Data Science & ML Projects (Intermediate)

A Complete 8-Chapter Data Science & ML Course

Projects built:

A real sentiment analyzer, a full churn-prediction model comparison
A movie recommender, a first real CNN digit recognizer, a weather forecaster
A named entity extractor, an LLM-powered labeling pipeline
and a capstone mini app combining the entire track

Format: A4 · Dark-theme code examples, one complete project per chapter
Course 6 of 6 in the Data Science & ML subject · Part 2 of the Projects track

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 8

Sentiment Analyzer: Classifying Real Text at Scale

Data Science & ML Projects (Intermediate)

Chapter 1 · Sentiment Analyzer: Classifying Real Text at Scale

> *A social-media/email tone-and-sentiment analyzer was the very first project idea raised when this entire Data Science & ML subject was first agreed — long before `nlp1` existed to make it possible. This chapter delivers on that promise directly, by taking `nlp1-10`'s own exact capstone pipeline — unchanged — and pointing it at real data for the first time: 25,000 real IMDb movie reviews, not a handful of illustrative sentences.*

What We're Building

The identical architecture `nlp1-10` already built and proved works: preprocessing (`nlp1-1`) → frozen pretrained GloVe embeddings (`nlp1-9`) → an LSTM sequence classifier (`nlp1-6`). Nothing about the model changes here — what changes is scale: a real, freely available, canonical sentiment dataset instead of a toy example, and everything that comes with training on tens of thousands of real examples instead of a handful.

Step 1: Loading the Real Dataset

```
import pandas as pd

df = pd.read_csv("imdb_reviews.csv")   # columns: review, sentiment ("positive"/"negative")
print(df.shape)
print(df["sentiment"].value_counts())
```

The IMDb movie review dataset is the canonical benchmark for exactly this task — 25,000 real reviews, roughly balanced between positive and negative, freely available and genuinely unremarkable to work with (no scraping, no missing values, no cleaning pipeline needed here — that work belongs to Chapter 2's own dataset, not this one).

Step 2: Preprocessing — nlp1-1's Own Pipeline, Reused Directly

```
import re

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

df["tokens"] = df["review"].apply(preprocess)
```

`nlp1-1`'s own tokenization/cleaning approach, applied unchanged — nothing here needed to be redesigned for real data, since preprocessing was always meant to generalize beyond whatever toy examples originally demonstrated it.

Step 3: Building a Real Vocabulary

```
from collections import Counter

all_tokens = [tok for tokens in df["tokens"] for tok in tokens]
vocab = {word: i + 2 for i, (word, count) in enumerate(Counter(all_tokens).most_common(20000))}
vocab["<pad>"] = 0
vocab["<unk>"] = 1
print(f"Vocabulary size: {len(vocab)}")
```

A REAL-SCALE DETAIL NLP1'S OWN TOY EXAMPLES NEVER NEEDED

A real dataset's own raw vocabulary can easily exceed 100,000 distinct words. Capping it at the 20,000 most frequent (`most_common(20000)`) keeps the embedding table a manageable size — a genuinely practical necessity at this scale that a five-sentence toy example never has to face.

Step 4: Loading Only the GloVe Vectors This Vocabulary Actually Needs

```
import numpy as np

embedding_dim = 100
embedding_matrix = np.zeros((len(vocab), embedding_dim))

with open("glove.6B.100d.txt", encoding="utf-8") as f:
    for line in f:
        parts = line.split()
        word = parts[0]
```

```
        if word in vocab:
            embedding_matrix[vocab[word]] = np.array(parts[1:], dtype="float32")
```

The full GloVe file (`nlp1-9`'s own pretrained embeddings) contains vectors for roughly 400,000 words — loading every single one into memory just to use 20,000 of them would be genuine, avoidable waste. This loop reads the file once and keeps only the rows this specific vocabulary needs.

Step 5: The Exact Model From nlp1-6/nlp1-10 — Unchanged

```
import torch
import torch.nn as nn

class SentimentClassifier(nn.Module):
    def __init__(self, embedding_matrix, hidden_dim=128):
        super().__init__()
        self.embedding = nn.Embedding.from_pretrained(
            torch.tensor(embedding_matrix, dtype=torch.float32), freeze=True
        )
        self.lstm = nn.LSTM(embedding_matrix.shape[1], hidden_dim, batch_first=True)
        self.output = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        embedded = self.embedding(x)
        _, (hidden, _) = self.lstm(embedded)
        return torch.sigmoid(self.output(hidden[-1]))
```

NOT A REWRITE — THE SAME ARCHITECTURE, AT REAL SCALE

This is `nlp1-10`'s own capstone model, character for character. The frozen embedding choice, the LSTM reading only earlier context, the single sigmoid output — none of it needed to change to handle real data. What genuinely changes in the next two steps is everything *around* the model: how much data flows through it, and how long training actually takes.

Step 6: Training — Now With Real Batching

```
from torch.utils.data import Dataset, DataLoader

def encode(tokens, max_len=200):
    ids = [vocab.get(t, 1) for t in tokens[:max_len]]
    return ids + [0] * (max_len - len(ids))

class ReviewDataset(Dataset):
    def __init__(self, df):
        self.X = [encode(t) for t in df["tokens"]]
```

```
        self.y = (df["sentiment"] == "positive").astype("float32").values

    def __len__(self):
        return len(self.X)

    def __getitem__(self, i):
        return torch.tensor(self.X[i]), torch.tensor(self.y[i])

loader = DataLoader(ReviewDataset(df), batch_size=64, shuffle=True)

model = SentimentClassifier(embedding_matrix)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.BCELoss()

for epoch in range(3):
    for X_batch, y_batch in loader:
        optimizer.zero_grad()
        preds = model(X_batch).squeeze()
        loss = loss_fn(preds, y_batch)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} done.")
```

A GENUINE SCALE GOTCHA NLP1'S OWN TOY EXAMPLES NEVER HIT

25,000 reviews can't all pass through the model at once — `DataLoader` splits them into batches of 64, and fixed-length encoding (`max_len=200`, padding shorter reviews and truncating longer ones) is what makes batching possible at all, since a batch needs every sequence in it to be the same length. Training also genuinely takes real time here — minutes on a modern CPU, much less on a GPU — unlike `nlp1`'s own instant toy examples. This is the honest cost of real scale, not a sign anything is wrong.

Step 7: Evaluating on Real Reviews

```
sample_reviews = [
    "This movie was absolutely fantastic, best film I've seen all year.",
    "Waste of two hours. Terrible acting, worse plot.",
]
for review in sample_reviews:
    encoded = torch.tensor([encode(preprocess(review))])
    score = model(encoded).item()
    print(f"{score:.2f} — {review[:50]}...")
```

A genuinely satisfying moment this project earns that a toy example can't: real, freshly-written sentences, never seen during training, scored by a model trained on real human-written reviews rather than a handful of

illustrative examples.

The same nlp1-10 architecture

Frozen GloVe + LSTM + sigmoid — nothing about the model changed for real data.

