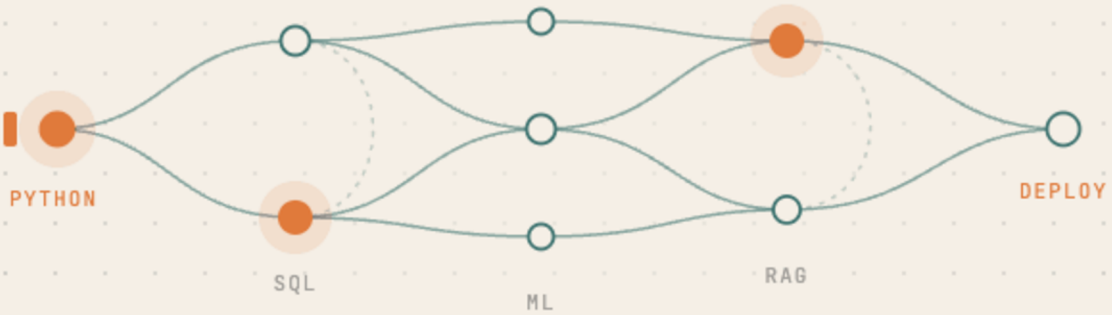
ADSIT PRESS

PREVIEW EDITION

FORWARD DEPLOYED ENGINEER · NOOB → EXPERT

`print("hello world")` → Production AI Systems

The FORWARD DEPLOYED ENGINEER'S Handbook (FDE)



WRITTEN BY

Abhijeet Verma

FIRST EDITION · 13 MODULES · 7 READING PATHS

180 CHAPTERS · FDE-READY

The Forward Deployed Engineer's Handbook

From print("hello world") to Production AI Systems

Abhijeet Verma

2026-07-16

Welcome

The Forward Deployed Engineer's Handbook

From `print("hello world")` to Production AI Systems

This is the complete edition — all 180 chapters across 13 parts, built around the CinemaStream portfolio project. Turn the page to begin, or jump to any chapter from the table of contents.

Chapter 42: The FDE Mindset — From Analyst to Data Engineer

0. Where You Are

You've spent four modules learning to write code, manipulate data in memory, and query databases — including, in Chapter 41a, when to step outside SQL into document stores and key-value caches. You can answer business questions. Module 5 is where that work gets industrialised — you stop running scripts manually and start building systems that run automatically, reliably, and at scale. This chapter doesn't teach a tool. It teaches the shift in thinking that separates an analyst who codes from a data engineer who builds. It's one of the most important chapters in the book.

---

1. The Concept

An **analyst** answers a question. A **data engineer** builds the system that answers that question automatically, correctly, every day, for the next five years, even when the data changes, the schema evolves, and the person who built it leaves the team.

This distinction is everything. Consider the lifecycle of a metric:

1. **Analyst stage**: Priya writes a Python script to calculate monthly churn rate. She runs it manually at the start of each month. It takes 3 minutes. It works.
2. **Engineer stage**: The script runs on a schedule, writes results to a database, sends Rohan an email, logs when it ran, alerts when it fails, handles upstream data delays gracefully, and produces exactly the same answer whether run at 2 a.m. or 2 p.m.

The analyst's job is to prove the metric is meaningful. The data engineer's job is to make it unstoppable.

The FDE (Forward Deployed Engineer) mindset has five principles:

1. Data is a product, not a script. Every dataset you build is consumed by someone — an analyst, a dashboard, an ML model, a billing system. Think of it as an API with consumers. Breaking changes require backward compatibility or migration plans. SLAs matter.

2. Trust is earned through reliability. A metric that's right 90% of the time and mysteriously wrong 10% of the time is worse than no metric. Engineers build in validation, alerting, and idempotency. Analysts build in confidence intervals.

3. Everything fails. Build for failure. Networks time out. APIs return unexpected errors. Upstream schemas change without warning. Files arrive late. The engineer's default assumption is that something will go wrong and the system should handle it gracefully — retry, alert, not silently produce wrong output.

4. Make the implicit explicit. A script that "just works" because of undocumented assumptions about column order, time zones, and row counts is a trap. Document assumptions. Assert them in code. Make the pipeline fail loudly when they're violated rather than silently producing wrong output.

5. Optimise for the reader, not the writer. Code is read far more often than it's written. The clearest, most maintainable implementation is usually the right one. Clever one-liners and over-engineered abstractions both fail this test.

