Predictive Modeling for Marketing Decisions

From Explaining the Past to Scoring the Future

Dr. Jose Mendoza, Academic Director and Clinical Associate Professor
Version 1.0 · July 2026

Chapter Information

ABSTRACT

This chapter develops predictive modeling as a distinct analytic practice with its own obligations and failure modes. It builds the predictive frame — unit, target, features, and horizon — and the label as a manufactured measurement, with a snapshot date placing a wall in time between what a model may know and what it must predict. Leakage is defined as the canonical error and cataloged by route. Honest evaluation follows: train/test splits and the in-sample flattery they defeat; mean and last-value baselines operationalized as stated opponents; MAE and RMSE read in dollars beside capture at the program's capacity as the decision metric; overfitting and underfitting read from the error signature; and training-only cross-validation as the instrument that keeps the test set sealed. The labs build a spend-prediction pipeline, compare candidates on a frozen leaderboard, manufacture and catch a leaking model, and grade a vendor's near-perfect accuracy claim.

KEYWORDS

predictive modeling; leakage; train/test split; baselines; overfitting; cross-validation; MAE; RMSE; expected value; model drift

VERSION AND DATE

Version 1.0 · July 2026 · Language: English (United States)

SUGGESTED CITATION

Mendoza, J. (2026). Predictive modeling for marketing decisions. In *Applied business analytics for marketing decision-making: Business analytics and data visualization* (Chapter 8, Version 1.0) [Open educational resource]. CC BY 4.0.

of GitHub, Inc. Product names are used for identification only and do not imply endorsement. StyleCraft Collective is a fictional company created for instruction.

COMPANION REPOSITORY
Datasets, notebooks, and figure sources for this chapter: https://github.com/jrmst102/businessanalytics

Chapter Learning Objectives

By the end of this chapter, students should be able to:

1. Distinguish predictive from explanatory modeling as goals with different obligations, citing the Section 7.10 distinction, and state what changes when a model's outputs become the product.
2. Specify a complete predictive frame — unit of prediction, target, feature set, and horizon — in writing before fitting, as the predictive extension of the analytic specification of Section 2.5, and declare its decision metric and baselines alongside it.
3. Define a prediction label precisely, including the snapshot date, feature window, outcome window, and eligibility rule, and explain why the label is a measurement decision rather than a given.
4. Define leakage, catalog its common routes in marketing data, and audit a feature list for information that crosses the snapshot line.
5. Construct an honest train/test evaluation, explain in-sample flattery, state what a random customer holdout does and does not validate, and apply the split hygiene rules that protect it.
6. Compute and interpret MAE and RMSE in dollars, compute top-decile spend capture as the decision metric of a ranked-list program, and choose the metric whose weighting matches the decision at stake.
7. Operationalize baselines — mean, majority-class, and last-value — as stated opponents, and express a model's value as its margin over the strongest honest baseline.
8. Diagnose overfitting and underfitting from the training-versus-validation error signature, and explain the bias–variance tradeoff at intuition level.
9. Run k-fold cross-validation on the training data to compare candidate models and select model complexity, and apply the comparison discipline of same folds, same metric, same baselines.
10. Translate predictions into decisions: build ranked targeting lists under a fixed budget, apply expected-value framing at concept level, distinguish predicted spend from spend caused by a treatment, and state the deployment realities of data drift, concept drift, performance decay, and retraining.
11. Verify an AI-built or vendor-supplied predictive model with the five-point audit — frame, leakage, split hygiene, baseline, and error in decision units — treating “too good” as a red flag, and document the audit per Appendix D.

Chapter 7 ended with a deliverable and a warning. The deliverable was the drivers analysis: spend and average order value modeled on the feature table, segment dummies read against the Urban Loyal Core, the designed interaction priced in dollars, and every coefficient guarded by a verb that stopped at the evidence. The warning came in the chapter's last paragraph: the drivers model was built to explain, and the moment a model's outputs — rather than its coefficients — become the product, the obligations change. This chapter is about that change. The question is no longer what moved spend in the window we observed; it is what spend will be in the window we have not yet seen, customer by customer, accurately enough to act on. Answering it requires machinery Chapter 7 never needed: a target defined in time rather than assumed, a dataset split so the model can be graded on customers it never met, baselines that convert Chapter 2's interpretive discipline into a working opponent, error metrics denominated in dollars and in the decision's own currency, and a new class of silent failure — leakage, the model that grades itself on information it should never have had — whose most reliable symptom is the one result every stakeholder wants to hear: a model that looks too good.

CONCEPT

What This Chapter Is Really About

Chapter 7 closed by separating two modeling goals: explanation, where the coefficients are the deliverable, and prediction, where the outputs are. The separation sounds administrative and is anything but, because the two goals fail differently. An explanatory model fails in public — a sign that contradicts known structure, a causal verb that overreaches — and Chapter 7 built the audit that catches it. A predictive model fails in private. It can be arithmetically flawless, statistically significant, beautifully fitted, and worthless, because everything that certified it happened on data it will never see again.

The predictive disciplines of this chapter — the frame declared before fitting, the wall in time between features and outcome, the test set touched once, the opponent named before the race — are all versions of a single idea: the model must be graded under the conditions of its actual job, predicting what it has not seen, against the alternative of not using it at all. And the chapter's characteristic red flag inverts every instinct a student brings to it: in predictive work, the result that looks too good almost never is, and the analyst's most valuable reflex is suspicion of their own success.

Source: Course concept developed for this guide, informed by Shmueli (2010) and Kaufman et al. (2012).

8.1 Marketing Decision Context: The Scored List and the Vendor's Promise

Chapter 7's drivers deliverable survived the budget-lock meeting: the install campaign was funded as a test rather than a certainty, the discount ladder survived on the within-segment exhibit, and the fall retention program's incremental budget was allocated across the four journeys the treatment map defined. Now the program is being built, and its most expensive component has a capacity constraint. The top journey tier — internally, the Backstage Preview

— pairs early access to the holiday drops with a printed lookbook and a small gifting touch. Merchandising and print work out to roughly twenty dollars per treated customer, and the budget covers eight hundred customers: the top decile of a base of roughly eight thousand. Eight hundred slots, eight thousand customers. Someone must decide who gets one, and the CRM manager has asked the obvious question in its modern form: can we predict, customer by customer, who will spend the most over the next six months — and use that prediction to fill the list?

Three answers are already in the building, and each conceals a problem this chapter exists to expose. The first is the incumbent rule, defended by the CRM team's operations lead: “Rank customers by what they spent in the last six months and take the top eight hundred. Past spend is the best predictor of future spend. We do not need a model; we need a spreadsheet sorted in descending order.” The second arrives from outside, in a vendor deck. StyleCraft's customer-data-platform vendor has been piloting a predictive-spend module, and its pitch slide is the kind that ends meetings: “Our AI model predicts each customer's next-six-month spend with 96 percent accuracy ($R^2 = 0.96$).” The annual license costs more than the entire Backstage print budget. The third comes from inside the analytics team, and it is the most tempting because it is nearly right: “We already built a spend model in the drivers analysis. Its signs survived the four-point audit — score everyone with it and take the top eight hundred.”

Each answer has a real virtue and a disqualifying flaw, and stating the flaws is the fastest tour of this chapter's territory. The incumbent rule is cheap, transparent, and — this chapter will insist — the legitimate baseline every model must beat before it earns a budget line. But Chapter 5's certified findings already show why it misfires: the base is lumpy. Suburban Occasion customers concentrate a season's spending into one or two large baskets, so a customer with zero recent spend may be three weeks from a four-hundred-dollar occasionwear order, and a customer who just placed one may be done for the year. A backward-looking sort systematically gifts the customers whose spending just happened and misses the customers whose spending is about to. The vendor claim fails differently: not on its arithmetic, which no one at StyleCraft can inspect, but on its vocabulary and its plausibility. Chapter 7 established that individual human spending is noisy and that fit statistics near 1 on individual behavior are a symptom rather than an achievement (Section 7.7). The in-house answer fails most instructively of all. The drivers model was built, in Section 7.10's terms, to explain: fitted on the full window, graded on the data it was fitted to, its variables chosen to make coefficients defensible rather than predictions accurate. Nothing about surviving the four-point audit certifies that its predictions hold on customers and months it has never seen.

One boundary belongs in the brief rather than in a footnote three sections later, because it constrains what the whole project can claim. A model of future spend can prioritize the customers expected to be most valuable. It cannot identify the customers whose spending will increase *because* they receive the Backstage treatment. Those are different quantities, and only the second is what a marketer means by the return on a gift. The program is therefore commissioned as a recognition and prioritization program — treat the customers we expect to

matter most — with the incremental question named as unanswered and Chapter 11's experimental machinery named as the instrument that would answer it.

So the VP of Marketing has commissioned the analyst — again, you — with a deliverable and a deadline. The deliverable: a scored list of eight hundred customers, produced by a method chosen honestly among the three candidates or a better fourth, with the evidence for the choice stated in a form the CFO can interrogate. The deadline: the scoring method must be locked in thirty days, when the holiday-drop calendar goes to print. Sections 8.2 through 8.4 build the frame, the label, and the leakage discipline. Sections 8.5 through 8.9 build honest evaluation: splits, error metrics, baselines, overfitting, and the comparison rules. Section 8.10 carries the score to the decision and into deployment. Section 8.11 turns to the AI assistant, which will build any pipeline fluently — including, without complaint, a leaky one. The labs in Section 8.12 build the honest pipeline, compare candidates on a frozen leaderboard, manufacture the leaky one on purpose, and catch it. Sections 8.13 through 8.15 rehearse the meeting.

8.1.1 Opening Case Questions

Keep these questions in mind while reading, and return to them after completing the labs.

1. The incumbent rule — rank by trailing six-month spend — is both a candidate answer and, in this chapter's terms, a baseline. What is the difference between rejecting it as an answer and using it as a baseline, and why does the second remain essential even if the first happens?
2. The vendor's “96 percent accuracy” claim contains two separate problems: one of vocabulary and one of plausibility. Name both, and state which sections of this guide equip you to press on each.
3. The in-house drivers model survived Chapter 7's four-point audit. List two specific reasons that audit does not certify the model for the scoring job, drawing on the distinction in Section 7.10.
4. The brief separates predicting who will spend most from identifying whose spending the gift would increase. State the practical consequence of that boundary for how the program's success should be measured next quarter.
5. The Backstage list will treat eight hundred customers and pass over seventy-two hundred. Describe one way each kind of error — gifting a customer who was going to spend little, missing a customer who would have spent much — costs the program, and recall which earlier section taught you to expect the two costs to differ (the answer returns in Section 8.10 and again in Chapter 9).

8.2 From Explanation to Prediction: The Predictive Frame

Section 7.10 drew the distinction this chapter now inhabits: explanation and prediction are different modeling goals, served differently by the same equation, with different success measures and different characteristic failures. One sentence of reminder suffices — an explanatory model is graded on whether its coefficients are honest structure; a predictive model

is graded on whether its outputs are accurate for cases it has not seen — and the rest of this section develops what the second goal demands before any model is fitted at all. The demand is a specification, and it is the direct descendant of the analytic specification of Section 2.5: just as no analysis in this guide begins without a written statement of the decision, the alternatives, and the deliverable, no predictive model begins without a written statement of four commitments this guide calls the predictive frame.

DEFINITION

Predictive Modeling and the Predictive Frame

Predictive modeling is the construction of a model whose purpose is to output accurate estimates of an unknown or future quantity for individual cases, evaluated on cases not used to build it. Its specification — the predictive frame — consists of four declared commitments: the unit of prediction (what entity receives a prediction), the target (the quantity predicted, precisely defined), the features (the information the model may use, fixed as of a stated moment), and the horizon (how far beyond that moment the target lies).

Source: Adapted from Provost and Fawcett (2013) and Kuhn and Johnson (2013).

Each commitment is a decision with consequences, and each connects to machinery this guide has already built. The unit of prediction is the grain question of Section 3.3 pointed forward: StyleCraft's scoring problem lives at the customer grain — one prediction per customer — but the same business could ask for predictions at the store grain (next quarter's revenue per store), the campaign grain (expected response per send), or the day grain (Chapter 10's subject). Declaring the unit settles what a row means in every table the pipeline touches, and undeclared units produce the same silent damage here that undeclared grain produced in Chapter 4's joins. The target is the subject of Section 8.3, because it deserves its own section: “future spend” is not a column that exists but a measurement that must be constructed, and every construction choice changes what the model learns to do. The features are the model's permitted information, and the discipline they require — nothing the model would not know at prediction time — is the subject of Section 8.4. The word itself was defined in Section 3.5 among the variable roles; what is new here is not the role but the restriction. The horizon is the forward distance of the claim: the Backstage decision needs six months, because the program will be judged on the fall-and-holiday arc, while a win-back trigger might need only thirty days. The horizon disciplines everything downstream — a model with a six-month horizon cannot be validated in two weeks, and Section 8.10 will show that the horizon also sets the clock on how often the model must be retrained.

Table 8.1 states the frame for the Backstage decision in full, and its final column carries the point of the exercise: every row is a choice that could have been made differently, and each alternative would produce a different model answering a different question. In other words, “predict customer spend” is not yet a modeling problem — it is a family of modeling problems, and the frame is the act of choosing one member and committing to it in writing. The

commitment is what makes verification possible later: a model can only be audited against a frame that exists.

Table 8.1

The predictive frame for the Backstage scoring decision

Commitment	**Declaration for this decision**	**What a different choice would change**
Unit of prediction	One prediction per customer (customer grain, per Section 3.3)	Store- or day-grain models answer planning questions, not list-building ones
Target	Total spend, in dollars, in the six months after the snapshot date (defined fully in Section 8.3)	A yes/no target (“will spend at all”) changes the method family — Chapter 9's subject
Features	The customer feature table, computed only from transactions inside the declared feature window	Features from after the snapshot are leakage (Section 8.4), not information
Horizon	Six months — the fall-and-holiday arc the program will be judged on	A 30-day horizon serves triggers, not seasonal list-building; longer horizons decay differently (Section 8.10)
Decision metric	Spend capture at the program's capacity share — 800 slots divided by the eligible population — against the incumbent list (Section 8.6)	A larger budget or a wider eligible population moves the cutoff and therefore the metric
Calibration metrics	MAE leading, RMSE reported beside it, both in dollars	A decision with convex costs would promote RMSE to the lead (Section 8.6)
Baselines	Mean prediction and the trailing-six-month ranking, on the same folds and the same test set	A baseline chosen after the results can be chosen to lose (Section 8.7)

One more inheritance completes the frame's paperwork, and Table 8.1's last three rows record it. Section 2.5's specification discipline required that the deliverable be named before the analysis begins; the predictive version has a sharper edge, because the frame must also name, in advance, the metric the model will be graded on and the baselines it must beat. Declaring them before fitting is not ceremony. It is the predictive analog of predict-then-verify (Section 1.7): a model builder who chooses the success metric after seeing the results has the same conflict of interest as an analyst who writes the hypothesis after seeing the data, and Section 8.9's comparison discipline exists because the temptation is real, routine, and mostly unconscious.

8.3 Defining the Target: Labels, Snapshots, and Windows

The frame's second commitment deserves its own section because it is the one students most reliably treat as given when it is in fact constructed. The drivers analysis of Chapter 7 modeled monetary — total spend across the observed window — a column that existed in the feature table before any model was imagined. The scoring decision has no such column. “What this customer will spend in the next six months” exists nowhere in StyleCraft's data today; for the model to learn it, the analyst must manufacture historical examples of it, and the manufacturing is a measurement act with all of Chapter 3's obligations attached. The manufactured quantity is called the label: the recorded value of the target for a case whose outcome is already known, from which the model learns the relationship between features and future.

> **DEFINITION**
>
> **Label (Target Variable)**
>
> A label is the constructed, recorded value of the target for a historical case — the answer the model trains on. Constructing labels requires choosing a snapshot date that splits time into a feature window (the past the model may know) and an outcome window (the future the label summarizes), together with eligibility rules for which cases receive labels at all. The label is a measurement decision: its windows, its filters, and its eligibility rules are choices that change what the model learns, and they are declared, not discovered.
>
> Source: Adapted from Provost and Fawcett (2013) and Kuhn and Johnson (2013).