Vocabulary capping

20,000 most-frequent words, not the full raw vocabulary — a real necessity at scale.

Loading only needed embeddings

One pass through the GloVe file, keeping only rows this vocabulary needs.

Batching & padding

Fixed-length encoding makes real-scale training via DataLoader possible.

Extend This Project

Try these on your own:

- Swap the frozen embeddings for fine-tuned ones (`freeze=False`, per `nlp1-9`'s own guidance) now that there's enough real data to justify it — compare accuracy.
- Save the trained model and vocabulary to disk so this project doesn't need retraining every time it's reused elsewhere in this course.
- Try the same pipeline on a genuinely different kind of text — real tweets or product reviews — and see whether accuracy holds up on a different writing style.
- Extend Step 7 into a small function that accepts any string and returns a plain "positive"/"negative" label instead of a raw score.

What's Next

Chapter 2: **Customer Churn Predictor** — the full ml1 depth Chapter 7 of the Beginner course deliberately declined to use: real feature engineering, a genuine model comparison, cross-validation, and the full precision/recall/F1 picture.

CHAPTER 1 QUICK REFERENCE

- Delivers directly on this subject's own original roadmap promise — a real sentiment analyzer, first raised before nlp1 even existed
- Reuses nlp1-10's own exact capstone architecture unchanged — frozen GloVe embeddings + LSTM + sigmoid
- What genuinely changes at real scale: vocabulary capping, loading only needed embeddings, and batching/padding via DataLoader
- Training now takes real time (minutes, not instant) — an honest cost of real scale, not a sign of a problem

- Evaluated on genuinely new, never-seen sentences — not just held-out examples from the same training set

CHAPTER 2 OF 8

Customer Churn Predictor: A Full ml1 Model Comparison

Data Science & ML Projects (Intermediate)

Chapter 2 · Customer Churn Predictor: A Full ml1 Model Comparison

> *`dsproj1-7`'s own warn-box was explicit: plain accuracy is the simplified metric, and `ml1-6` covers precision, recall, F1, and the confusion matrix in full. That chapter also deliberately used KNN specifically to avoid preempting `ml1-5`'s logistic regression or `ml1-7`'s decision trees. This chapter is the direct payoff of both deferrals.*

What We're Building

A churn predictor for a telecom-style customer dataset — real subscriber data (tenure, monthly charges, contract type, services subscribed) used to predict whether a customer will cancel. Two real models, compared honestly: logistic regression (`ml1-5`) and a random forest (`ml1-7`), evaluated with the full metric picture `ml1-6` covers, validated with real cross-validation (`ml1-8`).

Step 1: Real Feature Engineering

```
import pandas as pd

df = pd.read_csv("telco_churn.csv")

df["avg_monthly_spend"] = df["TotalCharges"] / df["tenure"].replace(0, 1)
df["is_long_term"] = (df["tenure"] > 24).astype(int)

categorical_cols = ["Contract", "InternetService", "TechSupport"]
df = pd.get_dummies(df, columns=categorical_cols, drop_first=True)

y = (df["Churn"] == "Yes").astype(int)
X = df.drop(columns=["customerID", "Churn", "TotalCharges"])
```

Genuinely new territory beyond `dsproj1-7`'s own four ready-made numeric measurements: `avg_monthly_spend` and `is_long_term` are derived features, engineered because raw `tenure` and `TotalCharges` alone don't directly capture them. `pd.get_dummies()` one-hot encodes categorical columns

like `Contract` into numeric 0/1 columns a model can actually use — real tabular data is rarely all-numeric the way Iris was.

Step 2: Split, and Why One Split Isn't Enough

```
from sklearn.model_selection import train_test_split

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

`stratify=y` keeps the churned/not-churned ratio consistent between train and test sets — worth doing explicitly here, since churn datasets are rarely 50/50, exactly the situation Step 5 comes back to directly.

Step 3: Model 1 — Logistic Regression

```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(max_iter=1000)
logreg.fit(X_train, y_train)
```

`ml1-5`'s own model, applied here for real — a coefficient per feature, each one directly readable as "this feature pushes the prediction toward churn or away from it."

Step 4: Model 2 — Random Forest

```
from sklearn.ensemble import RandomForestClassifier

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

`ml1-7`'s own ensemble model, fit on the identical training data — an intentional apples-to-apples comparison against logistic regression.

Step 5: The Full Evaluation Picture

```
from sklearn.metrics import classification_report, confusion_matrix

for name, model in [("Logistic Regression", logreg), ("Random Forest", forest)]:
    preds = model.predict(X_test)
    print(f"\n--- {name} ---")
    print(classification_report(y_test, preds, target_names=["Stayed", "Churned"]))
    print(confusion_matrix(y_test, preds))
```

THE ACCURACY PARADOX, CAUGHT IN REAL DATA — NOT JUST DESCRIBED IN THE ABSTRACT

Churn datasets are typically imbalanced — most customers *don't* churn in any given period. A model that predicted "never churns" for every single customer could score a deceptively high accuracy while catching zero actual churners — exactly the scenario `ml1-6` warned about. `recall` on the "Churned" class specifically — how many actual churners the model actually caught — matters far more here than overall accuracy, since missing a real churner is the exact business cost this whole project exists to reduce.

Step 6: Cross-Validation — A More Honest Estimate

```
from sklearn.model_selection import cross_val_score

for name, model in [("Logistic Regression", logreg), ("Random Forest", forest)]:
    scores = cross_val_score(model, X, y, cv=5, scoring="f1")
    print(f"{name}: F1 = {scores.mean():.3f} (+/- {scores.std():.3f})")
```

Per `ml1-8`, a single train/test split can be a lucky or unlucky draw — `cv=5` fits and evaluates each model five separate times on five different splits, reporting the average and spread rather than trusting one number. The `+/-` spread itself is worth reading: a model whose F1 barely moves across folds is more trustworthy than one that swings wildly from one split to the next.

Step 7: Which Features Actually Drive Churn

```
importances = pd.Series(forest.feature_importances_, index=X.columns).sort_values(ascending=False)
print(importances.head(10))
```

A genuine business payoff beyond "the model works": `feature_importances_` reveals which factors the random forest actually leaned on most — real, actionable information for whoever would act on these predictions, not just a black-box score.

AN HONEST COMPARISON, NOT A DECLARED WINNER

A random forest often edges out logistic regression on tabular data with real feature interactions — but logistic regression's own coefficients are directly interpretable in a way a forest's aggregated feature importances aren't quite as cleanly. Neither model is simply "better" in the abstract; the right choice depends on whether interpretability or raw predictive performance matters more for a given real use case.

pd.get_dummies()

One-hot encodes categorical columns a model can't use as raw text.

stratify=y

Keeps class balance consistent across train/test splits on imbalanced data.

The accuracy paradox, concretely

High accuracy alone can hide a model that misses most real churners.

cross_val_score()

A more honest performance estimate than trusting a single split.