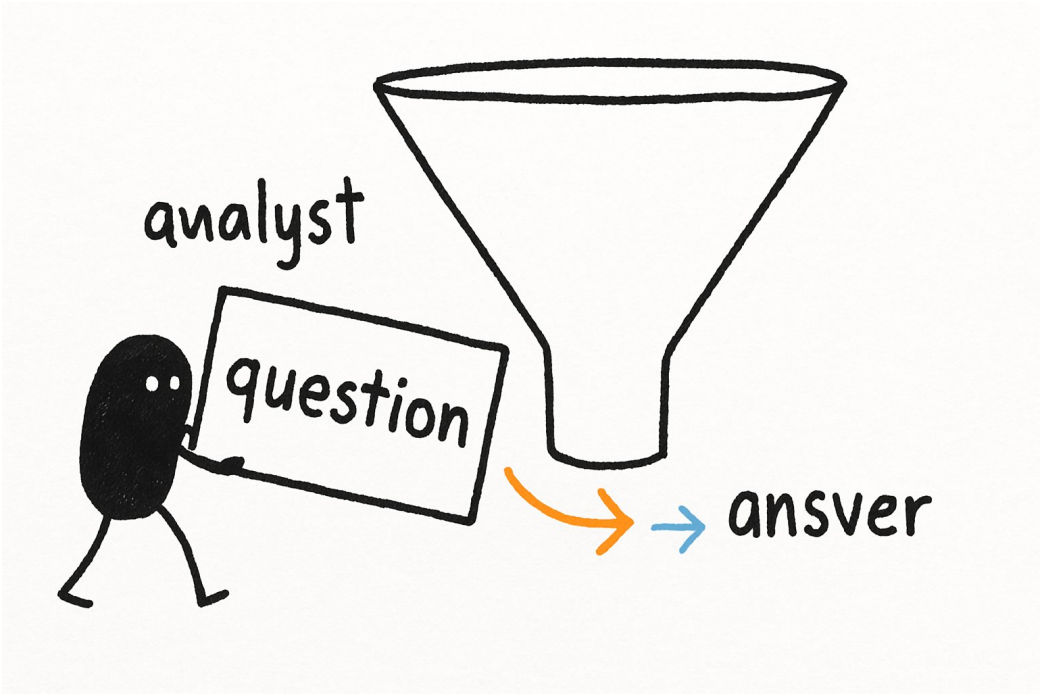
The 80/20 reframe.

If you're building an AI or ML product, here's the number that will surprise every client you work with: the model is 20% of the job. The rest — data pipelines, validation, monitoring, deployment, retraining schedules, feature stores, access control, cost management, documentation — is your job.

This is not a knock on models. A well-trained model is impressive and important. But a model sitting on a laptop that nobody can query, that drifts silently when the data distribution shifts, that nobody knows to retrain when a new country launches — that model is worthless. The engineering that surrounds it is what makes it valuable.

In the chapters ahead you will build every layer of that 80%: orchestration, observability, serving, evaluation, and the harness that makes all of it machine-enforceable. The model is the guest of honour. You're building the venue.

Think: The difference between an analyst and a data engineer is not the tools — it's the question they're answering. An analyst asks "what happened?" A data engineer asks "how do I make sure we can always know what happened?" And the difference between a data engineer and a Forward Deployed Engineer is scope: you're also responsible for the model, the stakeholder, and the business outcome.



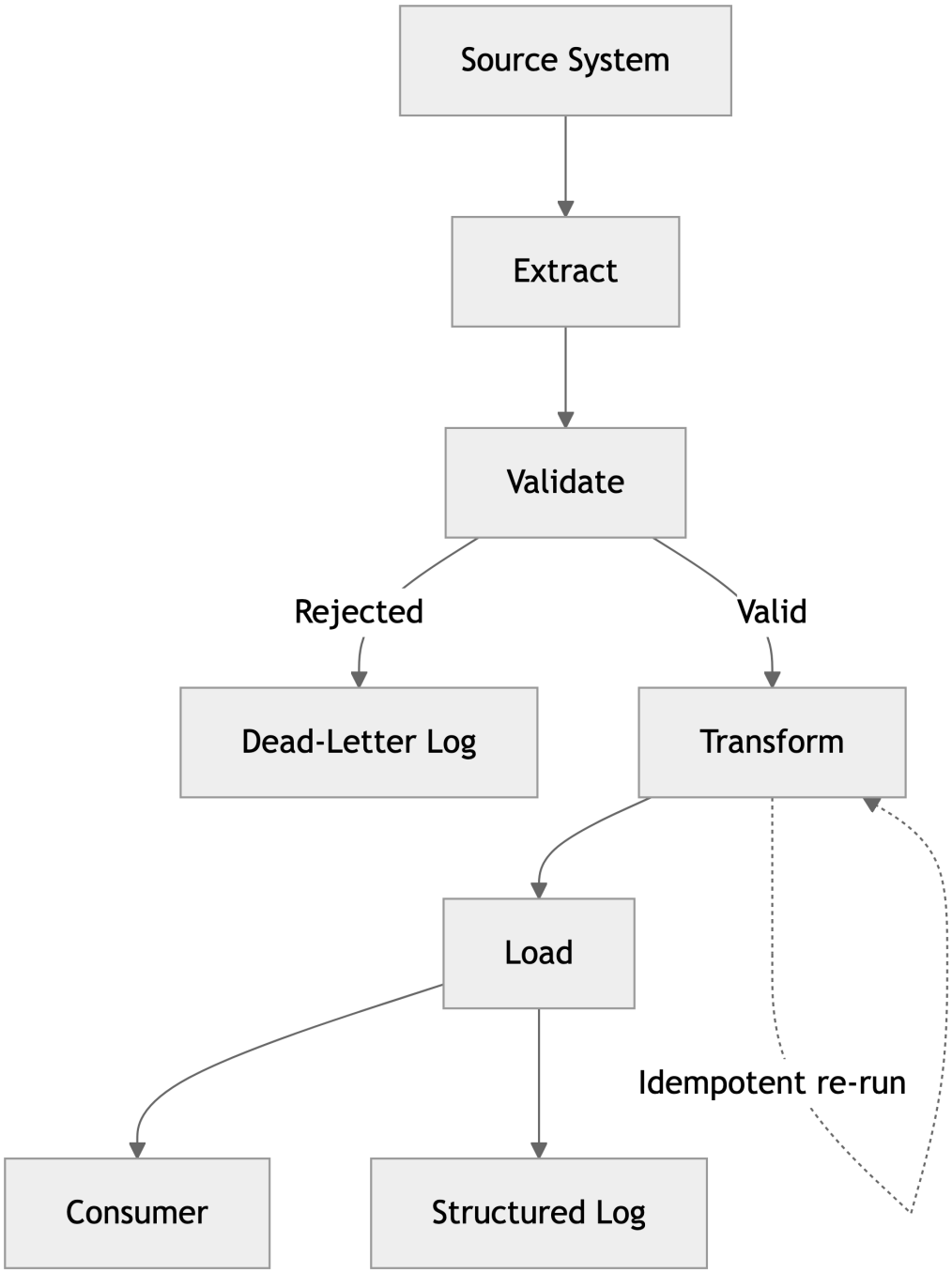


Figure 1: The pipeline mental model: extract, validate, transform, load — rejects dead-lettered, re-runs idempotent