The construction is easiest to see drawn on StyleCraft's own calendar. The certified transaction history runs twenty-four months, from July 1, 2024 through June 30, 2026 — the window every lab since Chapter 4 has used. To manufacture labels, the analyst plants a snapshot date inside that history: December 31, 2025. The feature window is the eighteen months from July 1, 2024 through December 31, 2025, and the customer feature table is rebuilt from it, exactly per Table 6.5's derivation rules but with the analysis date moved back to the snapshot. The outcome window is the six months from January 1 through June 30, 2026, and each customer's label, call it spend_next6m, is their total certified spend inside it: numerator, the sum of line_revenue on their transactions in the outcome window; denominator, none — it is a total, not a rate; window, January 1 through June 30, 2026; filters, none beyond the eligibility rule below. The four-element discipline of Section 3.5 applies to labels exactly as it applied to metrics, and writing the label in that form is not an homage — it is what makes two analysts' pipelines produce the same target.

Notice that both windows have two ends, and that both ends are load-bearing. A feature window declared as eighteen months but implemented as “everything before the snapshot” silently includes whatever history the file happens to carry, so two analysts working from the same declaration produce different recency and different monetary values. An outcome window declared as six months but implemented as “everything after the snapshot” silently lengthens the horizon for any customer whose file extends further, so the label stops meaning what the frame

says it means. Lab 8.1 therefore names four dates rather than one and asserts that the minimum and maximum transaction dates on each side fall inside their declared window. This is the four-element metric discipline turned into two assertions.

Three consequences of the construction deserve a paragraph each, because each becomes a lab step. First, the snapshot creates a wall in time, and the wall is what the whole chapter protects. On the feature side, the model may know everything StyleCraft knew on December 31, 2025: eighteen months of recency, frequency, spend, discount behavior, channel mix, and engagement. On the outcome side, the model may know nothing — not one transaction, not one field updated after the snapshot — because at the moment the real scoring decision is made, the future is exactly that unavailable. Every feature must therefore be re-derived as of the snapshot: recency_days recomputed against December 31, tenure_days truncated at the wall, monetary summing only feature-window transactions. Reusing the June 2026 feature table, the one Chapters 6 and 7 worked with, would hand the model six months of the very future it is being asked to predict — the error Section 8.4 names.

Second, eligibility is part of the label, and this chapter states its rule in the form the code can enforce. A customer receives a labeled row when two conditions hold: their signup_date falls on or before the snapshot, and they placed at least one order inside the feature window. The first condition is obvious — a customer acquired in March 2026 has no feature-window history and would enter the training data as a row of zeros wearing a label they never had a chance to earn. The second is the one drafts forget, and it is not a convenience: recency, average order value, discount share, and store share are all undefined for a customer with no feature-window purchases, because each of them divides by a quantity that is zero. This is the same population rule Chapter 6 declared for the clustering base (Section 6.6), inherited here deliberately rather than reinvented, and it carries the same obligation: the excluded customers are counted and named, not silently dropped.

Third, a label of zero is a label, not a gap. Customers eligible at the snapshot who bought nothing in the outcome window have spend_next6m equal to zero — and they are the single most important group in the training data, because they are the customers the incumbent sort-by-past-spend rule cannot distinguish from the lumpy occasion shoppers about to place a large order. Dropping zero-label rows, a mistake AI assistants make with some regularity when asked to “clean” a training table, deletes precisely the contrast the model exists to learn. The missing-data disciplines of Chapter 4 apply, but the prior question is definitional: here, absence of transactions is the outcome, not a defect in recording it.

CONCEPT

The Modeling Population Is Not the Deployment Population

The eligibility rule defines who the model learns from. It also defines, by inheritance, who the model can honestly score — and the two populations drift apart the moment the model is deployed. StyleCraft's Backstage model learns from customers who had purchased by December 31, 2025. By the June scoring run, the base contains customers acquired since the snapshot, customers who signed up long ago and bought for the first time in the spring, and customers whose only history sits outside any window the model has seen.

The instinct to say the model "cannot" score them is wrong, and the correction matters. Technically it can: the same features can be constructed for anyone with a purchase history, and the model will return a number for every row it is handed. The real constraint is representation. A customer whose feature combination barely appears in the training data receives a prediction the evaluation never tested, and a customer with no feature-window history at all receives one built from undefined ratios. Those customers are an activation problem, not a scoring problem, and the deliverable says so in writing: the model prioritizes established customers, a separate rule handles the newly acquired, and the boundary between them is declared rather than discovered in production.

Source: Course concept developed for this guide, informed by Kuhn and Johnson (2013).

8.4 Leakage: The Canonical Error

Chapter 7 closed its multicollinearity section with a designed trap: regress monetary on aov and frequency, watch R^2 leap toward 1, and recognize that the model had been handed the answer — monetary is, by Table 6.5's own derivation rules, their product. Chapter 7 promised that Chapter 8 would name the general disease. Here is the name. Leakage is the use, in training or evaluating a predictive model, of information that would not be available at the moment the model makes its real predictions — information from the future, from the target's own construction, or from the evaluation data itself. It is the canonical error of predictive modeling: the most common serious failure, the hardest to see in a finished pipeline, and the one whose symptom is a model that performs implausibly well right up until it is deployed, at which point the borrowed information is no longer there to borrow and performance collapses to what the legitimate features always supported (Kaufman et al., 2012).

DEFINITION

Leakage

Leakage is the presence, in a model's training features or evaluation procedure, of information that will not legitimately be available when the model is used for its real predictions — most often information from after the prediction moment, information arithmetically derived from the target, or information shared between training and test data. A leaky model's measured performance reflects the leaked information rather than genuine predictive relationships, and its deployed performance falls to the level the legitimate features support.

Source: Adapted from Kaufman et al. (2012).

In other words: a leaky model is not a bad model that can be tuned into a good one. It is an honest measurement of the wrong thing — of how well the future predicts itself — and no amount of refitting repairs it, because the defect is in the data the model was allowed to see. That is why leakage is caught by audit rather than by statistics: no diagnostic printed by the fitting software distinguishes leaked performance from earned performance, and the only reliable detector is an analyst tracing where each feature came from and when it became knowable. The trace has a natural checklist, because leakage in marketing data arrives by a small number of well-worn routes. Table 8.2 catalogs the five this guide's students will actually meet, each with its StyleCraft instance and its repair; the first two account for most real cases.

Table 8.2

Common leakage routes in marketing data

Route	Mechanism	StyleCraft instance	Repair
Temporal leakage	Features computed from data after the snapshot date	Building the training table from the June 2026 feature table when the label starts January 2026	Re-derive every feature as of the snapshot; audit each column's window
Target-derived (arithmetic) leakage	A feature that is an arithmetic relative of the label	Full-window monetary as a feature: it equals feature-window spend plus the label itself	Trace every feature's derivation rule (Table 6.5) against the label's
Outcome-window contamination	A feature that quietly summarizes part of the outcome window	Days since last purchase computed at the June 2026 analysis date — small values are only possible because of outcome-window orders	Recompute against the snapshot; range-check features for impossible values
Split contamination	A learned preprocessing or feature-construction step fitted before splitting	Standardizing features, or refitting Chapter 6's segmentation, using values that include the test customers	Fit every learned step inside a pipeline, on training data only, then apply to validation and test

Route	Mechanism	StyleCraft instance	Repair
Population leakage	Eligibility or filtering rules that use outcome-window information	Training only on “active customers,” with activity defined over the full 24 months	Define eligibility using feature-window information only

Three of the table's rows repay a closer look, because their surface innocence is the lesson. Outcome-window contamination is temporal leakage's subtle cousin: no one typed “use the future” — someone reused a convenient existing table whose columns were computed at the wrong date, and every derived value silently absorbed six months of the answer. The tell is often a range check, which is why the audit in Section 8.11 includes one: at a December 31, 2025 snapshot, no eligible customer can have a recency of fewer than 181 days when recency is computed correctly, so a table full of smaller values has been computed at a later date, and a single impossible value convicts the whole column.

Split contamination is smaller in effect than the first two rows and larger in principle, and it needs its boundary drawn precisely, because the boundary is where students most often over-apply the rule. What contaminates a split is a step that *learns something from the data* and is fitted before the split: a standardizer that estimates means and spreads, an imputer that estimates fill values, a feature selector that ranks columns by their relationship to the target, a clustering model that estimates centroids, a target encoder that averages the label within categories. Each of those carries a whisper of the test customers' distribution into training, and the whispers compound across steps. A transformation that learns nothing — taking a logarithm, multiplying two columns, expanding a predictor into fixed polynomial powers — does not contaminate anything, because applying it before or after the split produces identical values for every row. The rule to memorize is therefore about estimation rather than about ordering: anything estimated from data — a mean, a scale, an imputation value, a centroid, a category encoding — is estimated from the training data only, which in practice means it lives inside a pipeline that is refitted in every fold (scikit-learn developers, 2026a).

Population leakage is the one that hides inside a sentence rather than inside a column. A filter is a modeling decision, per Section 3.5, and a filter that consults the outcome window chooses the training population using the answer. “Customers who were active during the study” sounds like housekeeping and is a prediction: it quietly removes the customers who went dark, which is the outcome the model is supposed to be able to anticipate. The repair is mechanical — every eligibility condition must be evaluable using only information available on or before the snapshot — and this chapter's eligibility rule was written in Section 8.3 to satisfy exactly that test.

What makes leakage the chapter's central discipline rather than one hazard among several is its relationship to the chapter's red flag. Every route in Table 8.2 improves measured performance; none improves real performance; therefore unexplained excellence is evidence of a leak. The logic deserves a box, because it inverts the instinct every student brings to model building — and because it is the exact instrument the vendor claim of Section 8.1 requires.

CONCEPT

Too Good Is a Finding

In descriptive work, a strong result invites celebration and a write-up. In predictive work, a strong result invites an audit, and the stronger the result, the more urgent the audit. The reasoning is blunt in practice: genuine predictability of individual human behavior is bounded — Chapter 7 established that noisy individual outcomes cap what any honest model can explain — while leakage routes are numerous, quiet, and always performance-enhancing. A reported R^2 of 0.96 on individual customer spend is therefore far more likely to be a leak than a breakthrough, and the response it deserves is not admiration but the five-point audit of Section 8.11, starting with the feature trace.

The discipline extends beyond your own models. Applied to the vendor's deck, “too good” is not an accusation but a question the vendor must be able to answer: show us the frame, the snapshot, and the feature windows. A vendor who cannot is quoting in-sample fit or leaked features; either way, the number does not mean what the slide implies. This box is the chapter's verification theme in one sentence: in predictive modeling, suspicion scales with success.

Source: Course concept developed for this guide, informed by Kaufman et al. (2012) and Domingos (2012).

8.5 The Out-of-Sample Standard: Train/Test Splits and In-Sample Flattery

Leakage corrupts what the model knows; this section addresses how the model is graded, and the two disciplines are halves of one standard. A predictive model's job is to be accurate on cases it has not seen. Its performance on the cases it was fitted to — in-sample performance — is therefore not merely a weaker version of the evidence; it is systematically misleading evidence, and misleading in a known direction. Least squares, by construction, chooses the coefficients that minimize error on the fitting data (Section 7.6). The fitted model is thus tailored to that data's every quirk: its whales, its noise, its accidental patterns. Grade it on the same data and the tailoring is graded as skill. This guide calls the phenomenon in-sample flattery, and its size is not constant — it grows with model flexibility, shrinks with sample size, and vanishes only in the one place it cannot be observed: the future. Every impressive statistic computed on training data should be read with this discount in mind, and the discount cannot be computed from the training data itself.

The repair is structural, not statistical: hold data back. Before any fitting, the labeled customers are split at random into a training set, which the model may learn from without restriction, and a test set — also called a holdout — which is sealed until evaluation and consulted exactly once, for the finalist. The split is performed at the unit of prediction — whole customers go to one side or the other, per the frame's grain declaration — with a fixed random seed so the split is reproducible, and with proportions this guide sets at 80/20: enough training data to fit stable models on eight thousand customers, enough test data to grade them with useful precision.

DEFINITION

Train/Test Split and the Test Set

A train/test split randomly partitions labeled cases, at the unit of prediction, into a training set used to fit and select models and a test set held out to evaluate the selected model. Test-set performance estimates out-of-sample performance — how the model will do on cases it has not seen — which is the only performance that matters for a predictive model's real job. The estimate is honest only if the test set contributes nothing to fitting, preprocessing, or model selection, and is consulted once for the final model rather than repeatedly during development.

Source: Adapted from James et al. (2021) and scikit-learn developers (2026b).

The definition's last clause carries the discipline most often violated in practice, so it earns its own paragraph. A test set consulted repeatedly stops being a test set. If the analyst fits a model, checks test error, adjusts, checks again, and repeats, the adjustments are being steered by the test data — a slow-motion version of training on it, and a cousin of Table 8.2's split contamination. By the tenth consultation, the “held-out” error is flattery with extra steps. The working protocol, which the labs of Section 8.12 follow to the letter: development decisions — which features, which model family, how much flexibility — are made using the training data alone, with Section 8.9's cross-validation supplying out-of-sample estimates during development; the test set is opened at the end, once, to grade the finalist; and if development resumes after that, the honest analyst treats the old test error as spent. On StyleCraft's eight thousand customers this protocol costs little and buys the deliverable its central sentence: “on sixteen hundred customers the model never saw, the average error was X dollars” — a sentence the CFO can trust precisely because of everything the protocol forbade.

One qualification completes the section, and it is more important than it looks, because the chapter later teaches drift. It is tempting to say that the test customers “stand in for the future.” They do not, quite. A random customer holdout estimates cross-sectional generalization at the historical snapshot: how the model performs on other customers drawn from the same population, at the same moment in the business's life, under the same acquisition mix, pricing, and merchandising. It does not measure time-forward transport — whether a relationship estimated at a December 2025 snapshot still holds at a June 2026 scoring date after the world has moved. Measuring that requires a different design: fit on an earlier snapshot, evaluate on a later one, or, once deployed, monitor realized error as each fresh outcome window matures. Standard random folds assume a stable sampling distribution, and where the intended generalization is explicitly into later periods, a time-ordered evaluation is the honest instrument (scikit-learn developers, 2026b).

The random split is nonetheless the right design here, and the reason is worth stating so the qualification is not mistaken for a defect. The frame already handled time: the wall of Section 8.3 lives inside the label construction — every customer's features stop at the snapshot and every label starts after it — so shuffling customers cannot shuffle the future into the past. What the shuffle cannot do is manufacture a second point in time, which is why the deliverable's

performance sentence names its population (“customers like these, at this snapshot”) and the monitoring plan of Section 8.10 exists to cover the rest. When the unit of prediction is itself a time period, as in Chapter 10's daily forecasting, random splitting breaks even the first guarantee, and a time-aware holdout is mandatory rather than supplementary.

> **BRIDGE**
> Chapter 10 will show why forecasting cannot randomly split its data — and what time-aware holdouts do instead.

8.6 Measuring Error in Dollars and in Decisions: MAE, RMSE, and Capture

An honest test set answers where to measure; this section answers what to measure, and the answer has two parts, because the Backstage program asks two different questions of the same predictions. The first question is calibration: how close is a customer's predicted dollar amount to what they actually spend? The second is the decision: which eight hundred customers appear above the cutoff? Those are not the same question, a model can improve on one while worsening on the other, and a frame that declares only the first has quietly agreed to be graded on something other than its job.