Extend This Project

Try these on your own:

- Try `class_weight="balanced"` on both models and see whether recall on the churned class improves.
- Add a threshold-tuning step — instead of the default 0.5 cutoff, find the probability threshold that maximizes recall while keeping precision above some minimum.
- Engineer one more derived feature of your own from the raw columns, and check whether it appears near the top of Step 7's own importance ranking.
- Add a third model — a gradient-boosted tree (`GradientBoostingClassifier`) — to the Step 5/6 comparison.

What's Next

Chapter 3: **Movie Recommender** — returning to dsproj1-5's own ratings dataset, but with a genuinely different question: not what can be learned from the data, but what should be recommended from it.

CHAPTER 2 QUICK REFERENCE

- Delivers directly on dsproj1-7's own deferred promise — full ml1 depth: real feature engineering, model comparison, cross-validation, precision/recall/F1
- Logistic regression (ml1-5) vs. random forest (ml1-7), fit on identical data for an honest comparison
- The accuracy paradox (ml1-6) caught concretely in real, imbalanced churn data — recall on the minority class matters more than overall accuracy

- cross_val_score() (ml1-8) gives a more honest performance estimate than one train/test split alone
- Neither model is simply "better" — the right choice trades off interpretability against raw predictive performance

CHAPTER 3 OF 8

Movie Recommender: A New Question for dsproj1-5's Own Dataset

Data Science & ML Projects (Intermediate)

Chapter 3 · Movie Recommender: A New Question for dsproj1-5's Own Dataset

> `dsproj1-5` *asked "what can we learn about this data?" — a real EDA, on this exact MovieLens-style dataset. This chapter returns to the identical data and asks a fundamentally different kind of question: "given that someone liked this movie, what should we recommend next?" Not a label to predict, not a pattern to describe — a ranked, personalized output. Recommendation is a genuinely different problem shape than anything* `ml1` *covered.*

What We're Building

A "users who liked this also liked..." recommender, using item-based collaborative filtering — recommending movies not by their genre or plot, but by whether the *same users* tended to rate them similarly.

Step 1: Loading & Merging — dsproj1-5's Own First Step

```
import pandas as pd

movies = pd.read_csv("movies.csv")
ratings = pd.read_csv("ratings.csv")
df = ratings.merge(movies, on="movie_id")
```

Step 2: Reshaping Into a User-Item Matrix

```
matrix = df.pivot_table(index="user_id", columns="title", values="rating")
print(matrix.shape)
print(matrix.iloc[:5, :5])
```

A GENUINELY NEW DATA SHAPE

Every earlier chapter kept data in long, "tidy" format — one row per rating. Recommendation needs the opposite: one row per *user*, one column per *movie*, the rating (or a missing value) at each intersection. `.pivot_table()` is the reshape this specific problem genuinely requires, not a stylistic choice.

Step 3: An Honest Simplifying Assumption

```
matrix_filled = matrix.fillna(0)
```

0 DOESN'T MEAN "DISLIKED" — IT MEANS "UNRATED"

Most cells in this matrix are missing — no user has rated most movies. Filling with 0 is the standard, honest first approach at this scale, but it's a genuine approximation, not a claim that an unrated movie was actively disliked. Real production recommenders use more sophisticated handling of missing ratings; this project uses the simpler, more transparent version deliberately, so the actual mechanism stays visible.

Step 4: Cosine Similarity — nlp1-5's Own Idea, on Movies Instead of Words

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

similarity = cosine_similarity(matrix_filled.T)
similarity_df = pd.DataFrame(similarity, index=matrix_filled.columns,
columns=matrix_filled.columns)
```

`nlp1-5` used cosine similarity to say two *words* are similar when their vectors point in similar directions. Here, a movie's own "vector" is the column of every user's rating for it — two movies are similar when the same users tended to rate them similarly, regardless of whether those users happen to rate everything high or everything low overall. It's the identical mathematical idea, applied to a completely different kind of vector.

Step 5: Building the Recommender

```
def recommend(title, top_n=5):
    if title not in similarity_df.columns:
        return f"'{title}' not found in this dataset."
    scores = similarity_df[title].sort_values(ascending=False)
    return scores.iloc[1:top_n + 1]   # [0] is the movie itself, always similarity 1.0
```

`scores.iloc[1:]` skips the movie's own perfect self-similarity — a small but easy detail to overlook.

Step 6: Trying It on a Real Movie

```
print(recommend("The Matrix"))
# The Matrix Reloaded       0.71
# Terminator 2               0.64
# Inception                  0.61
# ...
```

Worth a sanity check against `dsproj1-5`'s own genre data: recommendations clustering around genuinely similar genres, without genre ever being used to compute them, is a real, satisfying confirmation the similarity signal is picking up something meaningful in actual user behavior.

DSPROJ1-5'S OWN STATISTICAL TRAP, BACK AGAIN IN A NEW FORM

A movie rated by only two or three users can appear artificially "similar" to another niche movie purely by coincidence — the exact small-sample-extremes problem `dsproj1-5` caught in raw averages resurfaces here in similarity scores. A movie with a genuinely tiny rating count showing up as a "top match" is worth the same skepticism `dsproj1-5`'s own `num_ratings` filter applied.

The Complete Recommender

```
import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity

movies = pd.read_csv("movies.csv")
ratings = pd.read_csv("ratings.csv")
df = ratings.merge(movies, on="movie_id")

matrix = df.pivot_table(index="user_id", columns="title", values="rating").fillna(0)
similarity_df = pd.DataFrame(
    cosine_similarity(matrix.T), index=matrix.columns, columns=matrix.columns
)

def recommend(title, top_n=5):
    if title not in similarity_df.columns:
        return f"'{title}' not found."
    return similarity_df[title].sort_values(ascending=False).iloc[1:top_n + 1]

