentences

```
sentiment_pipeline("I really loved the acting and story.")
```

Example Output

```
[{'label': 'POSITIVE', 'score': 0.97}]
```

🔹 **Quick Notes**

- Pipeline is best for apps
- Cleaner and safer

9️⃣ Test on Multiple Custom Inputs

```
texts = [
    "This was the worst movie ever.",
    "I enjoyed the visuals but the story was boring.",
    "Absolutely amazing experience!"
]

for t in texts:
    print(t, "→", sentiment_pipeline(t))
```

Why this matters

- Confirms model generalization
- Tests real-world sentences

🔟 Add Confidence Threshold (Professional Touch)

We add logic:

- If confidence < threshold → uncertain

```
def predict_with_confidence(text, pipe, threshold=0.80):
    result = pipe(text)[0]
    label = result["label"]
    score = result["score"]

    if score < threshold:
        return "UNCERTAIN", score
    return label, score
```

Test it

```
predict_with_confidence(
    "The movie was okay, not great, not terrible.",
    sentiment_pipeline
)
```

🔹 Quick Notes

- Very useful for apps
- Improves trust

1️⃣1️⃣ How This Becomes a Real Application

Now your model can be used in:

- Chatbot
- Web app (Streamlit / Flask)
- API
- Mobile backend

Because:

- Model is saved
- Can be reloaded anytime
- Works on new text

1️⃣2️⃣ Common Beginner Mistakes (Very Important)

❌ Forgetting to save tokenizer

❌ Loading wrong tokenizer

❌ Running inference without .eval()

❌ Not using torch.no_grad()

❌ Assuming pipeline auto-uses fine-tuned model

✅ Best Practice

Always load:

- model from saved directory
- tokenizer from same directory

✅ DAY 6 SUMMARY (Week 15)

Today you learned:

✔ Why saving model is necessary

✔ What files Hugging Face saves

✔ How to save fine-tuned model

✔ How to reload model & tokenizer

✔ Manual inference step-by-step

✔ Pipeline inference with fine-tuned model

✔ Confidence threshold for real apps

🧾 Day 6 Quick Revision Sheet (Exam-Ready)

- Fine-tuned models must be saved
- Save model + tokenizer together
- Reload using from_pretrained()
- Use .eval() and torch.no_grad() for inference
- Pipeline simplifies deployment
- Confidence threshold improves reliability

✅ DAY 7 —

FULL MINI PROJECT (End-to-End): Fine-Tune DistilBERT → Evaluate → Save → Reload → Use like an App

You asked: **“Explain full project step by step”**

So today I’ll give you a **complete beginner-friendly project guide** that you can follow from **start to finish**.

We will build a **Sentiment Analysis Model** (Positive/Negative) by fine-tuning **DistilBERT** on **IMDB**.

✅ You can later convert the same structure to **Fake News Detector** easily.

🎯 Project Goal (1 line)

> Train a transformer model (DistilBERT) on movie reviews so it can predict **Positive** or **Negative** sentiment for new text.

✅ What You Will Create (Outputs)

By the end, you will have:

1. A fine-tuned model folder (fine_tuned_model/)
2. Evaluation results (Accuracy, F1, Confusion Matrix)
3. A prediction function you can use in an app
4. A mini report summary for submission

📌 Project Steps Overview (Big Picture)

Full Pipeline:

1. Setup + install libraries
2. Load dataset
3. Create train/validation split
4. Load tokenizer
5. Tokenize dataset
6. Load model
7. Define metrics
8. Train with Trainer
9. Evaluate
10. Error analysis
11. Save model + tokenizer
12. Reload model
13. Make predictions like an app

✅ STEP 0 — Setup (Install Libraries)

Code

pip install transformers datasets torch scikit-learn accelerate

Why each library?

- transformers → model + trainer
- datasets → IMDB dataset
- torch → training backend
- scikit-learn → metrics (accuracy, F1, confusion matrix)
- accelerate → helps Trainer run properly

✅ STEP 1 — Import Everything

```
import numpy as np
import torch

from datasets import load_dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    Trainer,
    TrainingArguments,
    pipeline
```

)

from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, classification_report

Beginner explanation

- numpy helps with arrays (argmax)
- torch runs model computations
- load_dataset downloads IMDB
- tokenizer converts text → tokens
- model gives predictions
- trainer trains automatically
- sklearn gives evaluation metrics

✅ STEP 2 — Load IMDB Dataset

raw = load_dataset("imdb")
raw

What you get

- raw["train"] → training reviews
- raw["test"] → test reviews

IMDB labels:

- 0 = Negative
- 1 = Positive

✅ STEP 3 — Create Validation Split (IMPORTANT)

IMDB does not provide validation by default. So we create it:

split = raw["train"].train_test_split(test_size=0.1, seed=42)

train_ds = split["train"]
val_ds = split["test"] # We will use this as validation
test_ds = raw["test"] # Final test set (optional, for final report)