Chapter 67: Supervised Learning — Regression vs Classification, KNN, SVM

0. Where You Are

Welcome to Part 7 — Applied ML for Deployment (Chapters 067–077). In Chapters 66a, 66b, and 66c you built the statistical and mathematical foundations that make Part 7 legible: descriptive stats and distributions, hypothesis testing and A/B design, vectors and matrix operations. Module 8 ended with "Ask Anything" getting a clear architecture for its next iteration; that work is documented and parked. This chapter starts something new: over the next eleven chapters, Mei Tan will mentor you through building, evaluating, and deploying a real **churn prediction model** for CinemaStream — the same kind of model that, in Chapter 077, gets wrapped in a FastAPI endpoint and, in Chapter 093, becomes part of the capstone Data Hub. This chapter lays the foundation: the core distinction between **regression** and **classification**, and your first two classification algorithms — **K-Nearest Neighbors (KNN)** and **Support Vector Machines (SVM)**. Everything here is abstract and algorithm-focused; Section 3 takes the first real step toward the churn model by building a baseline classifier on a synthetic dataset shaped like CinemaStream's real features. Next chapter (068) builds on this with decision trees and ensemble methods — the algorithms that will likely outperform today's baseline.

---

1. The Concept

Before any algorithm, there's a hierarchy of questions a model can answer — and most of what's commonly called "AI" lives at the bottom rung of it. Judea Pearl's **Causal Hierarchy** describes three levels:

Rung	Type	Question	Example
1	Correlation	What is associated with what?	"Ambulance arrivals track with higher mortality"
2	Intervention	What happens if we **do** X?	"If we put every patient in an ambulance — what happens?"
3	Counterfactual	What if X had been different?	"Would the patient have survived had they come by car?"

Every algorithm in this chapter — and almost every model in this book — operates at **Rung 1**. A supervised learning model finds patterns of association between inputs and outputs in historical data. It does not know *why* the association exists, and it cannot, on its own, tell you what would happen if you changed something. That gap matters enormously in production: a model can tell you "users with zero watch activity in the last 30 days usually churn" (Rung 1), but it cannot directly tell you "if we send those users a re-engagement email, will fewer of them churn?" (Rung 2) — that requires an experiment, not just a model. Keep this distinction in your back pocket. It will resurface throughout Part 7, especially in Chapter 072 (Metrics) and again in the FDE chapters, because the gap between Rung 1 and Rung 2 is exactly where a lot of "the model said X, so we did Y, and it didn't work" failures come from.

With that framing in place: **supervised learning** is the family of techniques where a model learns from *labeled examples* — pairs of inputs and known correct outputs — and then predicts outputs for new, unseen inputs. "Supervised" refers to the labels: during training, the model is told the right answer for every example, the way a supervisor corrects a trainee's work. The two flavors of supervised learning split based on what kind of answer you're predicting:

Regression predicts a *continuous number*. "How many minutes will this user watch next week?" "What will next month's revenue be?" "What's the predicted house price given its size?" The output can take any value along a range, and "how close" the prediction is to the true value is itself meaningful — predicting 251 when the true answer is 250 is a tiny error; predicting 50 is a big one.

Classification predicts a *category* — one of a fixed, known set of labels. "Will this user churn next month? (yes/no)" "Is this email spam? (spam/not spam)" "What

genre is this movie? (Action/Drama/Comedy/...)" The output is discrete. There's no "almost right" in the same sense as regression — a prediction of "churn" when the truth is "no churn" is wrong, full stop, regardless of how confident the model was (though that confidence itself is often useful, which is why many classifiers output *probabilities*, not just labels).

The same real-world question can sometimes be framed either way. "How likely is this user to churn?" could be a regression problem (predict a probability between 0 and 1) or a classification problem (predict churn / no-churn, possibly *using* a predicted probability internally and a threshold to convert it to a label). Which framing you choose depends on what the business actually needs to *do* with the answer — a yes/no flag for an automated email trigger, or a ranked list of "highest risk" users for a retention team with limited capacity. This is itself an FDE judgment call, not a purely technical one.

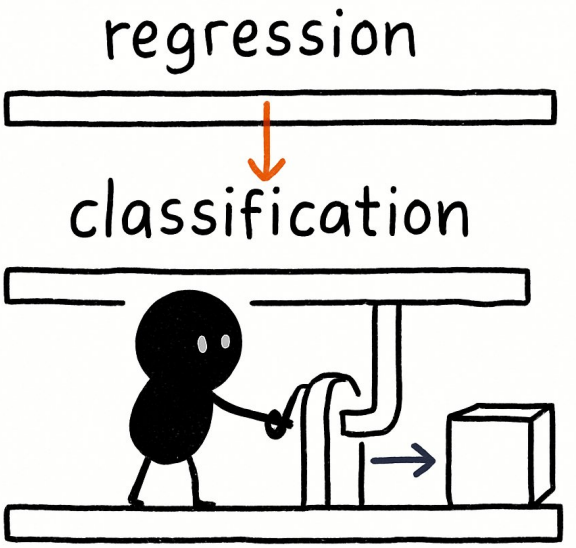
Now, two specific algorithms — both useful primarily for classification, though each has a regression variant.