print(recommend("The Matrix"))
```

THE COLD-START PROBLEM — AN HONEST LIMITATION

A brand-new movie with zero ratings has an empty column in the matrix — nothing for this method to compute similarity from at all. Collaborative filtering fundamentally needs behavior data to work; it can say nothing about

anything nobody has rated yet. This is a genuine, well-known limitation of the whole approach, not a bug in this specific implementation.

pivot_table()

Reshapes long ratings data into a wide user-item matrix.

Cosine similarity on columns

nlp1-5's own word-similarity idea, applied to movie rating patterns.

Item-based collaborative filtering

Recommends by shared rater behavior, not by genre or content.

The cold-start problem

No behavior data means no recommendation — a real, fundamental limit.

Extend This Project

Try these on your own:

- Build a content-based alternative using genre overlap instead of ratings, and compare its recommendations for the same movie.
- Add a minimum-rating-count filter (per dsproj1-5's own finding) before trusting any similarity score.
- Build a simple user-based version: find the most similar *users* to a given one, and recommend movies they rated highly that this user hasn't seen.
- Try Pearson correlation instead of cosine similarity and see whether the recommendations for the same movie change.

What's Next

Chapter 4: **Digit Recognizer** — this course's own first real CNN, applying nn1-7's own convolutional material to real image classification for the first time in any Projects course.

CHAPTER 3 QUICK REFERENCE

- Same dataset as dsproj1-5, a genuinely different question — recommendation instead of exploration
- `.pivot_table()` reshapes long ratings data into a wide user-item matrix — a real, necessary shape change
- Cosine similarity applies nlp1-5's own "similar vectors" idea to rating-pattern vectors instead of word vectors
- Filling missing ratings with 0 is an honest simplifying assumption, not a claim about genuine dislike
- dsproj1-5's own small-sample-extremes trap resurfaces here as artificially high similarity for rarely-rated movies

- The cold-start problem is a fundamental, well-known limitation of collaborative filtering, not a fixable bug

CHAPTER 4 OF 8

Digit Recognizer: A First Real CNN

Data Science & ML Projects (Intermediate)

Chapter 4 · Digit Recognizer: A First Real CNN

> `nn1-7` *built the theory — convolution, kernels, feature maps, pooling — and told AlexNet's own real 2012 story. Every project in this track so far has worked with tabular data or text. This chapter is the first time any Projects course points a real CNN at real images.*

What We're Building

A digit recognizer for MNIST — 70,000 real handwritten digit images (0-9), the classic "hello world" of computer vision, comparable in stature to Iris for classification or IMDb for sentiment. It ships directly through `torchvision`, needs no cleaning, and is small enough to train in a reasonable time even without a GPU.

Step 1: Loading the Real Dataset

```
import torch
from torchvision import datasets, transforms

transform = transforms.ToTensor()
train_data = datasets.MNIST(root="data", train=True, download=True, transform=transform)
test_data = datasets.MNIST(root="data", train=False, download=True, transform=transform)

print(len(train_data), len(test_data))
print(train_data[0][0].shape)   # torch.Size([1, 28, 28]) — 1 grayscale channel, 28x28 pixels
```

Step 2: Looking at the Actual Images

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 5, figsize=(10, 2))
for i, ax in enumerate(axes):
    image, label = train_data[i]
```

```
    ax.imshow(image.squeeze(), cmap="gray")
    ax.set_title(str(label))
    ax.axis("off")
plt.savefig("sample_digits.png")
```

Worth seeing before building anything — these are genuinely messy, human-written digits, not clean printed text. A model that works here has to tolerate real handwriting variation, not just recognize one fixed font.

Step 3: The CNN — nn1-7's Own Architecture, Applied

```
import torch.nn as nn

class DigitCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2)
        self.fc = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))   # 28x28 -> 14x14
        x = self.pool(torch.relu(self.conv2(x)))   # 14x14 -> 7x7
        x = x.view(x.size(0), -1)             # flatten for the final linear layer
        return self.fc(x)
```

Per `nn1-7`: each `Conv2d` layer slides a small learned kernel across the image, extracting local features (edges, curves, strokes) rather than treating each pixel independently. `MaxPool2d(2)` halves the spatial size after each convolution, keeping the strongest signal from each small region — exactly the convolution-then-pooling pattern that chapter covered, applied here to real pixels instead of illustrative diagrams.

Step 4: Training — Batching, Reused From Chapter 1

```
from torch.utils.data import DataLoader

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64)

model = DigitCNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()
```

```
for epoch in range(3):
    for images, labels in train_loader:
        optimizer.zero_grad()
        loss = loss_fn(model(images), labels)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} done.")
```

The same `DataLoader`-based batching pattern `dsproj2-1` already introduced — no new concept here, just applied to image tensors instead of encoded text sequences.

Step 5: Evaluating

```
correct, total = 0, 0
with torch.no_grad():
    for images, labels in test_loader:
        preds = model(images).argmax(dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

print(f"Test accuracy: {correct / total:.2%}")   # typically well above 98%
```

Step 6: Looking at What It Actually Gets Wrong

```
wrong = []
with torch.no_grad():
    for images, labels in test_loader:
        preds = model(images).argmax(dim=1)
        mismatches = (preds != labels).nonzero()
        for idx in mismatches[:5]:
            i = idx.item()
            wrong.append((images[i], labels[i].item(), preds[i].item()))
        if wrong:
            break

fig, axes = plt.subplots(1, len(wrong), figsize=(10, 2))
for ax, (img, actual, predicted) in zip(axes, wrong):
    ax.imshow(img.squeeze(), cmap="gray")
    ax.set_title(f"actual {actual}, predicted {predicted}")
    ax.axis("off")
plt.savefig("misclassified.png")
```

A REAL, SATISFYING QUALITATIVE CHECK

Most of the model's own mistakes turn out to be genuinely ambiguous handwriting — a badly-formed 4 that looks like a 9, a sloppy 7 that resembles a 1 — the same kind of confusion a human might have on the same image, not arbitrary or nonsensical errors. That pattern is itself a real, reassuring signal about what the model actually learned.

HONEST NOTE: MNIST DOESN'T PROVE CONVOLUTION IS ALWAYS NECESSARY

MNIST is famously "easy" — a plain fully-connected network with no convolution at all can also reach fairly high accuracy on this specific dataset, since digits are small, centered, and low-resolution. Convolution's own real advantage — detecting a feature (an edge, a curve) regardless of exactly where it sits in the image — matters far more on larger, more realistic images, where a fully-connected network would need to separately learn every feature at every possible position. MNIST is the right dataset to learn the CNN pattern on; it isn't the dataset that proves why the pattern matters.

Conv2d + MaxPool2d

nn1-7's own convolution-then-pooling pattern, applied to real pixels.

Reused batching

The exact DataLoader pattern from Chapter 1, now on image tensors.

Qualitative error review

Looking at actual misclassified images, not just a single accuracy number.

Convolution's real advantage

Position-independent feature detection — most valuable on harder, larger images.

Extend This Project

Try these on your own:

- Build a plain fully-connected network (no Conv2d at all) on the same flattened data and compare its accuracy to this chapter's own CNN.
- Add a third convolutional layer and see whether accuracy improves further, or plateaus.
- Add `nn.Dropout` (per nn1-6's own regularization material) between the flatten step and the final linear layer.
- Try the identical architecture on `torchvision.datasets.FashionMNIST` — a harder, same-shaped dataset of clothing images — and see how much accuracy drops.

What's Next

Chapter 5: **Weather Forecaster** — returning to dsproj1-3's own logged weather data, this time treating it as a genuinely different task shape: predicting tomorrow from today, not classifying independent rows.