Take calibration first, because it supplies the vocabulary. The raw material is the per-customer error — actual label minus predicted value, the out-of-sample sibling of Section 7.6's residual — and the reporting need is a single number that summarizes sixteen hundred such errors in units a decision can use. Two summaries dominate numeric prediction, both built by this guide's oldest recipe (aggregate the individual deviations, per Chapter 5), and their difference is a policy choice about big misses. The mean absolute error takes each customer's error, strips its sign, and averages. Its virtues are bluntness and units: an MAE of 54 means the model's predictions miss actual six-month spend by 54 dollars on average, a sentence any stakeholder can absorb, denominated in the same dollars as the budget. The root mean squared error squares each error before averaging, then takes the square root to return to dollars. The squaring is the policy: it weights a single 300-dollar miss like nine separate 100-dollar misses, so RMSE is dragged upward by the largest errors far more than MAE is. Neither metric is more correct; they encode different sensitivities, and the gap between them is itself diagnostic — RMSE materially above MAE on the same predictions announces that the error is concentrated in a few large misses rather than spread evenly (Willmott & Matsuura, 2005), which on StyleCraft's data means one thing before anyone plots it: the whales and the lumpy occasion baskets of Chapter 5's concentration analysis are where the model hurts.

DEFINITION

MAE and RMSE

Mean absolute error (MAE) is the average of the absolute differences between actual and predicted values: the typical miss, in the target's native units, with every error weighted equally. Root mean squared error (RMSE) is the square root of the average squared difference: also in native units, but disproportionately sensitive to large errors, so that it exceeds MAE whenever misses are uneven and exceeds it greatly when error is concentrated in a few cases. Both are computed out of sample; the choice between them is a statement about which misses the decision cares about most.

Source: Adapted from Willmott and Matsuura (2005) and James et al. (2021).

Now the decision question, and the correction it forces on a tempting argument. It is easy to justify leading with MAE by saying that a wasted twenty-dollar slot costs the same whether the prediction behind it was moderately or catastrophically wrong, so the program's economics do not square the pain. The premise is true and the inference does not follow, because the cost of the decision is not a function of the point-prediction error at all. What costs money is whether an error moved a customer across the selection threshold, who was displaced from the list when it did, and how far the ranking shifted. A catastrophic prediction error four hundred ranks below the cutoff costs exactly nothing; a small error at rank 800 changes who receives a lookbook. Error metrics summarize distance; the program acts on order.

The metric that grades order under a fixed capacity is capture. Score every customer, sort descending, take as many customers as the budget funds, and compute what share of the population's actual outcome-window spend those customers turned out to account for. A model whose funded customers capture 34 percent of next-window spend is ordering customers better than one whose selection captures 28 percent, and the difference converts directly into dollars by multiplying the percentage-point gap by total outcome spend. Because the incumbent rule produces its own ranked list, the two can be graded head to head on exactly the artifact the meeting is deciding between, which makes the incremental capture over the incumbent the single most persuasive number the project produces.

One quantity inside that paragraph has to be settled rather than assumed, because it decides what the metric means: how many customers is “as many as the budget funds,” as a share of the population being ranked? The Backstage budget funds eight hundred treatments. That is a capacity, not a proportion — it does not grow if the eligible population turns out to be larger, and it does not shrink if the eligibility rule of Section 8.3 excludes more customers than anyone expected. The selection share is therefore computed rather than typed: eight hundred divided by the population the model can honestly score. On StyleCraft it lands near a tenth, which is why this guide is comfortable calling the exhibit top-decile capture in conversation and why Chapter 9 generalizes it to deciles. The labs still compute it, because a proportion typed into a metric is a proportion nobody re-derives when the population changes.

The frame of Table 8.1 therefore declares three metrics with three different jobs, and the labs report all three for every candidate. Capture at the capacity share is the decision metric, because

the program acts through a fixed ranked cutoff. MAE is the lead calibration metric, because it gives an interpretable typical miss in the currency of the budget and keeps the deliverable's language honest about how much the model does not know. RMSE is reported beside MAE as the concentration alarm. A different decision could reorder them: if predictions fed inventory commitments, where one enormous miss costs more than many small ones combined, RMSE's sensitivity is the point, and if every customer received a personalized offer amount, calibration would become the decision metric rather than its companion. The general rule follows the asymmetric-cost discipline of Section 2.7 — choose the metric whose weighting matches the decision's actual cost structure — and note, for honesty, that percentage-based error metrics are deliberately absent here; they have a characteristic failure on near-zero actuals, and this guide takes them up where they natively live, in Chapter 10's forecasting.

CONCEPT

Grade the Model on the Artifact the Meeting Is Choosing

Every predictive deliverable in this guide reports the metric that matches the act. When the act is a ranked list under a capacity constraint, the model is graded on the list: what share of the outcome the funded customers capture, against what the incumbent's funded customers capture at the same capacity share, with the overlap counted so the meeting can see how many of the eight hundred names actually change. When the act is a dollar commitment per customer, the model is graded on calibration. Reporting only a distance metric for a ranking decision is not wrong arithmetic; it is a quiet substitution of a question the analyst can answer for the question the business asked. Chapter 9 develops the full family — lift, gains, capture by decile — where ranked targeting is the chapter-long subject; here the top-decile version carries the decision.

Source: Course concept developed for this guide, informed by Provost and Fawcett (2013).

8.7 Baselines, Operationalized

Section 2.7 introduced baselines as interpretive discipline: no number means anything until you know what it is being compared against. Section 7.7 revealed R^2 as secretly a baseline comparison — the model against guessing the mean. This section completes the thread's promised operationalization: in predictive modeling, the baseline stops being a lens and becomes an opponent — a fully specified, deliberately simple prediction rule, evaluated on the same folds and the same test customers with the same metrics as the model, whose defeat is the model's entry fee. The reframing sounds small and changes everything about how results are reported: a model with a test MAE of 54 dollars is not good or bad in itself; it is good if the best naive rule scores 80 and embarrassing if the best naive rule scores 55.

DEFINITION

Baseline Model

A baseline model is a deliberately simple prediction rule — requiring no fitting beyond, at most, one summary statistic — evaluated under exactly the conditions of the candidate model: same folds, same test set, same metrics. Standard baselines include the mean baseline (predict the training mean for everyone; numeric targets), the majority-class baseline (predict the most common class for everyone; categorical targets), and the last-value baseline (predict that the last observed value, or window, repeats). A predictive model's demonstrated value is its margin over the strongest applicable baseline, not its raw score.

Source: Adapted from Provost and Fawcett (2013).

Table 8.3 gives the three standard baselines their StyleCraft form. The mean baseline is the floor: predicting the training-set average of spend_next6m for every customer uses no customer information at all, and a model that cannot beat it has learned nothing — it is R^2's comparison (Section 7.7), now made to stand in the open and fight on held-out customers. It is also the baseline that needs one qualification on the decision metric, and the qualification is more instructive than it looks. A rule that gives every customer the same score produces no ranking at all, so its capture is not a property of the rule; it is a property of whatever order the rows happened to arrive in, and reported carelessly that is row order masquerading as performance. Reported honestly — by breaking the ties on a declared random draw, which is what the lab does — it becomes the quantity everyone actually wanted: the capture a program would get by selecting customers at random, which is simply the selection share itself. That is the floor every candidate must clear, and it is a fact about random ordering rather than about the mean. The majority-class baseline is the floor's categorical sibling, recorded here at concept level because this chapter's target is numeric; Chapter 9 will show it doing serious work, where “predict that nobody churns” turns out to embarrass many a classifier. The last-value baseline is the one with teeth in this chapter, because it is not a strawman — it is the incumbent. “Predict that the next six months look like the last six” is exactly the CRM operations lead's sorted spreadsheet, expressed as a prediction rule, and it is often strong: trailing spend genuinely carries signal, which is why the rule survives in so many businesses. The lab will discover its designed weakness — the miniature of Lab 8.1 shows it losing to the mean baseline on lumpy customers — but the weakness must be demonstrated, not asserted, and on the same field the model plays on.

Table 8.3

The three standard baselines, in StyleCraft form

Baseline	Rule	StyleCraft instance	What beating it proves
Mean	Predict the training mean for everyone	Every customer predicted the average spend_next6m; with ties broken on a declared random draw, its capture is the selection share — the random-ordering floor	The model learned something from customer differences
Majority class	Predict the most common class for everyone	(Categorical targets; operational in Chapter 9)	The model beats blind agreement with the crowd
Last value	Predict the previous window repeats	Each customer predicted their own July–December 2025 spend	The model beats the incumbent rule — and the sorted spreadsheet costs nothing

CONCEPT

The Baseline Is the Opponent

Every predictive deliverable in this guide reports at least two baselines beside the model, specified before the model exists — because a baseline chosen afterward can be chosen to lose. The practice has three payoffs. It converts model quality from an absolute into a margin, which is the only form a budget argument can use: "the model's list captures X dollars more than the sorted spreadsheet's list" prices the model; a naked MAE does not. It disciplines humility: on many marketing problems the last-value baseline is genuinely hard to beat, and knowing that early redirects effort from modeling to framing, where the leverage usually is.

And it brackets honest performance from both sides, which is the second half of the chapter's red-flag logic. A model performing far worse than every honest baseline has a defect or a broken frame — a mis-specified feature, a mismatched population, a target built at the wrong grain. A model performing implausibly better than every honest baseline warrants an urgent leakage audit. Remember which direction is which: for MAE and RMSE, lower is better, so "far below the baselines" is the suspicious result and "far above them" is the broken one. That sentence is worth reading twice, because the error metrics run the opposite way from the capture metric, where higher is better and the same two suspicions simply swap ends.

Source: Course concept developed for this guide, informed by Provost and Fawcett (2013).

The section closes with the thread's forward arc stated once, in this chapter's terms: baselines began as interpretation (Section 2.7), operationalize here as opponents, extend in Chapter 10 to time — where naive and seasonal-naive rules inherit the last-value role — and take visual form

in Chapter 12 as reference lines. One idea, four chapters, each adding machinery and none re-teaching it.

8.8 Overfitting, Underfitting, and the Bias–Variance Intuition

The holdout discipline of Section 8.5 exists to measure a gap; this section explains where the gap comes from and what its size means. Fit a sequence of models of increasing flexibility to the same training data — a straight line, then a curve, then a wigglier curve, then one flexible enough to pass near every point — and watch two error curves. Training error falls, or at least does not rise, with every step: more flexibility can only fit the fitting data better. Held-out error typically falls, then turns, then climbs. The turn is the most important shape in predictive modeling, and its two sides have names. A model on the left side, too simple to capture real structure — the mean baseline is the extreme case — is underfitting: it misses systematically, in the same direction for whole regions of customers, and both its training and its held-out errors are high. A model on the right side has begun learning its training data's noise: the accidental quirks, the specific whales, the coincidences that will not recur. It is overfitting, and its signature is the diverging pair — training error excellent and still improving, held-out error worsening — because every quirk memorized is a point of training flattery and a liability on customers whose quirks are different (James et al., 2021).

One precision about that first curve, because the lab will print it and the claim is easy to overstate. What is guaranteed, when nested polynomial terms are added to a least squares model, is that the training sum of squared errors cannot increase — least squares minimizes squared error, so a richer model can always reproduce the poorer one's fit and usually does better. Training RMSE therefore cannot rise with degree. Training MAE is a different summary of the same residuals, and nothing in the fitting criterion protects it: a richer model can reduce squared error while nudging the typical absolute miss slightly upward. The lab tracks training RMSE for the monotone claim and reports training MAE beside it as the quantity that generally improves without being required to.

DEFINITION

Overfitting and Underfitting

A model overfits when it captures patterns specific to its training data — noise, outliers, coincidence — that do not generalize, so that its training performance substantially exceeds its held-out performance and additional flexibility widens the gap. A model underfits when it is too simple to capture structure that does generalize, so that both training and held-out performance are poor. The two are diagnosed jointly from the training-versus-held-out error signature, not from either number alone.

Source: Adapted from James et al. (2021) and Domingos (2012).

The intuition underneath the U-shape is the bias–variance tradeoff, which this guide states at intuition level and uses forever after. A too-simple model is wrong on average — its errors have

bias, systematic and shared across any sample you might have drawn. A too-flexible model is wrong differently every time — refit it on a different sample of customers and it memorizes different noise, so its predictions have variance, swinging with the accidents of the draw. Total error carries both, flexibility trades one for the other, and the honest minimum lies between the extremes — which is why “more sophisticated” is not a direction of improvement in predictive modeling, and why Domingos (2012) lists the belief that it is among the field's canonical beginner errors. Two of this guide's running facts sharpen the intuition. Sample size moves the turn: with more training customers, noise averages out more, and flexibility becomes safer — eight thousand customers tolerate a richer model than eight hundred. And irreducible uncertainty sets a limit: StyleCraft's spend labels contain genuine randomness — the wedding that did or did not happen in the outcome window — that no feature measured at the snapshot can carry.

That limit deserves a careful sentence, because it is easy to turn into a number the analysis has not earned. This guide does not estimate an irreducible error and does not claim to know where the limit sits. What it claims is weaker and more usable: individual human purchasing is volatile, spend is concentrated in a small number of customers, repeated windows of the same customer's history disagree with each other substantially, and honest baselines on this data land in a known range — so a result far stronger than all of that context would suggest is implausible on its face and belongs in the audit queue rather than in the deck. Implausible relative to domain variability, outcome concentration, repeated-history stability, and the honest baselines: that is the standard, and it is a judgment supported by evidence rather than a threshold computed from a formula.

Table 8.4 collects the diagnostic signatures, because the pair of error numbers is the instrument the lab will actually read. The table's repair menu sits at the level this chapter needs: against overfitting — simplify the model, reduce the feature list to the frame's justified variables, or get more data; against underfitting — add real structure, which in this guide's practice usually means a feature the frame should have included, not a fancier algorithm. The menu's modesty is deliberate. The regularization machinery that automates the flexibility dial belongs to later coursework; what transfers to every future tool is the signature-reading skill. Note the column headings: during development the second number is a cross-validated estimate computed inside the training data (Section 8.9), not a test-set figure, because reading this table off the test set is how the test set gets spent.

Table 8.4

Reading the training-versus-validation error signature

Signature	Diagnosis	Response
Training error high, validation error high and similar	Underfitting — missing real structure	Add justified features or model structure; revisit the frame

Signature	Diagnosis	Response
Training error low, validation error much higher	Overfitting — memorizing training noise	Simplify; prune features; more data if available
Training and validation error both low and close together	Honest fit — the target zone	Report against baselines; proceed to the sealed test set, then Section 8.10
Validation error implausibly low relative to domain variability, outcome concentration, and the honest baselines	Too good — suspect leakage, not brilliance	Run the Section 8.11 audit before believing anything

8.9 Cross-Validation and the Discipline of Model Comparison

Section 8.5's protocol left a gap this section fills: development needs out-of-sample estimates too. Choosing between candidate models — the explanatory specification or a prediction-oriented one, the shorter or the longer feature list, a straight line or a curve — requires comparing their out-of-sample errors, but the test set is sealed until the end, and consulting it per candidate would spend it. The standard answer is cross-validation: rotate the holdout inside the training data. Partition the training customers into k equal folds — five is this guide's default; split at the unit of prediction, as always. Then, k times over, fit the candidate on all folds but one and evaluate it on the one left out, so every training customer serves once as validation and never validates a model that saw them. The k scores average into a single cross-validated estimate per candidate, with a bonus the single split cannot give: the spread across folds, which reads like every spread since Chapter 5 — a candidate whose error swings widely from fold to fold is telling you its performance depends on which whales it drew, the variance half of Section 8.8 made visible.

DEFINITION

Cross-Validation

Cross-validation estimates a model's out-of-sample performance using training data alone, by repeatedly refitting the model with one of k folds held out and averaging the k held-out scores. It is the standard instrument for development-time decisions — feature choices, model family, settings, and the amount of flexibility — allowing the true test set to stay sealed for the final grade. The averaged estimate remains honest only if every learned preprocessing step is refit inside each fold, and dishonest the moment any fold's validation data influences its own fitting.

Source: Adapted from James et al. (2021) and scikit-learn developers (2026b).

The definition's last two clauses are the reason this chapter's labs put every model inside a pipeline rather than fitting steps by hand. A pipeline is an object that chains learned steps — a standardizer, an imputer, a clustering model, the regression itself — and refits all of them

together whenever it is fitted. Handed to a cross-validation routine, it is refitted inside each fold automatically, so the standardizer of fold three never sees fold three's validation customers and the segmentation of fold three never learned its centroids from them. Doing the same work by hand is possible and reliably forgotten under deadline; scikit-learn's own guidance is to use pipelines precisely so that this discipline is structural rather than remembered (scikit-learn developers, 2026a). The consequence for Chapter 6's segmentation is direct and worth stating before the lab reaches it: a segment label is a learned feature, not a raw column, so a segmentation refitted on all labeled customers before the split lets the test customers shape the scaler and the centroids that training then uses. Inside a pipeline, the same segmentation is honest.