K-Nearest Neighbors (KNN) is the simplest possible supervised learning idea: to classify a new point, find the **K** training examples that are *closest* to it (by some distance measure — usually Euclidean distance), and let them vote. If 4 of the 5 nearest neighbors are labeled "churn" and 1 is "no churn," KNN predicts "churn." There's no "training" in the usual sense — KNN just memorizes the entire training set and does the comparison at prediction time. This makes it simple to understand and a good baseline, but it has real costs: prediction is slow when the dataset is large (every prediction compares against every training point), and it's extremely sensitive to the *scale* of features — a feature measured in the thousands (like `watch_minutes_avg`) will dominate the distance calculation over a feature measured in single digits (like `tenure_months`) unless you scale them first.

Support Vector Machines (SVM) take a different approach: instead of comparing to neighbors, SVM tries to find the **best separating boundary** between classes — the line (or, in higher dimensions, the *hyperplane*) that divides the two classes with the **widest possible margin** on either side. The training points closest to that boundary — the ones that "support" it, i.e., that would change the boundary's position if removed — are called **support vectors**, which is where the algorithm gets its name. For data that isn't linearly separable (no straight line cleanly divides the classes), SVM can use a **kernel** — a function that implicitly projects the data into a higher-dimensional space where a straight-line separation *does* exist, without actually computing that higher-dimensional space directly (the "kernel trick"). The most common non-linear kernel is the **RBF (radial basis function)** kernel, which can carve out curved decision boundaries.

Both algorithms are **distance-based** — they work by measuring how "close" or "far apart" points are in feature space. That single shared property is the source of both their main strength (they make few assumptions about the shape of the relationship between features and labels) and their main shared pitfall (they're sensitive to feature scaling, which Section 2 will demonstrate directly).

> **Think:** Imagine you're new to a neighborhood and someone asks "is this a good area for families?" KNN's approach: walk to the 5 nearest houses, ask each resident whether they think it's family-friendly, and go with the majority answer. SVM's approach: look at the whole neighborhood at once and draw the clearest possible dividing line between "family" and "not family" blocks — then check which side of that line the new house falls on. KNN asks "who are my neighbors?" SVM asks "where's the boundary, and which side am I on?" Both can reach the same answer for the same house — but they fail differently. KNN fails badly if your 5 nearest neighbors happen to be unrepresentative outliers. SVM fails badly if the true boundary isn't the kind of shape its kernel can represent.