CHAPTER 4 QUICK REFERENCE

- The Projects track's first real CNN — nn1-7's own theory, applied to real image data for the first time
- Conv2d extracts local features via learned kernels; MaxPool2d reduces spatial size while keeping the strongest signal
- Batching reused directly from dsproj2-1 — the same DataLoader pattern, now on images instead of text
- Reviewing actual misclassified digits reveals genuinely ambiguous handwriting, not arbitrary errors — a real signal the model learned something sensible
- **Honest note:** MNIST is easy enough that a plain fully-connected network can also do reasonably well — convolution's real advantage shows up most on harder, larger images

CHAPTER 5 OF 8

Weather Forecaster: Predicting Tomorrow From dsproj1-3's Own Log

Data Science & ML Projects (Intermediate)

Chapter 5 · Weather Forecaster: Predicting Tomorrow From dsproj1-3's Own Log

> *Every model built so far in this course — churn, movies, digits — treated each row as independent. Shuffle them freely; nothing breaks. This chapter's data isn't like that. It's `dsproj1-3`'s own growing weather log, and the entire point is that today depends on yesterday. A genuinely different problem shape, with its own genuinely different rules.*

What We're Building

A next-day temperature forecaster, built by turning a sequential problem into an ordinary regression problem through feature engineering — teaching a model to predict today's temperature from the last few days' own values, then reusing that same model to genuinely forecast tomorrow.

Step 1: Loading the Log, In Order

```
import pandas as pd

df = pd.read_csv("weather_log.csv", parse_dates=["time"])
df = df.sort_values("time").reset_index(drop=True)
```

THE RULE THAT CHANGES EVERYTHING ABOUT THIS CHAPTER

Every earlier chapter's own data could be shuffled without consequence — a customer row, a digit image, a movie rating, none of them cared about order. This dataset's own entire value comes from its order. Sorting by `time` explicitly, and never shuffling this DataFrame at any later step, is the one rule this whole chapter depends on.

Step 2: Resampling to Daily Granularity

```
daily = df.set_index("time").resample("D")["temperature"].mean().dropna().reset_index()
```

`dsproj1-4`'s own `.resample()` tool, reused directly — `dsproj1-3`'s own log is hourly; a next-day forecast is a daily-granularity problem.

Step 3: Engineering Lag Features

```
daily["temp_lag1"] = daily["temperature"].shift(1)
daily["temp_lag2"] = daily["temperature"].shift(2)
daily["temp_lag3"] = daily["temperature"].shift(3)
```

`.shift(1)` pulls each row's value from one position earlier into the current row — `temp_lag1` on today's row literally *is* yesterday's temperature, now sitting as an input feature a regular regression model can use. This is the actual trick that turns a sequential forecasting problem into an ordinary tabular one.

Step 4: The Honest, Structural Missing Values This Creates

```
daily = daily.dropna().reset_index(drop=True)
```

The very first rows have no earlier days to pull a lag value from at all — `temp_lag3` is undefined for day 1, 2, and 3. This is a structural byproduct of the lag-feature technique itself, not messy data the way `dsproj1-2`'s own missing values were — nothing to fill or investigate, just a real, small, unavoidable cost of the method.

Step 5: The Split — Why Random Would Be Wrong Here

```
split_point = int(len(daily) * 0.8)
train = daily.iloc[:split_point]
test = daily.iloc[split_point:]
```

ML1-2'S OWN TRAIN_TEST_SPLIT() WOULD BE A REAL MISTAKE HERE

Every classification/regression project so far in this course correctly used a random split. Here, a random split would let the model train on a later day and get tested on an earlier one — effectively letting it "see the future" relative to what it's being tested on, producing a misleadingly good score that would never hold up in real forecasting. The correct split for any time-ordered problem is chronological: train on the earliest data, test only on data that comes strictly after it in time.

Step 6: Fitting the Model

```
from sklearn.linear_model import LinearRegression

features = ["temp_lag1", "temp_lag2", "temp_lag3"]
model = LinearRegression()
model.fit(train[features], train["temperature"])
```

`ml1-3`'s own linear regression, applied here without modification — the model itself doesn't know or care that its inputs happen to be lagged values; that's entirely a property of how the features were engineered in Step 3.

Step 7: Evaluating — Against a Genuinely Honest Baseline

```
from sklearn.metrics import mean_absolute_error

predictions = model.predict(test[features])
model_mae = mean_absolute_error(test["temperature"], predictions)

naive_predictions = test["temp_lag1"]   # "tomorrow will be the same as today"
naive_mae = mean_absolute_error(test["temperature"], naive_predictions)

print(f"Model MAE: {model_mae:.2f}")
print(f"Naive baseline MAE: {naive_mae:.2f}")
```

THE REAL STANDARD FOR "DOES THIS MODEL ACTUALLY HELP"

`ml1-4`'s own MAE metric, applied against a genuinely honest comparison: a "naive" forecast that just predicts tomorrow will match today. If the trained model's own MAE isn't meaningfully lower than this trivial baseline, it isn't actually adding predictive value — a real, standard sanity check in forecasting that a model beating "chance" or "average" (the usual bar elsewhere in this course) isn't automatically enough here.

The Complete Forecaster

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

df = pd.read_csv("weather_log.csv", parse_dates=["time"]).sort_values("time")
daily = df.set_index("time").resample("D")["temperature"].mean().dropna().reset_index()

for lag in (1, 2, 3):
```

```
    daily[f"temp_lag{lag}"] = daily["temperature"].shift(lag)
daily = daily.dropna().reset_index(drop=True)

split = int(len(daily) * 0.8)
train, test = daily.iloc[:split], daily.iloc[split:]
features = ["temp_lag1", "temp_lag2", "temp_lag3"]

model = LinearRegression().fit(train[features], train["temperature"])
model_mae = mean_absolute_error(test["temperature"], model.predict(test[features]))
naive_mae = mean_absolute_error(test["temperature"], test["temp_lag1"])

print(f"Model MAE: {model_mae:.2f} vs. naive baseline: {naive_mae:.2f}")
```

Order-dependent data

Never shuffled — the whole problem depends on strict chronological sequence.

Lag features via .shift()

Turns a sequential forecasting problem into an ordinary tabular one.

Chronological split

Train on earlier data, test only on strictly later data — never random here.

The naive baseline

"Tomorrow equals today" — the real bar a forecasting model must beat.

Extend This Project

Try these on your own:

- Add `precipitation` as an extra lagged feature alongside temperature and see whether MAE improves.
- Try a `RandomForestRegressor` in place of `LinearRegression` and compare against both the linear model and the naive baseline.
- Extend the forecast horizon — predict two days ahead instead of one, and see how much the naive baseline's own advantage shrinks or grows.
- Once you've taken `nn1-8`, try replacing this chapter's lag-feature approach with a real LSTM sequence model and compare MAE against both approaches here.

What's Next

Chapter 6: **Named Entity Extractor** — applying nlp1-7's own NER material to a real batch-document extraction tool, a structural sequel to Chapter 1's own whole-document sentiment classification.

CHAPTER 5 QUICK REFERENCE

- A genuinely different problem shape — order-dependent, never shuffled, unlike every prior classification/regression project in this course
- `.shift()` creates lag features, turning sequential forecasting into ordinary tabular regression
- The train/test split must be chronological here — ml1-2's own default random split would leak future information
- ml1-4's own MAE, evaluated against a genuinely honest "tomorrow equals today" baseline, not just a bare number
- ml1-3's own linear regression, reused completely unmodified — only the feature engineering around it changed

CHAPTER 6 OF 8

Named Entity Extractor: Structured Information From Real Documents

Data Science & ML Projects (Intermediate)

Chapter 6 · Named Entity Extractor: Structured Information From Real Documents

> *`nlp1-7` built a BiLSTM tagger on a five-word toy sentence. Chapter 1 of this course already proved the pattern for sentiment: take `nlp1`'s own architecture unchanged, point it at real data. This chapter does that again for NER — and it's a structural sequel to Chapter 1 specifically, since tagging every token is a genuinely different shape of problem than classifying a whole document.*

What We're Building

An entity extractor identifying people, organizations, and locations across a real batch of documents — trained on CoNLL-2003, the canonical NER benchmark (comparable in stature to MNIST for images or IMDb for sentiment), then run over genuinely new text to produce structured, per-document extracted-entity output.

Step 1: A Real Labeling Scheme, More Detailed Than nlp1-7's Own