What must be owned by the analyst is the comparison discipline cross-validation serves, because model comparison is where predictive projects most often go quietly wrong — not through any single dishonest act but through drift in the conditions of comparison. The discipline has three clauses, each closing a real loophole. Same split: every candidate is estimated on the same folds, generated once with a fixed seed, because reshuffling between candidates lets chance pick the winner — with a mediocre model and enough reshuffles, some split flatters it. Same metric: the metrics declared in the frame (Section 8.2) grade every candidate, with the decision metric leading; a candidate that wins on RMSE after losing on the declared capture and MAE has not won — the goalposts have moved, and moving goalposts after results are visible is the model-comparison version of writing the hypothesis after the data. Same baseline: the baselines of Section 8.7 appear in every comparison table, all the way to the final one, because the question is never only “which model is best” but “does the best model beat the spreadsheet by enough to matter.”

The three clauses have a physical form in this chapter's labs: one leaderboard, built once, with a row per candidate and a column per declared metric, the two baselines occupying the first two rows, and every figure computed on the same five folds of the same training customers. Because the folds are shared, the leaderboard supports a comparison a table of separate averages cannot: the paired per-fold difference between a candidate and the incumbent. Pairing matters because folds differ in difficulty — a fold holding three whales is hard for everyone — so each candidate's own standard deviation across folds is inflated by difficulty the comparison should cancel. The honest question is not whether a candidate's mean beats the incumbent's mean by more than its own spread; it is whether the candidate beat the incumbent fold by fold. Only when the leaderboard is complete, and a selection rule written before the race has been applied to it, does the analyst name a finalist, refit it on all the training customers, and open the sealed test set — once. Everything before that moment is development; everything after it is reporting.

CONCEPT

One Leaderboard, Fixed Before the Race

Before any candidate is fitted, the analyst writes the leaderboard's rules — the folds, the metrics, the baselines, and the population — and every subsequent result joins that leaderboard unedited. The practice is predict-then-verify (Section 1.7) applied to model selection, and its enemy is not fraud but hope: the developer who has spent a week on a model wants it to win, and unfixed rules bend toward the wanted answer one small reasonable-seeming adjustment at a time.

Fixed rules also make delegation safe. An assistant asked to try ten model variants can be graded mechanically against the frozen leaderboard, which is exactly how Section 8.11 harnesses AI speed without inheriting AI judgment. When you read a claimed model comparison — a vendor's, a colleague's, an assistant's — the first question is not “what won?” but “when were the rules written, and was the test set part of the race or the finish line?”

Source: Course concept developed for this guide, informed by Provost and Fawcett (2013) and scikit-learn developers (2026b).

8.10 From Score to Decision: Expected Value and Deployment Realities

A graded model is still not a decision. The model's output is a score — a predicted spend_next6m per customer — and the Backstage question is an action: which eight hundred customers get the treatment. This section builds the translation, and then follows the model past the decision into the part of its life that analytics courses traditionally ignore and analytics careers cannot: deployment.

The translation's first form is the one the fixed budget dictates: rank and cut. Score every eligible customer, sort descending, take the top eight hundred. Under a hard capacity constraint the model's job is ordering quality, not clairvoyance — it does not matter whether the top customer's prediction is 700 or 900 dollars if the ranking puts the right customers above the line — and the deliverable therefore leads with the capture exhibit Section 8.6 declared: what share of actual outcome-window spend the model's funded customers capture, against the share the incumbent's funded customers capture at the same capacity, with the count of customers appearing on both lists. That head-to-head, on sealed test customers, is the single most persuasive table the project produces, because it grades the exact artifact the meeting is deciding about. One operational detail belongs in the deliverable rather than in the code comments: at the cutoff, ties must break the same way every run, or the eight hundredth name changes between the analysis and the campaign build. The lab declares a secondary sort key for exactly this reason.

The translation's second form asks the question the budget's size begs: is eight hundred even the right number? Answering it requires attaching money to predictions, and the instrument is expected-value framing: for any candidate action on any customer, expected value is each possible outcome's value weighted by its probability, minus the action's cost, and the action is worth taking where the expectation is positive (Provost & Fawcett, 2013). At concept level —

which is where this guide holds it, one notch above arithmetic and well short of decision theory — the framing does two jobs for the Backstage program. It converts the score threshold from an arbitrary rank into an economic line: treating a customer is justified where the treatment's expected incremental value exceeds twenty dollars. And it prices the two error types the opening case questions promised would differ: a wasted slot costs the treatment's twenty dollars; a missed high-spender costs the uncaptured margin of a customer the program existed to cultivate — an asymmetry in exactly Section 2.7's sense.

The word incremental in that paragraph is doing load-bearing work, and Section 8.1 flagged it in the brief so that it could be enforced here. This chapter's model predicts what a customer will spend. It does not predict what a customer will spend *because of the gift*, and no amount of out-of-sample accuracy converts the first quantity into the second. A perfectly calibrated spend model can rank the base beautifully and still send lookbooks to eight hundred customers who would have spent identically without one. That is not a flaw in the model; it is the boundary of what a predictive frame can establish, the same boundary Section 7.13's verb discipline enforced for coefficients, and only Chapter 11's instrument can cross it. The honest deliverable therefore prices the program two ways: the capture gap, which the analysis certifies, and the incremental return, which it explicitly does not — with a holdout design named as the purchase that would.

> **BRIDGE**
>
> Chapter 9 will make this asymmetry the whole instrument: when targets become yes/no, choosing the score threshold from the two error costs is threshold economics, and it is where the analyst meets the CFO.

Past the decision lies the deployed model's actual life, and this guide states its realities at concept level now so no student meets them first in production. A deployed model is a claim that the world it will score resembles the world it was trained on, and the world reliably breaks the claim in two distinct ways that deserve two distinct names. Data drift is change in the distribution or the quality of the inputs: StyleCraft's acquisition mix tilts suburban, a new store opens, the base gets younger, so the model is scoring customers whose feature combinations were rare in training. Concept drift is change in the relationship between inputs and outcome: the same recency, the same discount share, the same segment profile now imply a different level of future spending, because a pricing-and-packaging change restructured how customers buy (Gama et al., 2014). One term inside the first deserves precision, because it is often used as a synonym and is not one: covariate shift is the special case in which the predictor distribution changes while the conditional relationship to the outcome holds steady. Data drift is the umbrella; covariate shift is its benign corner. The distinction earns its keep in monitoring: data drift can be detected by watching the inputs alone, and under covariate shift a model may keep performing perfectly well, while concept drift cannot be diagnosed from input distributions at all and can degrade performance without moving a single input.

Performance decay is the observable consequence of either — measured predictive quality eroding as one or both accumulate, sometimes gradually, sometimes at a policy change's step

edge. The disciplines that answer them follow directly. Monitoring means tracking the deployed model's realized error on each period's fresh labels as they mature, against the benchmark the test set established, exactly as the test set once was — and, cheaply and separately, watching the input distributions, because an input alarm arrives months before an outcome alarm can. Retraining runs on a declared cadence tied to the horizon: a six-month-horizon model can only be fully graded six months after each scoring run, so the monitoring calendar is part of the frame, written before deployment, not improvised after the first bad quarter. And explicit triggers — a performance floor, a known structural change such as a repricing or a merchandising overhaul — force revalidation off-cadence, because concept drift's most expensive property is that it announces itself in the business before it announces itself in the metrics. Sculley et al. (2015), in the paper that named the field's hidden costs, put the engineering point this guide converts to an analyst's point: the model is the smallest part of the system, and the system — data feeds, definitions, monitoring, retraining, ownership — is what actually predicts. The analyst of record's obligations do not end at the test set. They end when the model is retired.

DEFINITION

Data Drift, Concept Drift, and Performance Decay

Data drift is post-deployment change in the distribution or quality of a model's input features, so that scored cases increasingly differ from training cases; covariate shift is its special case, in which the predictor distribution changes while the conditional relationship to the outcome remains stable. Concept drift is change over time in the relationship between the inputs and the target, so that the learned mapping no longer describes how the outcome is generated, and it cannot be diagnosed from the inputs alone. Performance decay is the resulting erosion of predictive quality, observed by monitoring realized error as fresh outcomes mature. All three are managed rather than prevented: by monitoring inputs and outcomes against the training-window benchmark, retraining on a cadence tied to the horizon, and declaring triggers that force revalidation when a structural change is known.

Source: Adapted from Gama et al. (2014) and Sculley et al. (2015).

8.11 AI as a Predictive-Modeling Assistant

Chapter 7's AI section ended with a division of labor: the assistant proposes relationships; the analyst tests whether they are honestly stated. Predictive work sharpens the division again, because the assistant's fluency now extends end to end. Prompted with a feature table and “build me a model that predicts customer spend,” a current assistant will construct the label, engineer features, split the data, fit several model families, tune them, and report a polished comparison table — an entire pipeline, delivered in one response, runnable as pasted. The productivity is real and this guide uses it. The hazard is equally real and has a precise shape: every failure mode this chapter has named is a failure the assistant commits fluently, silently, and with congratulations. A leaky pipeline does not look different from an honest one in the chat window. It looks better.

The characteristic failure modes, named so the audit can target them. The convenient-table leak: asked to predict future spend, the assistant builds features from whatever table it was given — including columns computed across the outcome window — because nothing in the prompt drew the snapshot wall; temporal leakage by default, Table 8.2's first row, and the single most common defect in AI-drafted pipelines. The celebration of the impossible: the leaky pipeline reports R^2 of 0.97, and the assistant's narration calls it excellent — assistants inherit their training data's enthusiasm for high fit statistics, not this chapter's suspicion of them, so the too-good reflex is precisely the judgment that does not transfer. The missing opponent: the report quotes MAE, RMSE, and R^2 with no baseline anywhere, leaving no way to know whether the model beats the sorted spreadsheet it exists to replace. The metric mismatch: distance metrics reported for a ranked-list decision, with no capture figure and no head-to-head against the incumbent's list, because nothing in the prompt named the act. The vocabulary shuffle: “96 percent accuracy” applied to a numeric target — accuracy is classification vocabulary (Chapter 9's, where its own failures await), and its appearance in a numeric-prediction claim is a tell that the claimant is quoting whatever number sounded best. The hygiene slips: preprocessing fitted on pooled data, the test set consulted in a tuning loop, complexity chosen by watching test error, the comparison metric switching mid-report — each individually small, each a clause of Sections 8.5 and 8.9 violated without announcement. And the deletion helpfulness: zero-spend labels “cleaned” away as missing data, quietly deleting the customers the model most needs to learn (Section 8.3).

Where the assistant genuinely helps, use it deliberately. Pipeline mechanics from a written frame — the split with a stated seed, the fold generator, the metric functions, the baseline estimators, the leaderboard table — are exactly the well-specified boilerplate assistants draft reliably. Feature derivation code is safe when, and only when, the prompt supplies the derivation rules and the window boundaries explicitly; the lab's prompts do. Audit exhibits — the feature-window trace, the range checks of Section 8.4, the signature table of Section 8.8 — are tedious, mechanical, and ideal for delegation. Error analysis is a strength: asked “which customers does this model miss worst, and what do they share?”, an assistant surveys the residuals faster than any manual pass, and the answer — on StyleCraft's data, the lumpy occasion shoppers — feeds the next framing conversation. And the assistant remains a good rehearsal partner for the vendor meeting.

The audit that governs all of it extends Chapter 7's four points to the predictive setting. Five points, in order, run on every model — the assistant's, the vendor's, and yours — before any result is believed or repeated. Table 8.5 states the checklist; the box operationalizes it as the two-prompt pattern this guide has used since Chapter 6, with the audit standing between mechanics and narration.

Table 8.5

The five-point audit of a predictive model

Point	The check	Fails when
1. Frame	Unit, target, features, horizon, metrics, and baselines declared in writing; label defined with snapshot, both window boundaries, filters, and eligibility	The frame is implicit, or reconstructed after the results
2. Leakage	Every feature traced to its window and derivation; range checks for impossible values; Table 8.2's five routes checked	Any feature crosses the snapshot, derives from the label, or cannot be traced
3. Split hygiene	Split at the unit of prediction, seeded; every learned step inside a pipeline fitted on training folds only; complexity chosen by cross-validation; test set consulted once	The test set steered development, or a learned step saw pooled data
4. Baseline	Mean and last-value baselines on the same folds and the same test set, on every declared metric; the model's margin stated	No baseline reported, or the baseline chosen after the results
5. Error in decision units	The declared decision metric leads; MAE and RMSE read in dollars beside it; the result checked for plausibility against domain variability, outcome concentration, and the baselines	Metrics shuffle mid-report, or excellence is celebrated instead of audited

AI IN PRACTICE

The Pipeline That Must Survive Its Audit

A two-prompt pattern with the audit between, extending Section 7.11's. Prompt one, the frame as specification: paste the seven declarations of Table 8.1, the label definition with its four dates and eligibility rule, the feature list with derivation rules and the instruction that every feature be computed from transactions inside the declared feature window, the declared metrics and baselines, and the split and cross-validation protocol with its seed — then ask for mechanics only: “Build this pipeline exactly as specified; select among the candidates using five-fold cross-validation on the training customers only; report one leaderboard with capture, MAE, and RMSE for every candidate and both baselines; do not touch the test set; change nothing in the specification; narrate nothing yet.”

Then run Table 8.5 yourself, in order: confirm the frame came back unedited; trace three features to their windows and run the recency range check; confirm every learned step sits inside a pipeline and the test set is untouched; confirm both baselines are present on every metric and the margin is computed; read the signature against Table 8.4, and if anything looks too good, stop and trace. Prompt two, only after the pipeline survives and the finalist has been graded once on the test set: “Draft the comparison paragraph for a marketing VP: the model's capture at the program's capacity against the incumbent list, its test MAE against both baselines in dollars, and no claim that any predicted difference is caused by anything.” Audit the draft's verbs against Section 7.2's registry — prediction language earns “expected” and “predicted,” never “will drive” — and file both exchanges per Appendix D. Predictions first, per Section 1.7; the analyst of record signs the scores.

Source: Course concept developed for this guide, informed by scikit-learn developers (2026a) and Kaufman et al. (2012).

8.12 Hands-On Application in Python and Google Colab

The preceding sections built the disciplines; this section runs them against the Backstage decision, in two labs that mirror the chapter's argument. Lab 8.1 builds the frame's raw material and the opponents: the miniature by hand, then the snapshot-safe feature table, the two windows, the eligible labeled population, and the split that seals the test customers. Lab 8.2 runs the race: a leaderboard of candidates cross-validated inside the training data, an overfitting demonstration conducted without ever consulting the test set, the finalist opened against the sealed customers once, a deliberate leak built and caught, and the score translated into the list. The labs use the certified transactions, customers, and products files; the feature table is rebuilt from Table 6.5's derivation rules with the analysis date moved back to the snapshot. AI assistants may draft any code cell (Appendix C has templates; Appendix A covers Colab mechanics); every output is predicted before it is computed, and every exchange is documented per Appendix D.

Two conventions hold across every cell, and both are this chapter's disciplines made structural. Every model, including every baseline, is an estimator evaluated through the same cross-validation call on the same folds, so that “same split, same metric, same baseline” is enforced by the code rather than remembered by the analyst — the labs use scikit-learn

(Pedregosa et al., 2011) throughout. And every step that learns anything from data — a standardizer, a segmentation, the regression itself — lives inside a pipeline, so that it is refitted within each training fold and never estimates a single quantity from a customer it is about to be graded on (scikit-learn developers, 2026a).

8.12.1 Lab 8.1, Part A: Baselines by Hand in Miniature

The miniature is the ten-customer table Labs 6.1 and 7.1 built intuition on, now given the chapter's time structure: for each customer, spend in the six months before the snapshot (spend_prior6m, the feature side) and spend in the six months after (spend_next6m, the label). The values are rounded to whole dollars for hand work and consistent with the miniature's running profiles: the steady Urban Loyal Core customers repeat, the occasion shoppers lump, the lapsed customer stays dark, and one customer arrives almost from nowhere. Before running anything, predict which baseline wins on these ten customers — the mean, or the incumbent's last-value rule — and write one sentence of reasoning.