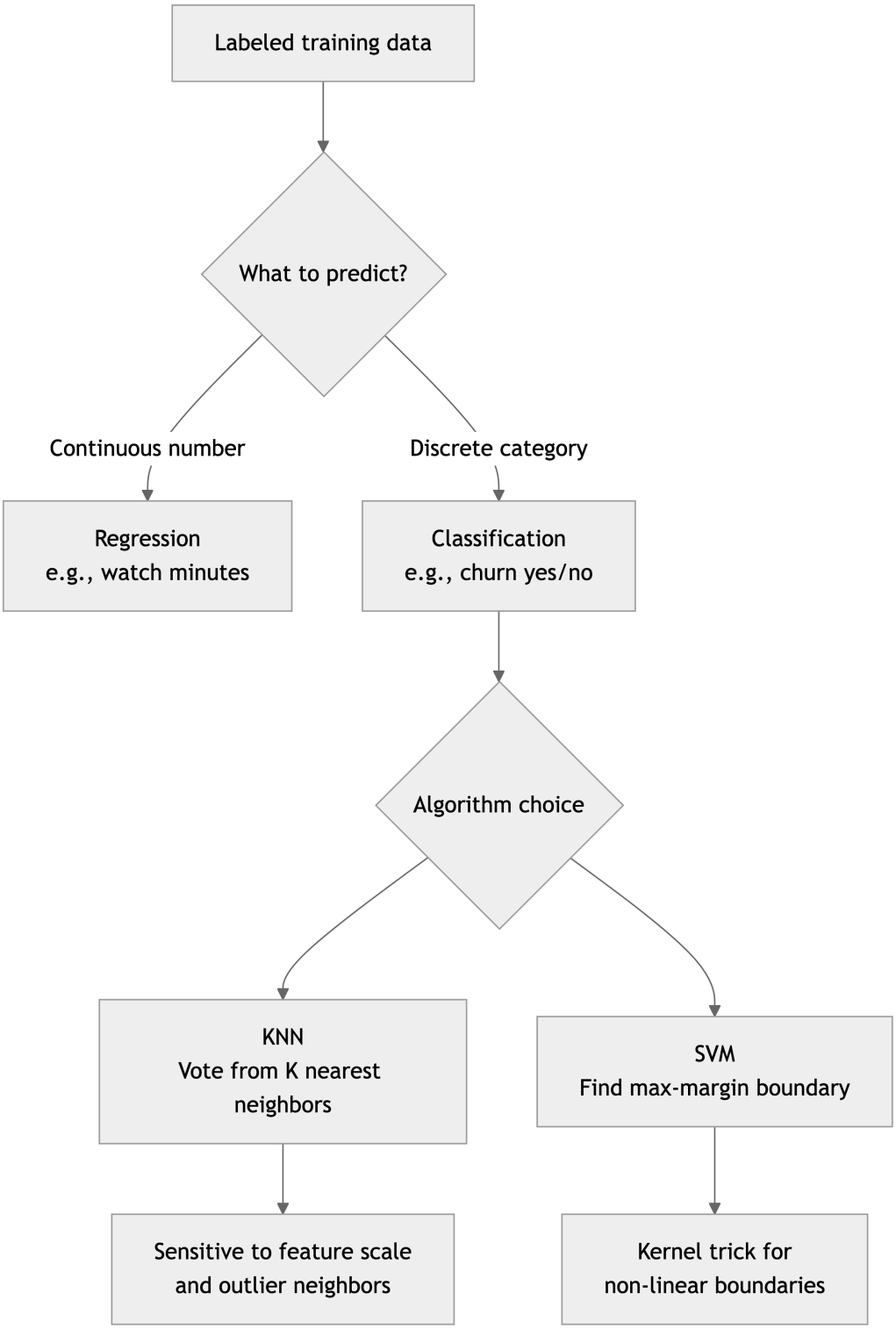


Figure 1: The supervised-learning fork: regression or classification, then the algorithm trade-offs

---

2. Theory & Mechanics

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

2.1 Regression: predicting a continuous number

Let's start with the simplest regression model, **linear regression**, on a tiny toy dataset: house size (in square meters) versus price (in thousands of dollars).

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Toy regression: predicting house price (in thousands) from size (sq
↪  meters)
sizes = np.array([50, 65, 70, 85, 100, 110, 120, 140]).reshape(-1, 1)
prices = np.array([150, 190, 200, 240, 280, 300, 330, 380])

reg = LinearRegression()
reg.fit(sizes, prices)

print("Slope (price per sq meter):", round(reg.coef_[0], 2))
print("Intercept:", round(reg.intercept_, 2))

new_size = np.array([[90]])
predicted_price = reg.predict(new_size)
print("Predicted price for 90 sq m:", round(predicted_price[0], 2))
```

Output:

```
Slope (price per sq meter): 2.55
Intercept: 22.88
Predicted price for 90 sq m: 252.38
```

A few mechanics to notice. First, `sizes.reshape(-1, 1)` — scikit-learn's `fit()` expects a 2D array of shape `(n_samples, n_features)`, even when there's only one feature. The `-1` tells NumPy "figure out this dimension automatically based on the data," so an 8-element 1D array becomes an 8×1 2D array. Forgetting this `.reshape()` is one of the most common first errors with scikit-learn — it raises a `ValueError` complaining about array dimensions.

Second, the model learned two numbers: a **slope** (2.55 — for every additional square meter, price goes up by about $2,550) and an **intercept** (22.88 — the baseline

price when size is 0, which is not meaningful here on its own, but is mathematically necessary to fit the line). Together, `price = 2.55 * size + 22.88` is the entire model. For a 90 sq m house, that's `2.55 * 90 + 22.88 = 252.38` — exactly what `.predict()` returned. This is the essence of regression: the model output is a number you can sanity-check by hand, and "how far off" a prediction is (the **error** or **residual**) is itself a number you can measure and average across many predictions. Chapter 072 covers the metrics built on this idea — MAE, MSE, RMSE.

2.2 Classification: predicting a category

Now the same shape of problem, but for a category instead of a number. Toy dataset: classify a piece of fruit as an apple (0) or an orange (1) based on its weight (grams) and texture (0 = smooth, 1 = bumpy).

```
import numpy as np
from sklearn.linear_model import LogisticRegression

X = np.array([
    [150, 0],
    [170, 0],
    [140, 0],
    [130, 1],
    [180, 1],
    [200, 1],
    [120, 0],
    [190, 1],
])
y = np.array([0, 0, 0, 1, 1, 1, 0, 1])  # 0=apple, 1=orange

clf = LogisticRegression()
clf.fit(X, y)

new_fruit = np.array([[160, 1]])
prediction = clf.predict(new_fruit)
probability = clf.predict_proba(new_fruit)

print("Prediction (0=apple, 1=orange):", prediction[0])
print("Probability [apple, orange]:", np.round(probability[0], 3))
```

Output:

```
Prediction (0=apple, 1=orange): 1
Probability [apple, orange]: [0.356 0.644]
```

Despite its name, `LogisticRegression` **is a classification algorithm**, not a regression algorithm — the "regression" in the name refers to how it works internally (it

fits a linear equation, then squashes the result through a *logistic function* to get a probability between 0 and 1), not to what it predicts. This naming quirk trips up almost everyone the first time. Notice two things about the output: `.predict()` returns a hard label (`1`, meaning "orange"), but `.predict_proba()` returns a probability for *each* class — `[0.356, 0.644]` means "35.6% confident this is an apple, 64.4% confident this is an orange." The predicted label is just whichever probability is higher. That second number — the probability — is often more useful in production than the label alone, because it lets you set your *own* threshold (Section 4 returns to this) rather than accepting scikit-learn's default of 0.5.

2.3 KNN: classification by neighbors

Now to KNN itself. We'll generate a synthetic 2-feature dataset using `make_classification` — a scikit-learn utility that creates a labeled dataset with controllable properties, useful for exactly this kind of teaching example (and used again in Chapters 078–079 per the bible's deviation notes).

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=200, n_features=2, n_informative=2, n_redundant=0,
    n_clusters_per_class=1, random_state=42
)

X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  test_size=0.25, random_state=42)

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

accuracy = knn.score(X_test, y_test)
print("KNN (k=5) test accuracy:", round(accuracy, 3))

for k in [1, 3, 5, 9, 15]:
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    acc = model.score(X_test, y_test)
    print(f"  k={k}: accuracy={acc:.3f}")
```