```
# CoNLL-2003 format: one token and tag per line
Sarah    B-PER
Chen     I-PER
works    O
at       O
New      B-ORG
York     I-ORG
Times    I-ORG
```

A REAL COMPLICATION NLP1-7'S OWN SIMPLIFIED TAGS SKIPPED

`nlp1-7` tagged each entity token with a single label (`PER`, `ORG`, `LOC`). Real NER data uses **IOB tagging**: `B-` marks the beginning of an entity, `I-` marks a continuation of the same entity, and `O` marks "not an entity." Without this distinction, two adjacent single-token entities ("Paris" then "London," two separate places) would be

> indistinguishable from one genuine two-token entity ("New York," one place) — the B-/I- split is what makes multi-word entities representable at all.

Step 2: Vocabulary — Reusing Chapter 1's Own Approach

```
from collections import Counter

all_tokens = [tok for sentence in sentences for tok, tag in sentence]
vocab = {word: i + 2 for i, (word, _) in enumerate(Counter(all_tokens).most_common(15000))}
vocab["<pad>"], vocab["<unk>"] = 0, 1

tag_set = sorted({tag for sentence in sentences for _, tag in sentence})
tag2idx = {tag: i for i, tag in enumerate(tag_set)}
```

Same frequency-capped vocabulary pattern `dsproj2-1` established — plus a second small vocabulary, this time for the tag set itself.

Step 3: The Exact BiLSTM Tagger From nlp1-7

```
import torch.nn as nn

class SequenceLabeler(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_tags):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True, bidirectional=True)
        self.output = nn.Linear(hidden_dim * 2, num_tags)

    def forward(self, x):
        embedded = self.embedding(x)
        outputs, _ = self.lstm(embedded)
        return self.output(outputs)   # one prediction per token, per nlp1-7
```

Character for character `nlp1-7`'s own architecture — every hidden state kept, bidirectional for exactly the "Washington could be a place or a person" reason that chapter demonstrated.

Step 4: Training — A Real Batching Gotcha Toy Data Never Surfaced

```
import torch
```

```
loss_fn = nn.CrossEntropyLoss(ignore_index=tag2idx["<pad>"])

for epoch in range(5):
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        logits = model(X_batch)                    # (batch, seq_len, num_tags)
        loss = loss_fn(logits.view(-1, len(tag_set)), y_batch.view(-1))
        loss.backward()
        optimizer.step()
```

A GOTCHA SPECIFIC TO SEQUENCE LABELING AT BATCH SCALE

Padding shorter sentences to a fixed length (needed for batching, per Chapter 1) means the padded positions have no real tag at all — without `ignore_index=tag2idx["<pad>"]`, the loss function would be penalized for its own predictions on positions that were never real tokens to begin with, actively teaching the model to care about padding. A single toy sentence, batch size one, never surfaces this problem — it only appears once real batches mix sentences of different lengths.

Step 5: Evaluating — The Accuracy Paradox, Again, in a New Form

```
from sklearn.metrics import classification_report

print(classification_report(all_true_tags, all_predicted_tags))
```

DSPROJ2-2'S OWN LESSON, BACK AGAIN

Most tokens in any real sentence are tagged `O` — not part of any entity at all. A model that predicted `O` for every single token would score deceptively high raw accuracy while catching zero real entities, exactly `dsproj2-2`'s own accuracy-paradox finding, now showing up in a per-token task instead of a per-customer one. Per-entity-type precision and recall (from `classification_report`) are what actually matter here, not overall token accuracy.

Step 6: Extracting Structured Output From New Documents

```
def extract_entities(text, model, vocab, idx2tag):
    tokens = text.split()
    ids = torch.tensor([[vocab.get(t, 1) for t in tokens]])
    predicted_tags = model(ids).argmax(dim=-1)[0]

    entities = []
    current = []
    for token, tag_idx in zip(tokens, predicted_tags):
        tag = idx2tag[tag_idx.item()]
        if tag.startswith("B-"):
```

```
            if current:
                entities.append(" ".join(current))
            current = [token]
        elif tag.startswith("I-") and current:
            current.append(token)
        else:
            if current:
                entities.append(" ".join(current))
            current = []
    return entities
```

This is the genuine payoff of the B-/I- scheme from Step 1 — reassembling consecutive `B-` / `I-` tokens back into whole multi-word entity strings, turning a stream of per-token tags into the structured, readable output the chapter's own title promised.

HONEST LIMITATION: DOMAIN SHIFT

A model trained on CoNLL-2003's own news-style text learned news-style entity patterns specifically. Pointed at a genuinely different kind of document — casual social posts, technical manuals, legal text — real accuracy can drop meaningfully, since the patterns that identify an entity in one writing style don't automatically transfer to another. This is a real, well-documented limitation of any trained model, not a flaw specific to this implementation.

IOB tagging

B-/I-/O distinguishes entity starts from continuations — required for multi-word entities.

ignore_index in the loss

Stops padding positions from corrupting real training signal.

Per-entity metrics, not token accuracy

The accuracy paradox, reappearing in a per-token task.

Reassembling tags into entities

Turning a per-token prediction stream back into structured output.

Extend This Project

Try these on your own:

- Run `extract_entities()` over several real documents and build a summary table counting entity mentions by type across the whole batch.
- Add a MISC tag category (CoNLL-2003 includes one) and see how it affects the per-entity-type metrics.
- Test the trained model on a document type genuinely different from news text, and measure how much real accuracy actually drops.

- Compare this from-scratch model's accuracy against a pretrained NER pipeline from a library like spaCy — a real, honest build-vs-use comparison.

What's Next

Chapter 7: **Using an LLM API as a Data Labeling Tool** — a light, deliberate touch of llm1/claude-adv1, using an LLM as one stage inside a larger data pipeline rather than training a model from scratch.

CHAPTER 6 QUICK REFERENCE

- nlp1-7's own exact BiLSTM tagger architecture, applied to the real CoNLL-2003 benchmark — a structural sequel to this course's own Chapter 1
- IOB tagging (B-/I-/O) is a real complication beyond nlp1-7's own simplified single-label tags — required to represent multi-word entities
- `ignore_index` on the loss function prevents padding positions from corrupting training — a batching gotcha toy data never surfaces
- dsproj2-2's own accuracy paradox reappears here — per-entity precision/recall matters more than raw token accuracy
- Domain shift is a real, honest limitation — a news-trained model won't automatically generalize to every document type

CHAPTER 7 OF 8

Using an LLM API as a Data Labeling Tool

Data Science & ML Projects (Intermediate)

Chapter 7 · Using an LLM API as a Data Labeling Tool

> *Every project so far in this course trained something — an LSTM, a random forest, a CNN, a BiLSTM tagger. This chapter trains nothing. It uses an already-trained LLM, through its API, as one stage inside a larger data pipeline — a genuinely different, genuinely modern data-science technique, distinct from anything a from-scratch model in this course could do alone.*

What We're Building

A pipeline that labels a large batch of unstructured customer-feedback text with a custom business category ("Billing Issue," "Bug Report," "Feature Request," "Praise," "Other") — categories with no existing labeled training data anywhere, making `dsproj2-1`'s own from-scratch approach genuinely impractical here. The LLM applies the labeling scheme directly, at scale, the moment it can be described in a prompt — then the results get analyzed statistically with this track's own established tools.

Step 1: The Unlabeled Batch