Code 8.1. Calculate the miniature baselines

```
import pandas as pd

mini = pd.DataFrame({
    "customer_id": ["C001","C002","C003","C004","C005",
                    "C006","C007","C008","C009","C010"],
    "spend_prior6m": [90.0, 400.0, 30.0, 0.0, 60.0,
                      0.0, 24.0, 70.0, 0.0, 0.0],
    "spend_next6m":  [80.0, 0.0, 30.0, 44.0, 60.0,
                      0.0, 0.0, 50.0, 38.0, 312.0],
}).set_index("customer_id")

mean_pred = mini["spend_next6m"].mean()   # mean baseline
lv_pred   = mini["spend_prior6m"]         # last-value baseline

err_mean = mini["spend_next6m"] - mean_pred
err_lv   = mini["spend_next6m"] - lv_pred

mae_mean, mae_lv   = err_mean.abs().mean(), err_lv.abs().mean()
rmse_mean = (err_mean ** 2).mean() ** 0.5
rmse_lv   = (err_lv ** 2).mean() ** 0.5

print(f"mean:       MAE {mae_mean:6.2f}  RMSE {rmse_mean:6.2f}")
print(f"last-value: MAE {mae_lv:6.2f}  RMSE {rmse_lv:6.2f}")

# The decision metric on ten rows: with one slot, who does each rule pick,
# and what share of the ten customers' next-six-month spend does it capture?
total = mini["spend_next6m"].sum()
ranked  = mini.sort_values("spend_prior6m", ascending=False)
pick_lv = ranked["spend_next6m"].iloc[0]
print(f"one-slot capture, last-value rule: {pick_lv / total:.1%}")
best = mini["spend_next6m"].max()
print(f"one-slot capture, perfect ranking: {best / total:.1%}")
```

Expected output: two lines of MAE and RMSE, then two capture figures showing what a single-slot program would have captured under the incumbent rule and under a perfect ranking.

Input: ten literal customers. Transformation: two prediction rules, three metrics. Output: the leaderboard of Section 8.7 at a scale where every figure can be checked by hand. The last two lines are the point of the cell's revision from a purely calibration-based exercise: they ask the ranking question rather than the distance question, and on these ten customers the two questions have spectacularly different answers.

VERIFICATION CHECK

Before you run: compute both baselines by hand. The label mean is 614 ÷ 10 = 61.40 dollars. The mean baseline's ten absolute errors (18.60, 61.40, 31.40, 17.40, 1.40, 61.40, 61.40, 11.40, 23.40, 250.60) sum to 538.40, so its MAE is 53.84; its squared errors average 7,642.44, so its RMSE is 87.42. The last-value baseline's errors (10, 400, 0, 44, 0, 0, 24, 20, 38, 312) sum to 848, so its MAE is 84.80; its squared errors average 26,180, so its RMSE is 161.80. Then predict the capture lines: if the program had one slot and filled it by trailing spend, which customer would receive it, and how much of the ten customers' next-window spend would that customer account for?

After you run: reconcile every figure to the cent, then confront two results most students do not predict. The first is that the incumbent rule loses to the crudest baseline in the book on both distance metrics — and loses while being the better predictor for half the customers. For the steady core (C001, C003, C005) and the dark customer who stays dark (C006), yesterday repeats and last-value is nearly perfect. The rule is destroyed by four customers: C002, whose 400-dollar November baskets do not recur; C010, whose 312-dollar occasion arrives from a prior spend of zero; and C004 and C009, who ignite from nothing. This is Chapter 5's lumpiness finding wearing prediction clothes, and the RMSE gap — 161.80 against 87.42, proportionally wider than the MAE gap — is Section 8.6's concentration alarm ringing on ten rows. The second result is sharper: the incumbent's one slot goes to C002, who spends nothing at all in the outcome window, so the rule captures zero percent of what a single perfect pick would have captured. Distance and decision are different questions, and the second one is the program's. Write the implication in one sentence: a model earns its budget line only if it can keep last-value's wins on the steady customers while fixing its catastrophes on the lumpy ones — and the features that could do that (occasionwear share, purchase rhythm, engagement) are precisely what the sorted spreadsheet ignores.

Investigate if: your hand arithmetic disagrees with the printed figures anywhere. Every number in this cell is reproducible with a calculator, and a mismatch is arithmetic rather than convention — unlike Chapter 7's covariance denominators, nothing here depends on a software default.

8.12.2 Lab 8.1, Part B: The Windows, the Labels, and the Eligible Population

Part B builds the real thing. Begin by writing the frame — the seven declarations of Table 8.1, the label definition with its four dates, the eligibility rule, and the split protocol — into a markdown cell at the top of the notebook. The frame precedes the code, and the lab is graded on that order.

Then build the feature side of the wall. Rather than describing the rebuild in a comment, the lab supplies it as a function, because a chapter distributed on its own cannot assume the reader has Lab 6.1's notebook open, and because a feature builder that takes its window as an argument is the only kind that can be pointed at a snapshot without being edited. Code 8.2 is Table 6.5's derivation rules with the analysis date and window start promoted to parameters. The companion repository ships the identical function as stylecraft_features.build_customer_features, so a student who prefers the import can use it and get the same columns; the version printed here exists so that nothing in the pipeline is a black box.

Code 8.2. Roll up the feature window to customer grain

```
import numpy as np
import pandas as pd

# The four dates the frame declared. Every window has two ends, and both
# ends are enforced in code rather than trusted to the file's contents.
FEATURE_START = pd.Timestamp("2024-07-01")
SNAPSHOT      = pd.Timestamp("2025-12-31")
OUTCOME_START = pd.Timestamp("2026-01-01")
OUTCOME_END   = pd.Timestamp("2026-06-30")


def snapshot_rollup(transactions, customers, products,
                    snapshot, window_start):
    """Lab 6.1's aggregation, parameterized by window rather than
    hard-coded. One row per customer with at least one order inside
    [window_start, snapshot]. Nothing dated after the snapshot can reach
    the output, because nothing after it is ever read. Normalizing here
    rather than in the caller keeps a late-in-the-day December 31 order
    inside the feature window."""
    tx = transactions.assign(
        order_date=transactions["order_date"].dt.normalize())
    tx = tx[tx["order_date"].between(window_start, snapshot,
                                     inclusive="both")]

    # "min" and "first" are honest only if the attribute is constant
    # within the order. Check, do not assume (Chapter 4's rule).
    for col in ["customer_id", "order_date", "channel"]:
        assert (tx.groupby("order_id")[col].nunique(dropna=False)
                  .eq(1).all()), f"an order carries multiple {col} values"

    orders = (tx.groupby(["order_id", "customer_id"], as_index=False)
                .agg(order_date=("order_date", "min"),
                     order_revenue=("line_revenue", "sum"),
                     channel=("channel", "first")))
    # Shares live at line grain (Section 6.6), not in order totals.
    lines = tx.merge(products[["product_id", "is_occasionwear"]],
                     on="product_id", how="left", validate="many_to_one")
    assert lines["is_occasionwear"].notna().all(), "unmatched product_id"
    shares = (lines
              .assign(
                  disc=lambda d: d["line_revenue"]
                                   .where(d["discount_pct"] > 0, 0.0),
                  occ=lambda d: d["line_revenue"]
                                  .where(d["is_occasionwear"], 0.0))
              .groupby("customer_id")
              .agg(discounted_revenue=("disc", "sum"),
                   occasion_revenue=("occ", "sum")))
    return (orders.groupby("customer_id")
                  .agg(last_order=("order_date", "max"),
                       frequency=("order_id", "nunique"),
                       monetary=("order_revenue", "sum"),
                       store_orders=("channel",
                                     lambda s: (s == "Store").sum()))
                  .join(shares)
                  .join(customers.set_index("customer_id")
                                 [["signup_date"]]))
```

Expected output: nothing — this cell declares the four dates and defines a function.

Input: the certified line-grain, customer, and product files. Transformation: a window filter, an order-grain roll-up, and two line-grain shares. Output: one row per customer with at least one order inside the declared window, carrying the raw counts and sums that Code 8.3 turns into features. The split between this cell and the next mirrors Chapter 6's own Code 6.5 and Code 6.6, for the same reason: aggregation and derivation fail differently and are checked differently.

The function's most important property is negative: it never reads a transaction outside the window it was given, so temporal leakage cannot enter through it no matter what the caller does downstream. That is why the window filter is the first statement rather than a later convenience — and why the same function will be used in Code 8.13 to manufacture the leak, by handing it a later snapshot, which is exactly how the error happens in practice.

Code 8.3. Derive the ratio and intensity features at the snapshot

```
def build_customer_features(transactions, customers, products,
                            snapshot, window_start):
    """Table 6.5's ratio and intensity derivations, computed at a declared
    snapshot. Indexed by customer_id; every column is snapshot-safe by
    construction, because snapshot_rollup read nothing later."""
    cf = snapshot_rollup(transactions, customers, products,
                         snapshot, window_start)

    signup = cf["signup_date"].clip(lower=window_start)
    observed_days = (snapshot - signup).dt.days
    cf["exposure_months"]    = np.maximum(observed_days / 30.44, 1.0)
    cf["recency_days"]       = (snapshot - cf["last_order"]).dt.days
    cf["tenure_days"]        = (snapshot - cf["signup_date"]).dt.days
    cf["aov"]                = cf["monetary"] / cf["frequency"]
    cf["orders_per_month"]   = cf["frequency"] / cf["exposure_months"]
    cf["revenue_per_month"]  = cf["monetary"] / cf["exposure_months"]
    cf["discount_share"]     = cf["discounted_revenue"] / cf["monetary"]
    cf["store_share"]        = cf["store_orders"] / cf["frequency"]
    cf["occasionwear_share"] = cf["occasion_revenue"] / cf["monetary"]

    flags = (customers.set_index("customer_id")
                      [["app_user", "email_opt_in"]].astype(int))
    return (cf.join(flags)
              .drop(columns=["last_order", "signup_date", "store_orders",
                             "discounted_revenue", "occasion_revenue"]))
```

Expected output: nothing — this cell defines the function the next one calls.

Input: the roll-up. Transformation: exposure, recency, tenure, the three shape ratios, and the two intensity rates, all measured against the snapshot rather than against a hard-coded analysis date. Output: the customer feature table as StyleCraft could have built it on December 31, 2025, and nothing else.

Every derivation here is Table 6.5's, unchanged in substance and changed in one respect that matters: the date it counts back from is an argument. A feature builder that hard-codes its analysis date cannot be pointed at a snapshot without being edited, and a pipeline whose feature

step must be edited by hand between runs is a pipeline that will eventually be run with the wrong date. Parameterizing the window is a leakage control, not a matter of style.

Code 8.4. Rebuild the feature table and check its contract

```
# The certified files are loaded per Appendix A as transactions_clean,
# customers, and products. Check the INPUTS before building anything:
# a builder cannot repair a dimension table with duplicate keys.
assert customers["customer_id"].is_unique
assert products["product_id"].is_unique
assert transactions_clean[["order_id", "customer_id",
                           "order_date"]].notna().all().all()

cf_snap = build_customer_features(transactions_clean, customers,
                                  products, snapshot=SNAPSHOT,
                                  window_start=FEATURE_START)

# A feature contract stated in code fails loudly when an upstream build
# changes; a contract stated in prose fails silently, six cells later.
EXPECTED = {"frequency", "monetary", "exposure_months", "recency_days",
            "tenure_days", "aov", "orders_per_month", "revenue_per_month",
            "discount_share", "store_share", "occasionwear_share",
            "app_user", "email_opt_in"}
assert set(cf_snap.columns) == EXPECTED, "the feature contract changed"
assert cf_snap.index.is_unique
assert cf_snap.notna().all().all(), "a feature is undefined"
assert (cf_snap["recency_days"] >= 0).all()
tol = 1e-9
for share in ["discount_share", "store_share", "occasionwear_share"]:
    assert cf_snap[share].between(-tol, 1 + tol).all(), share

print("customers with a feature-window purchase:", len(cf_snap))
print(cf_snap[["recency_days", "frequency", "monetary", "aov",
               "discount_share", "occasionwear_share"]].describe().round(2))
```

Expected output: a customer count for the feature window, then a six-column summary in which recency is never negative and every share lies between 0 and 1; eleven assertions pass silently.

Input: the certified files and the two functions above. Transformation: three input checks, one call, and eight output checks. Output: cf_snap, the snapshot-safe feature table, plus the evidence that it is what it claims to be. The checks run in both directions on purpose. Before the build, the cell confirms what the builder cannot repair: duplicate keys in a dimension table would silently multiply rows through the joins, and a missing order identifier would quietly drop an order from a customer's history. After the build, the column-set assertion is the guard worth copying into every project: a feature contract stated in code fails loudly when an upstream build changes, where a contract stated in prose fails silently six cells later, in a model whose numbers look fine.

Code 8.5. Enforce both windows and construct the labeled eligible table

```
# Dates are recorded at day grain; normalizing makes the boundary
# comparisons exact rather than dependent on a stored time component.
tx = transactions_clean.assign(
    order_date=transactions_clean["order_date"].dt.normalize())

in_feat = tx["order_date"].between(FEATURE_START, SNAPSHOT,
                                   inclusive="both")
in_out  = tx["order_date"].between(OUTCOME_START, OUTCOME_END,
                                   inclusive="both")
tx_feat, tx_out = tx[in_feat], tx[in_out]

# Both ends of both windows, asserted rather than assumed.
assert tx_feat["order_date"].min() >= FEATURE_START
assert tx_feat["order_date"].max() <= SNAPSHOT
assert tx_out["order_date"].min()  >= OUTCOME_START
assert tx_out["order_date"].max()  <= OUTCOME_END
assert len(tx_feat.index.intersection(tx_out.index)) == 0, \
    "the two windows overlap"

labels = (tx_out.groupby("customer_id")["line_revenue"]
                .sum().rename("spend_next6m"))

# The incumbent's own column: the trailing six months of the feature window.
prior6 = (tx_feat[tx_feat["order_date"] >= pd.Timestamp("2025-07-01")]
          .groupby("customer_id")["line_revenue"]
          .sum().rename("spend_prior6m"))

# ELIGIBILITY (Section 8.3), both conditions evaluable at the snapshot:
# signed up on or before it, and at least one purchase inside the window.
signed_up = set(customers.loc[customers["signup_date"] <= SNAPSHOT,
                              "customer_id"])
purchased = set(cf_snap.index)
eligible  = sorted(signed_up & purchased)

model_df = (cf_snap.loc[eligible]
            .join(labels)
            .join(prior6)
            .fillna({"spend_next6m": 0.0, "spend_prior6m": 0.0}))

print("customers in the file:              ", len(customers))
print("signed up by the snapshot:          ", len(signed_up))
print("purchased inside the feature window:", len(purchased))
print("eligible (both conditions):         ", len(model_df))
print("signed up, no feature-window purchase (activation population):",
      len(signed_up - purchased))
print("acquired after the snapshot -- outside the population this",
      "training snapshot can support:", len(customers) - len(signed_up))
```

Expected output: six counts that reconcile — customers in the file, signed up by the snapshot, purchasers in the feature window, eligible customers, the activation population, and the post-snapshot acquisitions — and five assertions passing silently.

Input: the certified transactions and customers, plus the snapshot feature table. Transformation: two window filters with both boundaries enforced, one

trailing-window aggregation, and a two-condition eligibility rule. Output: the modeling table, one row per eligible customer, with features on one side of the wall and the label on the other.

Three details carry the section's arguments. The window filters name four dates rather than one, so the feature window cannot silently absorb history older than the declaration and the outcome window cannot silently absorb months beyond the horizon; the assertions then check both ends of both, and the intersection check proves the two windows share no transaction. The eligibility rule intersects two conditions, both evaluable at the snapshot, which is what keeps it clear of Table 8.2's population-leakage row. And the printed counts partition the customer file exactly, in the manner Chapter 6 established for its clustering base: the eligible customers, the signed-up customers with no feature-window purchase — an activation population, not a modeling population — and the customers acquired after the snapshot, whom this model was not built to score. Reporting the excluded groups by name is the difference between a declared population and a convenient one.