Output:

```
KNN (k=5) test accuracy: 0.9
```

```
  k=1: accuracy=0.820
  k=3: accuracy=0.940
  k=5: accuracy=0.900
  k=9: accuracy=0.880
  k=15: accuracy=0.860
```

`random_state=42` makes `make_classification` and `train_test_split` **reproducible** — the same seed always produces the same synthetic dataset and the same train/test split, which is why these exact numbers will reproduce on any machine. (Chapter 071 covers train/test splitting properly; for now, just note that `.fit()` only sees `X_train/y_train`, and `.score()` evaluates on the held-out `X_test/y_test` that the model never saw during training.)

The `k` sweep shows KNN's central trade-off. **k=1** means "copy whatever your single nearest neighbor's label is" — this is *maximally sensitive* to noise; if that one neighbor happens to be mislabeled or an outlier, the prediction is wrong, with nothing to outvote it. Here k=1 gives the *worst* accuracy (0.820). **k=3** gives the best result on this dataset (0.940). As k keeps growing (9, 15), accuracy drifts back down — with enough neighbors, the "vote" starts including points that aren't really representative of the local neighborhood anymore, and the boundary gets smoother but less precise. There's no universally correct k; it's a hyperparameter to tune (Chapter 073 covers this systematically), and the right value depends on the dataset's size, dimensionality, and noise level. A common starting heuristic is $k = \sqrt{n}$, but always verify against held-out data rather than trusting the heuristic blindly.

2.4 SVM and the scaling requirement

Same synthetic dataset, now with SVM — and a deliberate demonstration of why distance-based algorithms need **feature scaling**.

```
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler

X, y = make_classification(
    n_samples=200, n_features=2, n_informative=2, n_redundant=0,
    n_clusters_per_class=1, random_state=42
)
X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  test_size=0.25, random_state=42)
```

```
# SVMs are distance-based -- scale features first
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

for kernel in ["linear", "rbf"]:
    svm = SVC(kernel=kernel, random_state=42)
    svm.fit(X_train_scaled, y_train)
    acc = svm.score(X_test_scaled, y_test)
    n_support = svm.n_support_
    print(f"SVM (kernel={kernel}): accuracy={acc:.3f}, support vectors
     ↪  per class={n_support}")
```

Output:

```
SVM (kernel=linear): accuracy=0.880, support vectors per class=[33 33]
SVM (kernel=rbf): accuracy=0.920, support vectors per class=[36 34]
```

Two new mechanics here. First, `StandardScaler`: it transforms each feature to have mean 0 and standard deviation 1, by subtracting the mean and dividing by the standard deviation. Critically, `.fit_transform()` is called on the *training* data (it learns the mean and standard deviation from training data only), and `.transform()` (not `.fit_transform()`) is called on the *test* data — reusing the training set's mean and standard deviation. If you instead called `.fit_transform()` on the test set too, you'd be leaking information about the test set's distribution into preprocessing — a subtle form of the **data leakage** problem that Chapter 071 covers in depth. The rule: **fit on train, transform on both**.

Second, the kernel comparison: the `"rbf"` (radial basis function) kernel slightly outperforms `"linear"` here (0.920 vs 0.880) — meaning this particular synthetic dataset has a boundary that's *somewhat* curved, and the RBF kernel can represent that curve while a straight line can't quite. The `n_support_` array tells you how many training points from each class became "support vectors" — the points that actually define the boundary. Notice this is a *meaningful fraction* of the 150 training points (33–36 out of roughly 75 per class) — for SVM, prediction speed depends on the *number of support vectors*, not the size of the original training set, which is part of why SVM can struggle to scale to very large datasets: the number of support vectors tends to grow with the dataset.

---

3. CinemaStream in Practice

Mei Tan has a whiteboard marker in hand and a half-erased diagram of the `users`, `subscriptions`, and `watch_events` tables behind her. "Right," she says, "Module 8 built tools that *answer questions about* CinemaStream's data. Starting today, we build something that *predicts* something about it. Priya's been asking for a churn model since Chapter 042 — 'why is Premium churn rising in Indonesia specifically?' We're not answering that whole question today. Today we build the simplest version that could possibly work, and we'll spend the rest of Part 7 making it better and getting it into production."

"What's the cost of being wrong here?" she asks — her usual opener. "If we predict someone *won't* churn and they do, we miss a chance to retain them. If we predict someone *will* churn and they don't, we maybe send them an unnecessary discount offer. Neither is catastrophic on its own, which is exactly why churn prediction is a good first ML project — the stakes are real but forgiving while we're learning the workflow."