```
import pandas as pd

feedback = pd.read_csv("customer_feedback.csv")   # one column: raw text, no labels at all
print(feedback.shape)
```

Step 2: Designing the Labeling Prompt

```
CATEGORIES = ["Billing Issue", "Bug Report", "Feature Request", "Praise", "Other"]

PROMPT_TEMPLATE = """Classify the following customer feedback into exactly one of these categories:
{categories}.

Feedback: "{text}"
```

```
Respond with only valid JSON: {{"category": "...", "confidence": "high"|"medium"|"low"}}"""
```

THIS IS PROMPT1'S OWN MATERIAL, APPLIED

Clarity, context, and an explicit output format — `prompt1`'s own anatomy of a good prompt, put directly to use rather than re-taught here. Asking for structured JSON specifically, rather than a free-form sentence, is what makes Step 4's own parsing step possible at all.

Step 3: Calling the API — claude-adv1's Own Territory, Applied Minimally

```
import anthropic, json, time

client = anthropic.Anthropic()   # API key read from environment, per claude-adv1

def label_feedback(text):
    response = client.messages.create(
        model="claude-sonnet-4-5",
        max_tokens=100,
        messages=[{"role": "user", "content": PROMPT_TEMPLATE.format(categories=CATEGORIES,
text=text)}],
    )
    time.sleep(0.5)
    return json.loads(response.content[0].text)
```

The full API surface — streaming, tool use, system prompts — is `claude-adv1`'s own territory in depth. This chapter uses exactly one call shape, deliberately, since the point here is what the API call accomplishes inside a larger pipeline, not the API itself.

A GENUINELY DIFFERENT COST MODEL THAN EVERY EARLIER API CHAPTER

`dsproj1-3`'s and `dsproj1-6`'s own APIs (Open-Meteo, REST Countries) were free, with no per-call cost. A real LLM API call has a real, per-token price, and rate limits that matter at genuine scale — 10,000 feedback rows means 10,000 real billed calls, not a rounding error. This is a real, practical planning consideration specific to this chapter, not present anywhere else in this track.

Step 4: Running the Batch and Parsing Results

```
results = []
for text in feedback["text"]:
    try:
```

```
        label = label_feedback(text)
        results.append({"text": text, **label})
    except (json.JSONDecodeError, KeyError):
        results.append({"text": text, "category": "parse_error", "confidence": None})

labeled = pd.DataFrame(results)
```

Same JSON-parsing reflex `ds1-3` established, now applied to a live model's own output instead of a file or a conventional API response — and the same defensive `try`/`except` instinct this track has used since `dsproj1-1`'s own scraper, since a model occasionally returning malformed JSON is a real, expected possibility, not a bug to be surprised by.

Step 5: Analyzing the Labeled Data — This Track's Own Familiar Tools

```
import matplotlib.pyplot as plt

category_counts = labeled["category"].value_counts()
category_counts.plot(kind="barh", color="#15803D")
plt.title("Feedback Volume by Category")
plt.savefig("feedback_categories.png")
```

Nothing new here at all — `.value_counts()` and a bar chart, the exact `ds1`-level analysis skill this whole track opened with, applied here to data an LLM helped create rather than data that arrived pre-labeled.

The Honest Trade-Off, Stated Directly

	DSPROJ2-1'S OWN FROM-SCRATCH MODEL	THIS CHAPTER'S LLM-AS-TOOL
Training data needed	25,000 labeled examples	None — the category scheme just needs describing
Cost per use once built	Free — a trained model runs at no marginal cost	Real per-call cost, every single time
Measured accuracy	A real number, against a held-out test set	No ground truth to measure against, unless one is built
Adapting to a new category scheme	Requires retraining on newly labeled data	Edit the prompt

HONEST LIMITATION: NO MEASURED ACCURACY, AND REAL HALLUCINATION RISK

Unlike every model trained earlier in this course, there's no test-set accuracy number for this chapter's own labels — no pre-existing ground truth exists to compare against. Per `llm1-10`'s own mechanical explanation, an LLM has no built-in fact-checker; it can mislabel ambiguous feedback confidently, the same way it can hallucinate anywhere else. The real, practical mitigation is what production pipelines actually do: pull a random sample of the labeled output and have a human spot-check it, the same "flag for review, don't blindly trust" instinct `dsproj1-2`'s own outlier-flagging and `dsproj1-6`'s own missing-value handling already established in this track.

No training data required

A custom category scheme works the moment it can be described in a prompt.

Structured output via JSON

Requesting a specific format makes the model's own output immediately parseable.

Real per-call cost

Unlike every free API used earlier in this track — a genuine planning factor at scale.

Sample-based spot-checking

The practical substitute for a held-out test-set accuracy number.

Extend This Project

Try these on your own:

- Pull a random sample of 50 labeled rows, label them by hand, and compute real agreement between your labels and the model's own — a genuine, honest accuracy estimate.
- Add a retry-with-backoff loop around `label_feedback()` for real API rate-limit errors, not just malformed JSON.
- Ask the model to also extract a one-sentence summary alongside the category, and add it as a new column.
- Compare this chapter's own category-volume chart against Chapter 1's own sentiment scores for the same feedback batch, if both are available — do certain categories skew more negative?

What's Next

Chapter 8: **Capstone — An End-to-End ML-Powered Mini App** — collection, cleaning, feature engineering, model training/evaluation, and a minimal interface, combined into one real application, with a chapter-attribution table spanning both dsproj1 and dsproj2.

CHAPTER 7 QUICK REFERENCE

- The only chapter in this course that trains nothing — an already-trained LLM used as one stage in a larger pipeline

- prompt1's own prompt-anatomy material applied directly, not re-taught; claude-adv1's own API surface used minimally, on purpose
- Real per-call cost and rate limits — a genuinely different consideration than this track's earlier free APIs
- No training data needed for a custom category scheme — but also no measured test-set accuracy, unlike every model trained earlier in this course
- llm1-10's own hallucination material applies directly here — sample-based human spot-checking is the honest, practical mitigation

CHAPTER 8 OF 8

Capstone: An End-to-End ML-Powered Mini App

Data Science & ML Projects (Intermediate)

Chapter 8 · Capstone: An End-to-End ML-Powered Mini App

> *Sixteen chapters across both Projects courses, and every one of them has been a script — producing files, printed output, saved charts. This capstone adds the one piece that's been missing the whole time: a way for someone who isn't running Python to actually use what was built.*

What We're Building

A "Churn Risk Checker" — reusing `dsproj2-2`'s own churn-prediction pipeline in full (collect, clean, engineer features, train, evaluate), then wrapping the trained model in a minimal Streamlit interface: a small web form where a user enters a customer's own details and gets back a churn risk prediction, with no Python knowledge required to use it.

Stage 1 — Collect, Clean, Engineer (dsproj2-2, Reused)