The fillna line deserves its own sentence, because deleting it would be the most consequential one-character edit in the lab. A customer eligible at the snapshot who bought nothing between January and June has no row in the labels series, so the join produces a missing value — and that missing value is not missing data. It is a label of zero, the outcome the model most needs to be able to anticipate (Section 8.3).

Code 8.6. Audit the labeled table before splitting

```
assert model_df.index.is_unique
assert model_df.notna().all().all(), "a feature or label is missing"
assert (model_df["spend_next6m"] >= 0).all()
assert (model_df["spend_prior6m"] >= 0).all()
assert (model_df["recency_days"] >= 0).all()
assert (model_df["tenure_days"] >= model_df["recency_days"]).all(), \
    "a customer purchased before signing up -- check the join"

# An exact count, not a quantile: ties at the 99th percentile can sweep in
# far more than one customer in a hundred.
n_top = max(1, int(np.ceil(len(model_df) * 0.01)))
top1 = (model_df["spend_next6m"].nlargest(n_top).sum()
        / model_df["spend_next6m"].sum())

# The program's capacity is a budget, not a proportion: $20 a slot and a
# fixed program budget fund a fixed number of customers. The selection
# share is that capacity divided by the population actually scored, and
# it is the share every capture figure in this chapter uses.
CAPACITY = 800
SELECTION_SHARE = CAPACITY / len(model_df)

print(f"labeled customers:             {len(model_df):,}")
print(f"zero-label share:              "
      f"{model_df['spend_next6m'].eq(0).mean():.1%}")
print(f"top {n_top} customers (1% by count) hold "
      f"{top1:.1%} of outcome spend")
print(f"program capacity:              {CAPACITY} slots")
print(f"selection share:               {SELECTION_SHARE:.1%}")
print(model_df["spend_next6m"].describe().round(2))
```

Expected output: the labeled row count, the zero-label share, the share of outcome spend held by the top one percent of customers, and an eight-row summary of the label; six assertions pass silently.

Input: the modeling table. Transformation: none — this cell computes nothing the model uses. Output: the evidence that the table means what the frame says it means. The assertions encode claims the prose has been making: labels are non-negative because they are sums of revenue, no feature is undefined because the eligibility rule guaranteed a denominator, and no customer purchased before signing up, which would indicate a broken join rather than an unusual customer.

The two printed shares are calibration for the too-good reflex, and they are worth recording in the notebook before any model exists. A base in which a quarter of eligible customers spend nothing in the outcome window, and in which the top one percent of customers hold a large share of the spend that does occur, is a base whose individual outcomes are volatile by construction. That is the context Section 8.8 said should replace an invented noise floor: not a computed limit, but a documented picture of how much of this target is genuinely unpredictable from anything measured six months earlier.

Code 8.7. Split the customers and freeze the test set

```
from sklearn.model_selection import train_test_split

train_df, test_df = train_test_split(model_df, test_size=0.20,
                                     random_state=42)
y_tr, y_te = train_df["spend_next6m"], test_df["spend_next6m"]

# Identities, not positions: every later comparison reuses these.
train_ids, test_ids = train_df.index, test_df.index
assert train_ids.intersection(test_ids).empty
assert len(train_ids) + len(test_ids) == len(model_df)

# Development diagnostics come from the TRAINING customers only. The test
# labels are stored above and deliberately not summarized here: knowing
# that the test half holds fewer whales, or a different zero rate, is
# knowledge that would leak into feature choices and expectations.
print(f"training customers: {len(train_ids):,}")
print(f"test customers:     {len(test_ids):,}")
print(f"training zero-label share: {y_tr.eq(0).mean():.1%}")
print(f"training mean / median:    {y_tr.mean():,.2f} / "
      f"{y_tr.median():,.2f}")
print(f"training 99th percentile:  {y_tr.quantile(0.99):,.2f}")
print("\nTEST SET SEALED. Its outcomes stay unopened until Code 8.14.")
```

Expected output: two identity assertions, a six-row comparison of the training and test halves, and the sealing notice.

Input: the audited modeling table. Transformation: a seeded 80/20 split at the customer grain. Output: two frames and, more importantly, two index objects — train_ids and test_ids — which every later cell reuses. Identities rather than positions is the point: a random seed reproduces which positions go where, not which customers, so any later comparison that re-splits a

differently ordered frame can quietly grade two models on two populations. Code 8.13 depends on this.

Notice what this cell does not print, because the omission is the discipline. It would be natural to compare the two halves here — zero-label shares, means, upper percentiles — to reassure yourself that the split was fair. That comparison reads the test labels, and a sealed test set is not sealed once its outcomes have been seen. Learning that the test half holds fewer whales, or a gentler mean, is knowledge that leaks forward into every later choice: which features feel necessary, how much flexibility feels safe, what result will feel plausible. None of that requires bad faith; it requires only a human who now knows something. The halves are compared in Code 8.14, after the grade is recorded, where the same numbers are context for reading a result rather than an influence on producing one.

What the cell prints instead is the training half's own profile, which is available without cost because the training labels are development data by definition. If the split were badly unbalanced, the cross-validated spreads of Code 8.11 would show it — folds drawn from a lumpy training half disagree with each other — and that signal, unlike a peek at the test set, is one the protocol permits.

8.12.3 Lab 8.2, Part A: The Candidates and the Cross-Validated Leaderboard

Lab 8.2 runs the race Section 8.9 specified. Five candidates enter, and the two baselines are candidates rather than commentary: they are fitted, scored, and ranked by the identical call on the identical folds, because a baseline evaluated by a different route is not evidence about the same question. The three models are chosen to make an argument the chapter has been building toward.

The first is the explanatory-style candidate: Chapter 7's interpretable predictors — recency, tenure, app use, email opt-in, discount share — refitted to the new frame, with behavioral cluster indicators standing where Chapter 7 put its segment dummies. Its presence is the chapter's continuity argument and its own quiet correction. Continuity, because the same equation family is being asked to do a different job under a different grade. Correction, because carrying that specification into a predictive frame exposes something the drivers analysis never had to think about: a segment label is not a raw column but a learned feature, produced by a scaler and a set of centroids estimated from data. Refit that segmentation on all labeled customers before the split and the test customers have shaped the very definitions the training data is described by — Table 8.2's split-contamination row, committed by a step nobody thinks of as a model. The lab's answer is structural: the clustering becomes a pipeline step, refitted inside every fold, assigning rather than learning from the customers it scores.

That repair costs something, and the chapter is explicit about the price rather than smuggling it. A clustering refitted inside each fold cannot carry Chapter 6's four named segments, because naming them was an act of judgment over full centroid profiles, performed once, on one estimation. What survives fold to fold is an ordering rule, not a nameplate — so the candidate's indicators say something like second-lowest purchase intensity rather than Suburban Occasion,

and two clusters carrying the same rank in different folds can differ materially in discount behavior or basket shape. The candidate is therefore called explanatory-style rather than the Chapter 7 specification. Recovering the named segments honestly would take a frozen assignment rule, established before this study and applied unchanged to every fold; that is a reasonable engineering answer and a different lab. Naming the limitation is this one's job.

The second is the prediction-oriented candidate, and the reason it exists is the sharpest methodological point in the lab. Chapter 7 chose its variables to make coefficients defensible. Nothing about that criterion selects the strongest predictive inputs, and the specification it produced omits, for prediction, almost everything a snapshot legitimately knows: feature-window spend, purchase frequency, average order value, orders and revenue per month, store and occasionwear mix — and the incumbent rule's own trailing-spend column, which belongs in the model precisely because a good model may beat the spreadsheet by keeping its signal and combining it with everything the spreadsheet ignores. Every one of those columns is computed inside the feature window, so every one of them is snapshot-safe; excluding them would not be caution but a category error, applying an explanatory chapter's variable-selection rule to a predictive chapter's job.

The third is a deliberately more flexible version of the second — the same features expanded to quadratic terms — so the leaderboard contains at least one candidate whose extra flexibility might or might not pay for itself. Whether it does is the leaderboard's business, not the analyst's hope, and Part B examines the general shape of that question.

Code 8.8. Express the two baselines as estimators

```
import numpy as np
from sklearn.base import BaseEstimator, RegressorMixin, TransformerMixin
from sklearn.cluster import KMeans
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler


class MeanBaseline(BaseEstimator, RegressorMixin):
    """Predict the training mean for everyone.

    A constant score carries no ranking information at all, so its
    capture would be decided entirely by whatever order the rows arrive
    in. Rather than let row order masquerade as a result, this baseline
    adds a seeded, financially negligible jitter: the dollar metrics are
    unchanged to the cent, and the ranking becomes explicitly random --
    which is what the capture floor actually is."""

    def __init__(self, random_state=42):
        self.random_state = random_state

    def fit(self, X, y):
        self.mean_ = float(np.mean(y))
        return self

    def predict(self, X):
        rng = np.random.default_rng(self.random_state)
        return self.mean_ + rng.normal(0.0, 1e-9, len(X))


class LastValueBaseline(BaseEstimator, RegressorMixin):
    """The incumbent, expressed as an estimator: predict that the trailing
    six months repeat. It fits nothing, so it can never leak anything."""

    def fit(self, X, y=None):
        return self

    def predict(self, X):
        return np.asarray(X, dtype=float).ravel()
```

Expected output: nothing — this cell defines two classes.

Input: none. Transformation: two small estimator definitions. Output: objects a cross-validation routine can fit, score, and refit per fold exactly as it does a regression. Read them rather than rewriting them.

LastValueBaseline is the incumbent expressed as an estimator: its fit method does nothing, which is the honest characterization of a rule that learns nothing, and it is therefore structurally incapable of leaking. Expressing the baseline this way is not decoration — it is what allows the same cross-validation call, on the same folds, with the same metrics, to produce the baseline's row of the leaderboard, which is the same-split clause of Section 8.9 enforced by construction rather than by discipline.

MeanBaseline carries the correction Section 8.7 argued for, and its docstring is the lesson. Predicting one number for everyone is a perfectly good calibration baseline and an empty ranking: with every score identical, which customers land in the funded tenth is decided by the order the rows happen to be in, and a sorting routine will report that accident as a capture figure without complaint. The baseline therefore breaks its own ties on a seeded draw of a billionth of a dollar — too small to move MAE or RMSE by a cent, large enough to make the ranking explicitly random. What the capture column then reports is the honest floor: what a program would capture by choosing customers at random, which is the selection share itself.

Code 8.9. Express the clustering as a pipeline step

```
class IntensityClusters(BaseEstimator, TransformerMixin):
    """Behavioral clustering as a pipeline step.

    Inside a pipeline the scaler and the centroids are refitted on each
    training fold and only ASSIGN the customers being scored, which is
    what Table 8.2's split-contamination row requires of a learned
    feature. Clusters are relabeled by ascending revenue_per_month so the
    indicator columns are comparable across folds. That ordering rule is
    what makes them comparable; it does NOT make a given rank a named
    Chapter 6 segment, and the prose below says why it cannot."""

    def __init__(self, k=4, random_state=42):
        self.k = k
        self.random_state = random_state

    def fit(self, X, y=None):
        self.scaler_ = StandardScaler().fit(X)
        self.kmeans_ = KMeans(n_clusters=self.k, n_init=10,
                              random_state=self.random_state)
        self.kmeans_.fit(self.scaler_.transform(X))
        centers = pd.DataFrame(
            self.scaler_.inverse_transform(self.kmeans_.cluster_centers_),
            columns=list(X.columns))
        order = np.argsort(centers["revenue_per_month"].to_numpy())
        self.rank_ = np.empty(self.k, dtype=int)
        self.rank_[order] = np.arange(self.k)
        return self

    def transform(self, X):
        labels = self.rank_[
            self.kmeans_.predict(self.scaler_.transform(X))]
        # k - 1 indicators, lowest-intensity cluster as the reference.
        return np.column_stack([(labels == j).astype(float)
                                for j in range(1, self.k)])
```

Expected output: nothing — this cell defines one class.

Input: none. Transformation: one transformer definition. Output: a learned feature that a cross-validation routine refits per fold, assigning rather than learning from the customers it scores.

Its one non-obvious line is the relabeling, and it repays attention: k-means cluster numbers are arbitrary integers that change between fits, so an indicator column built from raw labels

would mean a different thing in every fold — a bug that produces no error message and quietly destroys the candidate's estimate. Sorting clusters by ascending revenue per month is a declared labeling rule, and declaring it is what makes the columns comparable across folds.

It is worth being exact about what that ordering rule does not buy, because the temptation is to read these indicators as Chapter 6's segments returning. They are not. Chapter 6's four segments were named from full centroid profiles and validated against designed structure; these clusters are ranked on a single dimension, monthly revenue, and the second-lowest-intensity cluster in one fold can differ materially in discount behavior, channel mix, or basket shape from the second-lowest-intensity cluster in another. The rank is comparable; the customer type behind it is not guaranteed to be. Carrying the actual Chapter 6 segments into a predictive frame would require a different object: a frozen assignment rule, established before this study and applied unchanged to every fold, which is the third repair Section 8.9 listed. This lab takes the first, and names the candidate accordingly.

Code 8.10. Register the candidates and declare the selection rule

```
# Chapter 7's interpretable predictors, carried over unchanged.
EXPLAIN = ["recency_days", "tenure_days", "app_user",
           "email_opt_in", "discount_share"]
CLUSTER_INPUTS = ["recency_days", "frequency", "monetary",
                  "orders_per_month", "revenue_per_month", "aov",
                  "discount_share", "store_share"]
EXPLAIN_COLUMNS = sorted(set(EXPLAIN) | set(CLUSTER_INPUTS))

# A specification chosen for prediction: everything the snapshot
# legitimately knows, including the incumbent rule's own column.
PREDICT = ["spend_prior6m", "monetary", "frequency", "aov", "recency_days",
           "tenure_days", "orders_per_month", "revenue_per_month",
           "discount_share", "store_share", "occasionwear_share",
           "app_user", "email_opt_in"]

explanatory_style = Pipeline([
    ("features", ColumnTransformer([
        ("numeric", "passthrough", EXPLAIN),
        ("clusters", IntensityClusters(), CLUSTER_INPUTS)])),
    ("model", LinearRegression())])

prediction_oriented = make_pipeline(StandardScaler(), LinearRegression())

flexible = make_pipeline(StandardScaler(),
                         PolynomialFeatures(degree=2, include_bias=False),
                         LinearRegression())

INCUMBENT = "last-value baseline"
CANDIDATES = {
    "mean baseline":        (MeanBaseline(),      ["spend_prior6m"]),
    INCUMBENT:              (LastValueBaseline(), ["spend_prior6m"]),
    "explanatory-style + clusters": (explanatory_style, EXPLAIN_COLUMNS),
    "prediction-oriented":  (prediction_oriented, PREDICT),
    "prediction-oriented, degree 2": (flexible,   PREDICT),
}

# THE SELECTION RULE, declared here -- before a single candidate is
# fitted -- and executed unedited in Code 8.13.
MIN_CAPTURE_MARGIN = 0.02   # capture points a candidate must add
MIN_FOLDS_WON      = 4      # of five, on paired fold differences
SIMPLICITY_ORDER   = ["mean baseline", INCUMBENT, "prediction-oriented",
                      "explanatory-style + clusters",
                      "prediction-oriented, degree 2"]

# The cheapest leakage guard in the chapter: no candidate sees the label.
for name, (_, columns) in CANDIDATES.items():
    assert set(columns) <= set(train_df.columns), name
    assert "spend_next6m" not in columns, f"{name} was handed the label"
assert set(SIMPLICITY_ORDER) == set(CANDIDATES)
print("candidates registered:", len(CANDIDATES))
```

Expected output: the message that five candidates were registered, after two structural assertions confirm that every candidate's column list exists in the training frame and that none of them contains the label.