She sketches the feature list on the whiteboard, pulling from `users`, `subscriptions`, `watch_events`, and `support_tickets`: `plan` (Free/Basic/Premium), `country`, `watch_minutes_avg`, `days_since_last_watch`, `tenure_months`, `support_tickets_count`. "These are the features Chapter 069 will properly engineer from the real tables. For today, we generate a synthetic dataset shaped like what those joins would produce — same columns, same realistic churn rates per country and plan, fixed random seed so it's reproducible."

```
import numpy as np
import pandas as pd

RANDOM_SEED = 42
rng = np.random.default_rng(RANDOM_SEED)
N = 300

countries = rng.choice(
    ["VN", "PH", "ID", "MY", "TH", "SG", "IN"],
    size=N,
    p=[0.20, 0.18, 0.18, 0.12, 0.12, 0.10, 0.10],
)
plans = rng.choice(["Free", "Basic", "Premium"], size=N, p=[0.35,
 ↪  0.40, 0.25])

# Canonical churn rates (overall / Premium) from the team's "Ask
 ↪  Anything" work,
# Chapters 064-066: VN 8.2%/3.1%, PH 6.5%/5.9%, ID 5.0%/2.0%.
```

```
# Other countries use an invented baseline of 5.5% overall / 2.5%
 ↪  Premium.
OVERALL_CHURN = {"VN": 0.082, "PH": 0.065, "ID": 0.050, "MY": 0.055,
 ↪  "TH": 0.055, "SG": 0.055, "IN": 0.055}
PREMIUM_CHURN = {"VN": 0.031, "PH": 0.059, "ID": 0.020, "MY": 0.025,
 ↪  "TH": 0.025, "SG": 0.025, "IN": 0.025}

def base_churn_prob(country, plan):
    if plan == "Premium":
        return PREMIUM_CHURN[country]
    return OVERALL_CHURN[country]

tenure_months = rng.integers(1, 37, size=N)
watch_minutes_avg = np.clip(rng.normal(loc=85, scale=30, size=N), 5,
 ↪  None)
days_since_last_watch = rng.integers(0, 60, size=N)
support_tickets_count = rng.poisson(0.6, size=N)

rows = []
for i in range(N):
    base_p = base_churn_prob(countries[i], plans[i])
    # Behavioral multiplier: keeps each country/plan group centered
     ↪  near its
    # canonical base rate while giving the model real per-row signal.
    multiplier = 1.0
    if days_since_last_watch[i] > 21:
        multiplier *= 1.8
    if watch_minutes_avg[i] < 40:
        multiplier *= 1.6
    if support_tickets_count[i] >= 2:
        multiplier *= 1.5
    if tenure_months[i] < 3:
        multiplier *= 1.3
    if tenure_months[i] > 24:
        multiplier *= 0.6
    adj_p = np.clip(base_p * multiplier, 0.01, 0.95)
    churned = rng.random() < adj_p
    rows.append({
        "user_id": 1000 + i,
        "country": countries[i],
        "plan": plans[i],
        "watch_minutes_avg": round(float(watch_minutes_avg[i]), 1),
        "days_since_last_watch": int(days_since_last_watch[i]),
        "tenure_months": int(tenure_months[i]),
        "support_tickets_count": int(support_tickets_count[i]),
        "churned": int(churned),
    })

churn_df = pd.DataFrame(rows)
```

```
print(churn_df.head())
print("\nShape:", churn_df.shape)
print("\nOverall churn rate:", round(churn_df["churned"].mean(), 3))
print("\nChurn rate by country:")
print(churn_df.groupby("country")["churned"].mean().round(3))
```

Output:

```
   user_id country     plan  ...  tenure_months  support_tickets_count
    ↪  churned
0     1000      TH  Premium  ...             32                      0
 ↪  0
1     1001      ID     Free  ...             18                      1
 ↪  0
2     1002      SG    Basic  ...             36                      0
 ↪  0
3     1003      TH    Basic  ...             28                      0
 ↪  0
4     1004      VN  Premium  ...             24                      2
 ↪  0

[5 rows x 8 columns]

Shape: (300, 8)

Overall churn rate: 0.07

Churn rate by country:
country
ID    0.058
IN    0.154
MY    0.029
PH    0.085
SG    0.032
TH    0.108
VN    0.056
Name: churned, dtype: float64
```

"Notice the overall churn rate landed at 7%," Mei says. "That's realistic — and it's also the first warning sign for this whole project. Hold that thought; Carlos is going to ask about it in a minute." She adds the model code:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Encode categorical features (country, plan) as one-hot columns
```

```
features_df = pd.get_dummies(
    churn_df.drop(columns=["user_id", "churned"]),
    columns=["country", "plan"],
    drop_first=True,
)
X = features_df.values
y = churn_df["churned"].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=RANDOM_SEED, stratify=y
)
print("Train churn rate:", round(y_train.mean(), 3), "| Test churn
 ↪  rate:", round(y_test.mean(), 3))

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

log_reg = LogisticRegression(max_iter=1000, random_state=RANDOM_SEED)
log_reg.fit(X_train_scaled, y_train)
log_reg_acc = log_reg.score(X_test_scaled, y_test)
print("\nLogistic Regression test accuracy:", round(log_reg_acc, 3))

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_scaled, y_train)
knn_acc = knn.score(X_test_scaled, y_test)
print("KNN (k=5) test accuracy:", round(knn_acc, 3))

majority_class = int(round(y_train.mean()))
baseline_preds = np.full_like(y_test, majority_class)
baseline_acc = accuracy_score(y_test, baseline_preds)
print("Majority-class baseline accuracy:", round(baseline_acc, 3))
```

Output:

```
Train churn rate: 0.071 | Test churn rate: 0.067

Logistic Regression test accuracy: 0.933
KNN (k=5) test accuracy: 0.933
Majority-class baseline accuracy: 0.933
```

Mei doesn't move for a moment. "And there it is. All three numbers — logistic regression, KNN, and a model that does *nothing but always predict 'no churn'* — get exactly the same accuracy. 93.3%." She lets that sit. "This is the single most important lesson in this entire chapter, and it's not an algorithm. **Accuracy is a misleading metric when the classes are imbalanced.** Only 7% of our users churn. A model that learns absolutely nothing — that just always says 'this user will stay' — is *right* 93% of the time, by definition, because 93% of users *do* stay. Both our real

models matched that exactly. Did they learn anything at all, or did they just also learn to mostly say 'no churn'?"

Why logistic regression as the baseline, not KNN: for this chapter's baseline, we use **logistic regression**, for three reasons Mei gives the room. First, it's *interpretable* — the coefficients (one per feature) tell you the direction and rough size of each feature's effect, which matters when Priya or Dharani asks "why does the model think this user will churn?" KNN gives you a label and a neighbor list, not a reason. Second, logistic regression handles the mix of numeric features (`watch_minutes_avg`, `tenure_months`) and one-hot encoded categorical features (`country_VN`, `plan_Premium`) gracefully, without the curse-of-dimensionality concerns that affect distance-based KNN as feature counts grow. Third, it's the standard *first* model for binary classification in industry — "did logistic regression beat it?" is the natural question any more complex model (Chapter 068's trees, Chapter 102's XGBoost) will need to answer. KNN remains useful here as a *sanity-check comparison* — and the fact that both models tied the baseline is itself the finding worth reporting, not a failure to report around.

"So what do we do?" Rohan asks, joining late and already looking at his phone. "Can I get a 'will this user churn' number by EOD?"

"You can get a number," Mei says, "but right now that number is 'no, basically nobody, according to a model that's wrong in exactly the way that matters.' The model technically achieved 93.3% — and is *useless* for the actual business question, which is 'which of the 7% are at risk?' Accuracy doesn't distinguish between a model that's good at finding that 7% and a model that's ignoring them entirely. We need different metrics — precision, recall, the confusion matrix — to actually tell those apart. That's Chapter 072, and it's not optional; it's the chapter that turns 'the model ran' into 'the model is useful.'" She writes on the whiteboard: *Ch067: build it. Ch068: try better algorithms. Ch072: measure it honestly.* "Today's deliverable is the pipeline — data in, model out, accuracy reported honestly including its limitation. That's `cinemastream/ml/churn/train.py`, and it's the first file in a folder we'll be extending for the next ten chapters."

---

4. Pitfalls & Pro Tips

- **Accuracy on imbalanced data is close to meaningless on its own.** Section 3's punchline — a model tied with "always predict the majority class" — is not a synthetic-data quirk. Real churn, fraud, and defect-detection datasets

are almost always imbalanced (the "bad" outcome is rare by definition), and accuracy alone will flatter a model that's learned nothing useful. Always compute and report the majority-class baseline *alongside* your model's accuracy. If they're equal, your model hasn't demonstrated anything yet.

- **Forgetting `.reshape(-1, 1)` for single-feature data.** Section 2.1's `size_s.reshape(-1, 1)` converts a 1D array of 8 values into an 8×1 2D array, because scikit-learn's `.fit()` always expects `(n_samples, n_features)` — even when `n_features` is 1. The error message (`ValueError: Expected 2D array, got 1D array instead`) is common enough that it's worth memorizing the fix rather than looking it up every time.

- **Scaling leakage: fitting the scaler on the full dataset, or on the test set.** Section 2.4 fit `StandardScaler` on `X_train` only, then used that *same fitted scaler* (via `.transform()`, not `.fit_transform()`) on `X_test`. If you instead scale the entire dataset before splitting, information about the test set's mean and standard deviation leaks into the training process — a subtle violation of the "test set represents truly unseen data" assumption. Chapter 071 covers this family of leakage bugs in depth, including a real one the team finds in the churn pipeline.

- **KNN's distance calculation silently breaks with unscaled features.** If `watch_minutes_avg` ranges from 5 to 200 and `tenure_months` ranges from 1 to 36, the *raw* numeric distance between two points is dominated almost entirely by `watch_minutes_avg` — a 50-minute difference in watch time looks "bigger" to KNN than a 30-month difference in tenure, even if tenure is the more predictive feature. Always scale features before using KNN or SVM. Tree-based models (Chapter 068) don't have this problem, which is one reason they're often preferred for mixed-scale tabular data.

  FDE Standard: When a client asks for "a churn model" (or any classification model), the first question is never "which algorithm?" — it's **"what's the base rate, and what decision will this model's output drive?"** A 93% accurate model is worthless if 93% is also the base rate and the business decision (who gets a retention email, who gets flagged for a support callback) depends on correctly identifying the *minority* class. Establish the base rate and the downstream decision *before* writing any model code — it determines which metrics matter (Chapter 072) and sometimes whether classification is even the right framing versus, say, a ranked list of risk scores for a capacity-constrained retention team.

Enjoying this preview?

This was 24 of 3,008 pages — 2 chapters of 180.

Chapter 067 just hit the 0.933 wall.
Ten chapters fight it — and that's one module of thirteen.

Get the full book (EPUB + PDF bundle) at:
fde-handbook.adsit.work

180 chapters · 13 modules · 7 reading paths
A complete runnable portfolio repo — every concept applied to one realistic streaming company, CinemaStream. You are Employee #47.

The Forward Deployed Engineer's Handbook — Abhijeet Verma · Adsit Press
ISBN 978-81-688506-0-6 · books@adsit.work