```
import pandas as pd

df = pd.read_csv("telco_churn.csv")
df["avg_monthly_spend"] = df["TotalCharges"] / df["tenure"].replace(0, 1)
df = pd.get_dummies(df, columns=["Contract", "InternetService", "TechSupport"], drop_first=True)

y = (df["Churn"] == "Yes").astype(int)
X = df.drop(columns=["customerID", "Churn", "TotalCharges"])
```

Directly `dsproj2-2`'s own Step 1, unchanged — nothing new to teach here, only to reuse.

Stage 2 — Train & Evaluate (dsproj2-2, Reused)

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
random_state=42)
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
print(classification_report(y_test, model.predict(X_test)))
```

Stage 3 — Persisting the Model (New)

```
import joblib

joblib.dump(model, "churn_model.joblib")
joblib.dump(list(X.columns), "model_columns.joblib")
```

A GENUINELY NEW STEP — NO PRIOR CHAPTER NEEDED THIS

Every earlier chapter's own model lived and died inside a single script run. An interface needs the trained model to persist *between* runs — someone opening the app tomorrow shouldn't have to retrain a random forest first. `joblib.dump()` saves the trained model to disk; saving the exact column order alongside it matters too, since the interface will need to build input rows in that identical order.

Stage 4 — The Minimal Interface

```
# churn_app.py — run with: streamlit run churn_app.py
import streamlit as st
import pandas as pd
import joblib

model = joblib.load("churn_model.joblib")
columns = joblib.load("model_columns.joblib")

st.title("Churn Risk Checker")

tenure = st.slider("Tenure (months)", 0, 72, 12)
monthly_charges = st.slider("Monthly Charges ($)", 0, 150, 70)
contract = st.selectbox("Contract Type", ["Month-to-month", "One year", "Two year"])

if st.button("Check Risk"):
    row = pd.DataFrame([[0] * len(columns)], columns=columns)
    row["tenure"] = tenure
    row["MonthlyCharges"] = monthly_charges
    row["avg_monthly_spend"] = monthly_charges
```

```
    col_name = f"Contract_{contract}"
    if col_name in row.columns:
        row[col_name] = 1

    risk = model.predict_proba(row)[0][1]
    st.metric("Churn Risk", f"{risk:.0%}")
```

Streamlit turns each line into a real, interactive widget — `st.slider()`, `st.selectbox()`, `st.button()` — with no HTML, CSS, or JavaScript required. Building the input row with the exact saved `columns` order (Stage 3) is what lets a few manually-set values slot correctly into the same feature layout the model was trained on.

DELIBERATELY MINIMAL, NOT A SUBSTITUTE FOR FASTAPI1

Streamlit is the real, standard tool data scientists reach for to wrap a model in an interface without becoming a web developer — genuinely appropriate for this chapter's own scope. It is not a substitute for `fastapi1`'s own full web-framework depth: no proper routing, no authentication, no production-grade request handling. For a real, public-facing application, that's the right course to reach for instead.

Chapter Attribution Table — Spanning Both Courses

STAGE	DRAWN DIRECTLY FROM
Collection	dsproj1-1 (scraping), dsproj1-3 (single API), dsproj1-6 (combining APIs)
Cleaning	dsproj1-2's own reusable, report-generating pipeline pattern
EDA	dsproj1-5's own flowing, narrative exploration approach
Feature engineering	dsproj2-2's own derived features and one-hot encoding
Model comparison & evaluation	dsproj2-2's own logistic regression/random forest comparison, cross-validation, and full metric picture
Deep learning (where relevant)	dsproj2-1 (LSTM), dsproj2-4 (CNN), dsproj2-6 (BiLSTM tagger)
Model persistence	New in this capstone — joblib, saving trained state between runs
Interface	New in this capstone — Streamlit, deliberately minimal

SCOPE NOTE — WHAT THIS CAPSTONE DELIBERATELY DOESN'T DO

- No production deployment or hosting — running `streamlit run` locally is not the same as serving a real public application.
- No authentication, input validation hardening, or security review — this interface trusts its own inputs completely.
- No automated testing or CI — every project in both Projects courses has been run and checked by hand.

- No monitoring of the deployed model's own real-world accuracy over time — a genuinely separate discipline (MLOps) this course never claimed to cover.

Model persistence

joblib.dump()/load() let a trained model outlive the script that trained it.

Streamlit widgets

Real interactivity with no web development knowledge required.

Column-order consistency

Saved feature columns keep new input rows aligned with training data.

A real, honest scope boundary

Minimal by design — named explicitly, not glossed over.

Extend This Project

Try these on your own:

- Add the remaining feature inputs (internet service, tech support) as additional Streamlit widgets, matching every column the model was actually trained on.
- Show `feature_importances_` (from dsproj2-2) directly in the app, so a user sees which factors drove their own specific prediction.
- Swap the churn model for the `dsproj2-1` sentiment model, and build a second small Streamlit app around it instead.
- Read Streamlit's own deployment documentation and try hosting this app for real, on Streamlit Community Cloud or similar.

Data Science & ML Subject Complete

That's all 16 chapters of the **Data Science & ML Projects** track — `dsproj1`'s own 8 beginner projects and `dsproj2`'s own 8 intermediate ones — and with it, the complete Data Science & ML subject: Data Science Fundamentals → Machine Learning Fundamentals → Neural Networks & Deep Learning → NLP → LLMs → Projects, six courses built one on top of the last, ending here with a real, working, interactive application built from every piece of it.

CHAPTER 8 QUICK REFERENCE — COURSE & SUBJECT SUMMARY

- Reuses dsproj2-2's own full churn pipeline unchanged, adding the one genuinely new piece: persistence + a minimal interface

- `joblib.dump()` / `load()` let a trained model survive between script runs — necessary for any real interface
- Streamlit widgets provide real interactivity with zero web development knowledge — deliberately minimal, not a fastapi1 substitute
- Chapter-attribution table spans both dsproj1 and dsproj2 — the full track, combined in one final project
- This completes Data Science & ML Projects, Intermediate (8 chapters) and the entire six-course Data Science & ML subject