Input: the training frame's column names. Transformation: three pipelines and one registry. Output: a dictionary pairing each candidate with the columns it is allowed to see — which is also, read the other way, a written record of what each candidate is forbidden to see.

Note where the learned steps sit. The explanation-oriented candidate's segmentation is a step inside its pipeline, not a column computed beforehand; the two prediction-oriented candidates standardize inside their pipelines rather than on the table. Neither choice changes a single fitted value when a model is fitted once on everything. Both change the cross-validated estimate, because a pipeline is refitted per fold and a precomputed column is not.

The final assertion is the cheapest leakage guard in the chapter and the one most worth copying into every future project: no candidate's feature list may contain the label. It would not catch a subtle arithmetic relative, which is what the derivation trace is for, but it catches the version that actually happens under deadline — a column list built by selecting everything numeric.

Code 8.11. Build the training-only cross-validation leaderboard

```
from sklearn.metrics import mean_absolute_error, root_mean_squared_error
from sklearn.model_selection import (KFold, cross_val_predict,
                                     cross_validate)


def capture(y_true, y_pred, ids, share=None):
    """Share of actual outcome spend held by the customers a score ranks
    inside the program's capacity. One function, used by the folds, by
    the test set, and by the campaign list -- so all three apply the
    identical ranking rule, ties included."""
    share = SELECTION_SHARE if share is None else share
    ranked = (pd.DataFrame({"actual": np.asarray(y_true, dtype=float),
                            "pred": np.asarray(y_pred, dtype=float)},
                           index=ids)
              .rename_axis("customer_id").reset_index()
              .sort_values(["pred", "customer_id"],
                           ascending=[False, True]))
    k = max(1, round(len(ranked) * share))
    total = ranked["actual"].sum()
    if total <= 0:
        return np.nan
    return ranked.head(k)["actual"].sum() / total


def capture_scorer(estimator, X, y):
    """A scorer callable rather than a make_scorer wrapper: the
    tiebreak needs X's customer index, which a metric never sees."""
    return capture(y, estimator.predict(X), X.index)


SCORING = {"capture": capture_scorer,
           "mae": "neg_mean_absolute_error",
           "rmse": "neg_root_mean_squared_error"}

# One fold generator, built once, shared by every candidate: same split.
CV = KFold(n_splits=5, shuffle=True, random_state=42)

rows, fold_capture = [], {}
for name, (estimator, columns) in CANDIDATES.items():
    X = train_df[columns]
    scores = cross_validate(estimator, X, y_tr, cv=CV, scoring=SCORING)
    fold_capture[name] = scores["test_capture"]
    # Pooled out-of-fold predictions give ONE ranked list over all
    # training customers, which is the shape a campaign actually takes.
    oof = cross_val_predict(estimator, X, y_tr, cv=CV)
    rows.append({"candidate": name,
                 "cv_capture": scores["test_capture"].mean(),
                 "capture_sd": scores["test_capture"].std(),
                 "oof_capture": capture(y_tr, oof, X.index),
                 "cv_mae": -scores["test_mae"].mean(),
                 "cv_rmse": -scores["test_rmse"].mean()})

leaderboard = pd.DataFrame(rows).set_index("candidate")
print(leaderboard.round(4).to_string())
```

Expected output: one table with five rows and seven columns: cross-validated capture with its fold-to-fold standard deviation, cross-validated MAE with its standard deviation, cross-validated RMSE, and each candidate's capture and MAE margins over the incumbent.

Input: the training customers only. Transformation: five-fold cross-validation of every candidate on one shared fold generator, scored on all three declared metrics. Output: the leaderboard, and the only evidence the analyst is permitted to use when choosing a finalist.

The custom scorer is where Section 8.6's argument becomes executable, and its shape is deliberate. Capture is not a built-in metric because it is not a property of the errors; it is a property of the ranking, computed by sorting the fold's customers by predicted score, taking as many as the capacity share funds, and asking what share of that fold's actual outcome spend those customers held. That is why it is written as a scorer callable taking the estimator, the features, and the labels, rather than as a metric function wrapped by make_scorer: a metric function receives two arrays of numbers and never sees who the customers are, so it cannot break ties on customer identity. The campaign list does break ties on identity, and a leaderboard that ranks by a different rule than the list it is choosing has quietly changed the question. One capture function, used by the folds, by the test set, and by the final list, is the fix.

Two capture columns appear, and they answer different questions. The cross-validated figure averages five fold-level rankings: it estimates how well the model orders a population of that size, and its spread across folds is the reliability signal. The out-of-fold figure pools one held-out prediction per training customer into a single ranked list, which is the shape a campaign actually takes. For a metric that cannot be decomposed customer by customer — and capture, being a property of a cutoff, cannot be — the two need not agree, and a wide gap between them is itself informative about how much the ranking depends on the size of the pool being ranked.

The margin columns are the deliverable's sentences in numerical form — a model that beats the mean baseline handsomely and the incumbent barely has learned something and earned little, and only the second comparison is a budget argument. The per-fold scores are stored rather than discarded, because Code 8.13 needs them.

Read the standard-deviation columns as seriously as the means, because they are the variance half of Section 8.8 made visible. A candidate whose capture swings widely across folds is telling you its performance depends on which whales it drew, and a margin smaller than the fold-to-fold spread of the metric it is measured on is not yet a finding. The designed structure of the base fixes the pattern your predictions should anticipate: the mean baseline's capture should land near a tenth, since a constant score produces an arbitrary ranking; the incumbent should beat it comfortably, because trailing spend genuinely carries signal; and the models should beat the incumbent by a margin that is real and considerably smaller than anyone hoped, because most of what makes a customer valuable next season is visible in what they did last season.

VERIFICATION CHECK

Before you run: write four predictions. Which candidate wins on capture, and which on MAE — and state explicitly what you will do if they disagree. How far will the mean baseline's capture sit from one tenth, and why can it not sit far from it? Will the explanatory-style candidate beat the prediction-oriented one, and what would it mean about Chapter 7's variable selection if it did? Will the degree-2 candidate beat its linear parent, and by more than the fold-to-fold spread?

After you run: grade every prediction, then run the three comparison clauses of Section 8.9 as an audit of your own table. Same split: confirm that one fold generator produced every row — the code shares one CV object, and the point of reading the cell is to see that it does. Same metric: the finalist will be chosen on the declared decision metric by the rule in Code 8.13, and if a candidate wins on capture while losing on MAE, that is a real finding to report rather than a licence to switch metrics. Write the sentence it forces: this candidate orders customers better and prices them worse, which is acceptable for a fixed-capacity list and unacceptable if anyone later uses the score as a dollar estimate — so the permitted-use note of Section 8.15 acquires a clause. Same baseline: state each model's margin over the incumbent in both metrics, and say whether it exceeds the fold-to-fold spread.

Investigate if: a baseline wins. That is a legitimate outcome, not a failed lab, and it has a specific meaning: on this frame, with these features, at this horizon, the sorted spreadsheet is the honest answer, and the deliverable should say so and redirect the effort to framing — a shorter horizon, a different target, or a treatment question an experiment could settle. Investigate also if any model's capture exceeds roughly half, or its MAE falls to a small fraction of the incumbent's: against a base where a quarter of customers spend nothing and spend is concentrated in a small minority, that is the too-good signature of Section 8.4, and the feature trace comes before the celebration.

8.12.4 Lab 8.2, Part B: Manufacturing Overfitting Inside the Training Data

Part B makes the U-shape of Section 8.8 visible on StyleCraft's own data by turning the one dial this chapter's machinery offers: flexibility via polynomial features. The design of this demonstration is itself a lesson, because the obvious version of it violates the chapter's own rule. Fitting nine degrees and reading the test error at each one would let the test set choose the model's complexity — development steered by the sealed data, the exact practice Section 8.5 forbids and the practice that makes a final test figure meaningless. The dial is therefore turned entirely inside the training customers, with cross-validation supplying the held-out estimate at every degree, and the test set stays sealed until Part C names a finalist and opens it.

The second design choice is scale. Expanding all thirteen prediction-oriented features to the ninth degree would generate an unreadable number of terms and demonstrate mostly that a laptop can run out of patience. Two predictors — recency and tenure, both continuous, both genuinely related to future spending — expand to a few dozen terms at the top degree, which is enough to overfit visibly and few enough to interpret.

Code 8.12. Demonstrate overfitting inside the training data

```
# Two predictors, not the whole table. With eight inputs, degree 9 expands
# to more than twenty-four thousand terms before the regression is fitted;
# with two, it expands to fifty-four, which is a demonstration rather than
# a computation (scikit-learn developers, 2026c).
POLY_FEATURES = ["recency_days", "tenure_days"]

rows = []
for degree in range(1, 10):
    pipe = make_pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=degree, include_bias=False),
        LinearRegression())
    scores = cross_validate(pipe, train_df[POLY_FEATURES], y_tr, cv=CV,
                            scoring=SCORING, return_train_score=True)
    pipe.fit(train_df[POLY_FEATURES], y_tr)
    rows.append({
        "degree": degree,
        "terms": pipe.named_steps["polynomialfeatures"].n_output_features_,
        "train_rmse": -scores["train_rmse"].mean(),
        "train_mae": -scores["train_mae"].mean(),
        "cv_mae": -scores["test_mae"].mean(),
        "cv_mae_sd": scores["test_mae"].std(),
        "cv_capture": scores["test_capture"].mean(),
    })

flex = pd.DataFrame(rows).set_index("degree")
print(flex.round(3).to_string())

# Least squares minimizes SQUARED error, so training RMSE cannot rise as
# nested terms are added. Training MAE carries no such guarantee.
assert (flex["train_rmse"].diff().dropna() <= 1e-6).all()
mae_rises = int((flex["train_mae"].diff().dropna() > 0).sum())
print(f"\ndegrees at which training MAE rose anyway: {mae_rises}")
print(f"best cross-validated degree on MAE: {flex['cv_mae'].idxmin()}")
print(f"worst cross-validated degree on MAE: {flex['cv_mae'].idxmax()}")
```

Expected output: a nine-row table carrying, per degree, the number of polynomial terms, the training RMSE and MAE, the cross-validated MAE with its fold-to-fold spread, and the cross-validated capture; then an assertion on the monotone training RMSE, a count of degrees at which training MAE rose anyway, and the best and worst cross-validated degrees.

Input: two training-set predictors. Transformation: nine pipelines of increasing flexibility, each cross-validated on the shared folds. Output: the two curves of Section 8.8, printed side by side, with the test set untouched.

Read the columns in the order they were argued. The term count rises steeply, which is the whole meaning of flexibility. Training RMSE falls, or at worst holds, at every step, and the assertion says why that is guaranteed rather than observed: least squares minimizes squared error, and a richer nested model can always reproduce a poorer one's fit. Training MAE is printed beside it precisely because it carries no such guarantee — the fitting criterion protects squared error, not absolute error, so a richer model can trade a few cents of typical miss for a large

reduction in a handful of squared ones. The cell counts the degrees at which that happens rather than asserting it cannot.

The cross-validated column is the one that decides anything, and it should be read as a pattern to investigate rather than a shape guaranteed to appear. The expected behavior is a fall, a turn, and a climb, with the turn marking the honest amount of flexibility for this sample size. What is not guaranteed is a clean U in one finite sample, or that the highest degree is the worst performer: fold-to-fold noise is real, the printed spread column measures it, and a difference smaller than that spread is not a difference. Name the two sides with Section 8.8's vocabulary, locate the turn, and state whether the distance between the best and worst degrees exceeds the fold-to-fold noise.

VERIFICATION CHECK

Before you run: predict the three columns' shapes. Training RMSE: state the direction and say whether it is guaranteed or merely expected, and why. Training MAE: same two questions, and a different answer. Cross-validated MAE: predict where the turn falls, and write down what you will conclude if there is no visible turn at all within nine degrees.

After you run: check the guaranteed claim against the assertion, the unguaranteed one against the printed count, and the U-shape against the fold-to-fold spread. Then answer the leakage question this cell exists to pose, which the draft of this chapter got wrong and which is worth getting right in writing. Suppose you had expanded the polynomial features on the pooled data before splitting. Would that be split contamination? The answer is no, and the reason is the estimation test of Section 8.4: PolynomialFeatures computes predetermined powers and products, so fitting it mainly establishes how many output columns there will be and what to call them; it estimates nothing from the values, and expanding before or after the split produces identical numbers for every row (scikit-learn developers, 2026c). Now ask the same question about the first step in the same pipeline. StandardScaler estimates a mean and a standard deviation from the data it is fitted to, so fitting it on the pooled table before splitting would let every validation customer contribute to the centering and scaling applied to the training customers — split contamination, Table 8.2's fourth row, committed by the blandest step in the pipeline. Write both answers down with the reason attached, because the distinction between a learned step and a fixed transformation is what the rule actually is, and “do everything after the split” is the rule of thumb that people remember and then apply to the wrong things.

Investigate if: cross-validated MAE improves monotonically across all nine degrees. That is not the designed result and usually means the flexibility dial is not being turned far enough to hurt on this sample size — eight thousand customers tolerate more flexibility than eight hundred, exactly as Section 8.8 said. Extend the range, or add a third predictor, and report what you changed.

8.12.5 Lab 8.2, Part C: The Selection Rule and the Sealed Test Set

Everything so far happened inside the training customers. Part C ends development by executing the selection rule, and only then spends the test set — once, on the candidate the rule names.

The rule matters because the obvious alternative is wrong in a way that is easy to miss. Taking whichever candidate posts the highest cross-validated capture would name a winner even when its advantage over the incumbent is a fraction of a percentage point, even when the advantage appears in two folds out of five, and even when a materially simpler candidate performs indistinguishably. That is the leaderboard being read as a scoreboard rather than as evidence, and it contradicts this chapter's own warning that a margin smaller than the fold-to-fold spread is not yet a finding.

So the rule is written into Code 8.10, before any candidate is fitted, and Code 8.13 executes it unedited. It has four clauses: a candidate must add at least a stated number of capture points over the incumbent; it must win at least four of the five folds; among the candidates that clear both, the simplest wins, on a complexity ordering declared in the same cell; and if nothing clears the bar, the incumbent is retained. The threshold is a business judgment rather than a statistical one — two capture points on this program is roughly the margin at which the model's list is worth the operational change of adopting it — and writing it down in advance is what stops it from becoming whatever the results happen to justify.

The comparison is paired, and the reason is the one Section 8.9 gave. Folds differ in difficulty: a fold holding three whales is hard for every candidate at once. Comparing a candidate's mean capture against its own standard deviation across folds therefore measures mostly fold difficulty, which every candidate shares and which the comparison should cancel. Subtracting the incumbent's score fold by fold cancels it, and the spread of those differences is the honest measure of whether the candidate reliably wins or merely averages well.

Code 8.13. Apply the declared selection rule

```
# The rule declared in Code 8.10, executed unedited. Because every
# candidate ran on the same folds, the informative quantity is the PAIRED
# per-fold difference against the incumbent, not each candidate's own
# standard deviation: a fold that is hard for one model is usually hard
# for all of them, and pairing removes that shared difficulty.
base = fold_capture[INCUMBENT]

rows = []
for name, scores in fold_capture.items():
    if name == INCUMBENT:
        continue
    diff = scores - base
    clears = bool(diff.mean() >= MIN_CAPTURE_MARGIN
                  and (diff > 0).sum() >= MIN_FOLDS_WON)
    rows.append({"candidate": name,
                 "mean paired margin": diff.mean(),
                 "sd of paired margins": diff.std(),
                 "folds won": int((diff > 0).sum()),
                 "clears the rule": clears})

paired = pd.DataFrame(rows).set_index("candidate")
print(paired.round(4).to_string())

qualified = [n for n in SIMPLICITY_ORDER
             if n in paired.index and paired.loc[n, "clears the rule"]]
FINALIST = qualified[0] if qualified else INCUMBENT

print(f"\nrule: at least {MIN_CAPTURE_MARGIN:.2f} capture over the "
      f"incumbent and at least {MIN_FOLDS_WON} of 5 folds won")
print("qualifying, simplest first:", qualified or "none")
print("FINALIST:", FINALIST)
```

Expected output: a four-row table of paired margins — mean, spread, folds won, and whether the rule is cleared — then the rule restated, the qualifying candidates in simplicity order, and the finalist.