Explanation

- test_size=0.1 → 10% becomes validation
- seed=42 → same split every time (reproducible)

✅ Now you have:

- Train
- Validation
- Test

✅ STEP 4 — Load Tokenizer (Must match model)

```
model_name = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Explanation

- Loads tokenizer rules used during pretraining
- Using the same tokenizer is mandatory

✅ STEP 5 — Tokenize Dataset (MOST IMPORTANT DATA STEP)

Tokenization function:

```
def tokenize_batch(batch):
    return tokenizer(
        batch["text"],
        truncation=True,
        padding="max_length",
        max_length=128
    )
```

Line-by-line meaning

- batch["text"] → list of review texts
- truncation=True → cuts long reviews
- padding="max_length" → makes all equal length

- max_length=128 → faster training, enough for sentiment

Apply tokenization:

```
train_ds = train_ds.map(tokenize_batch, batched=True)
val_ds = val_ds.map(tokenize_batch, batched=True)
test_ds = test_ds.map(tokenize_batch, batched=True)
```

Explanation

- .map() applies tokenization to every example
- batched=True makes it fast

Remove raw text column (optional but recommended)

```
train_ds = train_ds.remove_columns(["text"])
val_ds = val_ds.remove_columns(["text"])
test_ds = test_ds.remove_columns(["text"])
```

Convert to PyTorch tensors

```
train_ds.set_format("torch")
val_ds.set_format("torch")
test_ds.set_format("torch")
```

✅ Now datasets are training-ready.

✅ STEP 6 — Load Model for Classification

```
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    num_labels=2
)
```

Explanation

- Loads DistilBERT weights
- Adds a classification head
- num_labels=2 because we have Positive/Negative

✅ STEP 7 — Define Metrics (Accuracy + F1)

Trainer uses loss internally, but we also want metrics:

```
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    preds = np.argmax(logits, axis=1)

    acc = accuracy_score(labels, preds)
    f1 = f1_score(labels, preds)

    return {"accuracy": acc, "f1": f1}
```

Explanation

- logits = raw scores
- argmax selects predicted class (0/1)
- accuracy & F1 computed with sklearn

✅ STEP 8 — TrainingArguments (Training Settings)

```
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
    logging_dir="./logs"
)
```

Explain each important one:

- learning_rate=2e-5 → safe transformer fine-tuning LR
- batch_size=16 → common value
- epochs=3 → usually enough
- weight_decay → reduces overfitting
- eval_strategy="epoch" → evaluate each epoch
- load_best_model_at_end=True → best validation model kept

✅ STEP 9 — Create Trainer + Train

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    compute_metrics=compute_metrics
)
```

Train:

```
trainer.train()
```

✅ Training will start and show:

- training loss
- validation loss
- accuracy
- F1

✅ STEP 10 — Evaluate Properly

(A) Quick evaluation:

```
trainer.evaluate()
```

(B) Predictions for confusion matrix:

```
pred_out = trainer.predict(val_ds)

logits = pred_out.predictions
y_true = pred_out.label_ids
y_pred = np.argmax(logits, axis=1)
```

(C) Classification Report:

print(classification_report(y_true, y_pred, target_names=["NEG", "POS"]))

(D) Confusion Matrix:

cm = confusion_matrix(y_true, y_pred)
print(cm)

✅ Now you can explain:

- TP, TN, FP, FN
- what the model gets wrong

✅ STEP 11 — Save Fine-Tuned Model + Tokenizer

save_dir = "./fine_tuned_model"
trainer.save_model(save_dir)
tokenizer.save_pretrained(save_dir)

✅ This creates a folder with:

- weights
- config
- tokenizer files

✅ STEP 12 — Reload Model (After Restart)

loaded_tokenizer = AutoTokenizer.from_pretrained(save_dir)
loaded_model = AutoModelForSequenceClassification.from_pretrained(save_dir)

✅ STEP 13 — Use Model Like an App (Pipeline)

sentiment_app = pipeline(

```
    "sentiment-analysis",
    model=loaded_model,
    tokenizer=loaded_tokenizer
)
```

Test:

```
sentiment_app("I loved the movie. It was amazing!")
sentiment_app("Worst movie ever. I wasted my time.")
```

✅ STEP 14 — Add “Uncertain” Output (Professional Feature)

```
def predict_sentiment(text, pipe, threshold=0.80):
    out = pipe(text)[0]
    label = out["label"]
    score = out["score"]

    if score < threshold:
        return {"label": "UNCERTAIN", "confidence": score}

    return {"label": label, "confidence": score}
```

✅ STEP 15 — Mini Report (What to write in submission)

Include these 7 points:

1. Model used: DistilBERT
2. Dataset used: IMDB
3. Train/Validation split: 90/10
4. Training settings: lr, batch size, epochs
5. Final results: accuracy, F1
6. Confusion matrix explanation
7. Error analysis: 3–5 wrong examples and why

✅ DAY 7 FINAL SUMMARY

You now know the full end-to-end project:

✔ Data loading

✔ Splitting

✔ Tokenization

✔ Trainer training

✔ Metrics + evaluation

✔ Confusion matrix

✔ Saving + loading

✔ Running model like an application

🧠 WEEK 15 — PRACTICE QUESTIONS (DAY-WISE)

DAY 1 — Transfer Learning & Fine-Tuning Fundamentals

🔹 A. Very Short Questions (1–2 lines)

1. What is transfer learning in NLP?
2. What is a pretrained transformer model?
3. What does fine-tuning mean?
4. What is a downstream task?
5. Why are transformers suitable for transfer learning?
6. What is feature extraction?

7. Which is better for accuracy: feature extraction or fine-tuning?

🔹 B. Short Answer Questions (3–5 lines)

8. Explain transfer learning with a real-life example.
9. Why is fine-tuning preferred over training from scratch?
10. What happens to model weights during fine-tuning?
11. Explain fine-tuning vs feature extraction.
12. What is the role of the classification head in transformers?

🔹 C. Conceptual / Thinking Questions

13. Why does fine-tuning require much less data than training from scratch?
14. Why can fine-tuning cause overfitting if not handled carefully?

📅

DAY 2 — Dataset Preparation & Tokenization

🔹 A. Very Short Questions

1. Why can transformers not train on raw text?
2. What does a tokenizer do?
3. What are input_ids?
4. What is an attention_mask?
5. Why must all sequences have the same length?
6. What is padding?
7. What is truncation?

🔹 B. Short Answer Questions

8. Explain the purpose of train, validation, and test splits.
9. Why is a validation set required during fine-tuning?
10. Why must tokenizer and model names match?
11. What happens if max_length is too large?
12. Why does the Trainer require labels as numbers?

🔹 C. Applied Questions

13. Why is padding="max_length" used during training?
14. What problems occur if tokenization is done incorrectly?
15. Why is 128 tokens often enough for sentiment analysis?

📅

DAY 3 — Trainer API & Model Training

🔹 A. Very Short Questions

1. What is the Hugging Face Trainer API?
2. Why is Trainer preferred over manual training loops?
3. What is TrainingArguments?
4. What does num_labels represent?
5. What is a training epoch?

🔹 B. Short Answer Questions

6. Explain the role of learning_rate in training.
7. Why is a small learning rate used for transformers?
8. What does load_best_model_at_end=True do?
9. Why do we track validation loss during training?
10. What happens internally when trainer.train() is called?

🔹 C. Applied / Practical Questions

11. What would happen if num_labels is set incorrectly?
12. Why does increasing epochs too much reduce performance?
13. How does Trainer automate backpropagation?

📅

DAY 4 — Evaluation, Metrics & Error Analysis

🔹 A. Very Short Questions

1. What is model evaluation?
2. Why should evaluation never be done on training data?
3. What is accuracy?
4. What is F1-score?
5. What is a confusion matrix?
6. What are false positives?

🔹 B. Short Answer Questions

7. Explain precision and recall in simple words.
8. Why is F1-score preferred over accuracy sometimes?
9. What information does a confusion matrix provide?
10. What is error analysis?
11. Why is validation loss important after fine-tuning?

🔹 C. Applied / Thinking Questions

12. Why do sentiment models fail on sarcastic sentences?
13. How does error analysis improve your project report?
14. What kind of errors are common in NLP sentiment models?

📅

DAY 5 — Overfitting Control & Training Efficiency

🔹 A. Very Short Questions

1. What is overfitting?
2. What is early stopping?
3. What does early_stopping_patience mean?
4. What is weight decay?
5. What does fp16=True do?

🔹 B. Short Answer Questions

6. How can overfitting be detected using loss curves?

7. Why are transformers more prone to overfitting?
8. How does early stopping prevent overfitting?
9. Why should learning rate be small for fine-tuning?
10. What are signs of a well-trained model?

🔹 C. Applied Questions

11. Why does validation loss increase even when training loss decreases?
12. What happens if early stopping is not used?
13. How does reducing max_length improve training efficiency?

📅

DAY 6 — Saving, Reloading & Inference

🔹 A. Very Short Questions

1. Why must a fine-tuned model be saved?
2. Which two components must always be saved together?
3. What file stores the trained weights?
4. Why is model.eval() used during inference?
5. What does torch.no_grad() do?

🔹 B. Short Answer Questions

6. Explain how a saved model is reloaded.
7. Why will inference fail if tokenizer is not saved?
8. Difference between manual inference and pipeline inference.
9. What is confidence score in prediction?
10. Why is pipeline useful for deployment?

🔹 C. Applied Questions

11. Why should confidence threshold be used in applications?
12. What problems occur if a wrong tokenizer is loaded?
13. How does saving the model help real-world deployment?