Input: the per-fold capture scores stored by Code 8.11. Transformation: subtraction, fold by fold, against the incumbent. Output: a named finalist, arrived at by a rule rather than by a preference.

Read the paired-margin table against the leaderboard and notice how differently the two read. On the leaderboard, a candidate's capture standard deviation is often larger than its margin over the incumbent, which invites the conclusion that nothing is distinguishable. The paired column usually tells the opposite story, and correctly: once shared fold difficulty is subtracted out, a candidate that wins every fold by a consistent amount is showing exactly the reliability the raw spread concealed. This is the same reasoning that makes a paired comparison stronger than two independent ones anywhere else in statistics, and it is available here only because the frame insisted on one fold generator for every candidate.

Note also what the simplicity clause does when several candidates qualify. It selects the simplest, not the highest — so a flexible candidate that edges out its linear parent by less than the declared margin does not win, and the deliverable carries the model whose behavior is easiest to

explain, monitor, and retrain. Section 8.8's bias–variance argument said flexibility must earn its place; the rule is what makes that sentence enforceable. And if no candidate clears the bar, the finalist is the incumbent, which is a legitimate and reportable outcome rather than a failed lab.

Code 8.14. Open the sealed test set, once

```
# The finalist was named by Code 8.12 from training folds alone. Refit it
# on ALL training customers -- the cross-validated estimate came from
# models fitted on four folds at a time -- and open the test set once.
final_model, final_columns = CANDIDATES[FINALIST]
final_model.fit(train_df[final_columns], y_tr)

pred_model = final_model.predict(test_df[final_columns])
pred_mean  = np.repeat(y_tr.mean(), len(y_te))   # trained mean, not test
pred_lv    = test_df["spend_prior6m"].to_numpy()

results = []
for name, pred in [("mean baseline", pred_mean),
                   (f"{INCUMBENT} (incumbent)", pred_lv),
                   (f"selected: {FINALIST}", pred_model)]:
    results.append({"rule": name,
                    "capture": capture(y_te, pred, test_df.index),
                    "MAE": mean_absolute_error(y_te, pred),
                    "RMSE": root_mean_squared_error(y_te, pred)})
print(pd.DataFrame(results).set_index("rule").round(3).to_string())

# NOW the halves may be compared: with the grade already recorded, this
# is context for reading it rather than knowledge that could steer it.
halves = pd.DataFrame(
    {"train": [y_tr.eq(0).mean(), y_tr.mean(), y_tr.quantile(0.99)],
     "test":  [y_te.eq(0).mean(), y_te.mean(), y_te.quantile(0.99)]},
    index=["zero-label share", "mean", "99th percentile"])
print("\n" + halves.round(3).to_string())

# An unconstrained linear model can predict negative spend. Count it in
# the open; do not pick a remedy by watching this number move.
negatives = int((pred_model < 0).sum())
print(f"\nnegative test predictions: {negatives} of {len(pred_model)} "
      f"({negatives / len(pred_model):.1%})")
```

Expected output: the name of the selected candidate, then a three-row table giving capture, MAE, and RMSE for the mean baseline, the incumbent, and the model on the same held-out customers, and finally the count and size of any negative predictions.

Input: the training customers, the sealed test customers, and the leaderboard's verdict. Transformation: one refit and three sets of predictions. Output: the deliverable's central table.

Two details enforce disciplines the chapter argued for. The finalist arrives as a variable set by Code 8.13, not as a name typed after seeing test results — which makes the order of operations auditable in the notebook itself rather than asserted in a methods paragraph. And the mean baseline's prediction is the training mean repeated, not the test mean: computing it from the test labels would hand the baseline a summary of the answers and turn the model's opponent into a cheat. That is the smallest possible instance of Section 8.5's rule that anything learned from data

is learned from training data, and it is worth pointing at because it is the version students most often get wrong while getting everything larger right.

The half-comparison table appears here rather than at the split, and its position is the argument. Read before selection, those numbers steer development; read after the grade is recorded, they explain it — a test half holding fewer whales makes a strong capture figure slightly less impressive, and one holding more makes a weak MAE slightly more forgivable. Same numbers, different epistemic status, decided entirely by when they were looked at.

The negative-prediction line is an honest disclosure rather than a repair. An unconstrained linear model can return a negative predicted spend for a customer whose features sit at the low end of every input, and the number is nonsense as a dollar estimate even when the ranking it implies is perfectly sensible. Four remedies exist: keep them and document that the score is used only for ranking, which is what this program does; clip at zero; model a transformed response; or use a model family that cannot go negative. Three of the four are modeling choices, and the rule that governs them is the rule that governed everything else — a remedy is chosen by cross-validation on the training customers, never by trying each one against the test set and keeping whichever looked best.

VERIFICATION CHECK

Before you run: write four predictions. Will the model beat both baselines on capture, and on MAE? How close will the test figures sit to the cross-validated ones, and in which direction do you expect them to differ? What would an alarming result look like in each direction — too poor, and too good? And what will you do if the test capture disagrees materially with the cross-validated capture?

After you run: place the result on Table 8.4's signature grid and write the two sentences that survive into Section 8.13's deliverable: the capture sentence, which prices the model in the decision's own currency, and the margin sentence in dollars, which prices its typical miss against the incumbent's. Then honor the protocol: the test set is now spent. If the result prompts a new idea — another feature, another family, a clipping rule — that idea goes back to the training customers and the cross-validated leaderboard, and any future test figure is reported as what it would then be, a second look rather than a clean grade.

Investigate if: the test figures sit far better than the cross-validated ones. A modest gap in either direction is ordinary sampling variation, and the fold-to-fold spread from Code 8.10 is the yardstick for what counts as modest. A large gap in the model's favor is the too-good signature again, and the most common cause is not exotic: a column that entered the feature list after the leaderboard was frozen, or a test frame rebuilt at some point with a later snapshot. Both are found by rerunning Code 8.4 and Code 8.5 from the top and confirming that the row counts and the recency minimum are unchanged.

8.12.6 Lab 8.2, Part D: The Deliberate Leak, Caught

Part D commits this chapter's canonical error on purpose, because a leak you have built is a leak you will recognize. Rebuild the modeling table the lazy way — the way an unframed prompt or a

hurried analyst would — by pointing the same feature builder at the June 30, 2026 analysis date, so that every column is computed across the full twenty-four months, outcome window included. Nothing else changes: same customers, same labels, same split, same features, same estimator. One argument moves, and the model's measured performance improves dramatically. That is the entire anatomy of the error, and the lab works to keep it that clean: the honest side of the comparison is a fresh copy of one fixed specification rather than whichever candidate the selection rule happened to name, because a demonstration that changed the model family and the feature table at the same time would prove nothing about leakage.

Code 8.15. Manufacture and detect temporal leakage

```
from sklearn.base import clone

# The lazy build: the June 30, 2026 feature table, whose every column was
# computed across the full twenty-four months -- outcome window included.
cf_full = build_customer_features(transactions_clean, customers, products,
                                  snapshot=pd.Timestamp("2026-06-30"),
                                  window_start=FEATURE_START)

leaky_df = (cf_full.loc[model_df.index]
            .join(labels).join(prior6)
            .fillna({"spend_next6m": 0.0, "spend_prior6m": 0.0}))

# The SAME CUSTOMERS, not merely the same seed. A seed reproduces
# positions, not identities, so a re-split could grade the two pipelines
# on different people -- which Section 8.9's same-split clause forbids.
lk_train, lk_test = leaky_df.loc[train_ids], leaky_df.loc[test_ids]
assert lk_train.index.equals(train_df.index)
assert lk_test.index.equals(test_df.index)
assert np.allclose(lk_test["spend_next6m"], y_te)

# ONE specification, fitted twice. clone() returns an unfitted estimator
# with identical parameters, so the only difference between these two
# models is which feature table they were shown. Using the selected
# finalist here would confound leakage with a change of model family.
honest_demo = clone(prediction_oriented).fit(train_df[PREDICT], y_tr)
leaky_demo  = clone(prediction_oriented).fit(lk_train[PREDICT],
                                             lk_train["spend_next6m"])

for tag, pred in [("honest", honest_demo.predict(test_df[PREDICT])),
                  ("leaky ", leaky_demo.predict(lk_test[PREDICT]))]:
    print(f"{tag} model  test MAE: "
          f"{mean_absolute_error(y_te, pred):8.2f}"
          f"   capture: {capture(y_te, pred, test_df.index):6.1%}")

# The audit line that convicts the column with no modeling at all: at a
# December 31 snapshot, a correct recency cannot be below 181 days.
impossible = int((cf_full["recency_days"] < 181).sum())
print(f"\nrange check -- customers whose recency is impossible at the "
      f"declared snapshot: {impossible:,} of {len(cf_full):,}")
```

Expected output: the honest and leaky models' test MAE and capture on the same customers, with the leaky figures decisively better, then the count of customers whose recency is arithmetically impossible at the declared snapshot; three assertions pass silently.

Input: the certified files, the existing labels, and the frozen train and test identities. Transformation: one feature table built at the wrong date, one model fitted on it. Output: the sensation of a model that is too good, produced on demand, and the audit line that convicts it.

clone() is the instrument that holds the specification fixed. It returns a new, unfitted estimator carrying identical parameters, so the honest and the leaky model differ in exactly one respect: which feature table they were shown. Using the selected finalist for the honest side would have been the natural shortcut and a confounded comparison — had the rule named a baseline, the cell would have been contrasting a last-value rule with a leaky regression and attributing the whole gap to leakage.

The three assertions are the cell's methodological spine, and the first two answer a question the draft of this chapter left open. Re-splitting the leaky frame with the same random seed would not guarantee the same customers on each side, because a seed reproduces positions rather than identities and any difference in row order sends different people to the test half. Selecting by the stored index does guarantee it, and the assertions prove it — after which the two pipelines differ in exactly one respect and the comparison means what it claims. The third assertion confirms that the labels themselves were not disturbed by the rebuild, which is what makes the two MAE figures comparable at all.

State the mechanism in writing before reading the numbers. The full-window table's recency was measured to June 2026, so a customer who bought in May has a small recency that is only possible because of an outcome-window order; its monetary sums feature-window spend plus the label itself, which is Table 8.2's arithmetic-leakage row; its frequency and its per-month rates absorb outcome-window orders the same way. The model is not predicting the future. It is reading it, and reporting how well the future predicts itself.

Then catch it the way an auditor would who had not built it. The range check is the instrument, and it needs no model at all: at a December 31, 2025 snapshot, a correctly computed recency cannot be smaller than 181 days, because a smaller value would require a purchase after the snapshot. The printed count of impossible values is the smoking gun — thousands of them, produced by a single wrong argument. Close with the sentence the whole chapter has been building toward, and keep it for the vendor meeting: this leaky model's excellent score is exactly what a near-perfect accuracy claim looks like from the inside, and note that it would survive a train/test split, a fixed seed, a cross-validated leaderboard, and a tidy comparison table, because the leak sits in the features, where no split can see it. Splits catch overfitting; only audits catch leaks.

8.12.7 Lab 8.2, Part E: The List, the Money, and the AI Round Trip

Part E translates the surviving model into the decision. Score the test customers with the honest model, rank, and cut the top decile; do the same with the incumbent's trailing-spend ranking;

then compare what each list captures of the test customers' actual outcome-window spend, and count how many names actually change.

Code 8.16. Compare the two ranked lists

```
scored = test_df.assign(pred=pred_model).reset_index()
assert "customer_id" in scored.columns

# The same capacity share, the same tiebreak, the same function the folds
# used -- so the number in the deliverable is the number that was raced.
k = max(1, round(len(scored) * SELECTION_SHARE))
top_model = scored.sort_values(["pred", "customer_id"],
                               ascending=[False, True]).head(k)
top_inc = scored.sort_values(["spend_prior6m", "customer_id"],
                             ascending=[False, True]).head(k)

total = scored["spend_next6m"].sum()
assert total > 0
assert len(top_model) == len(top_inc) == k

cap_model = top_model["spend_next6m"].sum() / total
cap_inc = top_inc["spend_next6m"].sum() / total
overlap = len(set(top_model["customer_id"]) & set(top_inc["customer_id"]))
assert np.isclose(cap_model, capture(scored["spend_next6m"],
                                     scored["pred"],
                                     scored["customer_id"]))

print(f"selection share {SELECTION_SHARE:.1%} -> {k} slots on this "
      f"test population ({CAPACITY} in deployment)")
print(f"model list captures:     {cap_model:6.1%}")
print(f"incumbent list captures: {cap_inc:6.1%}")
print(f"capture gap:             {cap_model - cap_inc:+6.1%}"
      f"  =  ${(cap_model - cap_inc) * total:,.2f} of outcome spend")
print(f"customers on both lists: {overlap} of {k}"
      f"  ({k - overlap} names change)")
```

Expected output: the number of slots on the test population, the capture of each list, the capture gap expressed both in percentage points and in dollars, and the overlap count with the number of names that change; three assertions pass silently.

Input: the sealed test customers and their scores. Transformation: two deterministic sorts and one arithmetic comparison. Output: the exhibit the meeting is actually deciding on.

The cutoff is the selection share computed in Code 8.6, not a typed tenth, so the number of slots on the test population is the program's capacity scaled to the population being ranked. The final assertion re-derives the same figure through the capture function the folds used, which is a small proof that the leaderboard and the campaign list are measuring one thing rather than two things with one name.

The sorts carry a declared secondary key, and the reason is operational rather than statistical. Predicted values tie, trailing-spend values tie far more often — every customer with no purchases in the trailing window ties at zero — and a sort with no tiebreaker returns whichever order the frame happened to be in. The eight hundredth name would then differ between the

analysis notebook and the campaign build, which is the kind of discrepancy that destroys trust in a program for reasons no one can reconstruct a month later. Sorting by score and then by customer identifier makes the list reproducible by anyone, on any machine, in any row order.

Read the three outputs as three different arguments. The capture gap in percentage points is the model's ordering quality. The same gap in dollars is the budget argument, and it is the number the deliverable leads with. The overlap count is the change-management argument, and it is the one the CRM manager will ask about first: a model that reorders the list without changing many names is buying a small improvement at the cost of an operational change, while a model that replaces a large fraction of the list is making a substantial claim about who the program has been missing — and the customers it adds should be recognizably the lumpy occasion profile the miniature predicted.

Close the lab with two exercises in judgment. First, the expected-value sketch of Section 8.10, one paragraph of arithmetic: at twenty dollars per slot, what would the capture gap have to be worth in incremental margin for the model to out-earn the spreadsheet — and underline the word incremental, with two sentences on why this lab cannot certify it. The model predicts spend, not spend caused by gifting; the capture gap says that the model's eight hundred customers spent more than the spreadsheet's eight hundred, and says nothing about how much either group would have spent untreated. Section 7.13's discipline names the error and Chapter 11's instrument is the only thing that closes it.

Second, the AI round trip. Hand an assistant the raw certified tables and the deliberately loose prompt “build me the best possible model to predict customer spend — maximize accuracy,” then run Table 8.5's five points on what comes back, in order, in writing. The designed likelihood — engineered by the loose prompt itself — is a pipeline that fails point 1 (no frame, no snapshot, no eligibility rule), point 2 (full-window features, exactly Code 8.13's leak, arrived at innocently), point 3 (complexity chosen against a test split, preprocessing fitted on pooled data), and point 4 (no baseline), while reporting a fit statistic the assistant celebrates. Your audit memo, filed per Appendix D, is the lab's final deliverable and the direct rehearsal of Section 8.13's vendor verdict. Deliverables for both labs: the notebook with all fourteen cells executed, the written predictions made before each cell was run, the reconciled population counts from Code 8.5, the frozen leaderboard from Code 8.10, the flexibility table from Code 8.11, the single test-set table from Code 8.12, the leakage audit from Code 8.13 with its range-check count, the list comparison from Code 8.14, and the AI-Use Appendix documenting every assistant exchange, including at least one delegated step you corrected and why.

8.13 Marketing Interpretation and Managerial Insight

The lab's outputs are leaderboards and ranked lists; the thirty-day deadline runs on sentences. This section translates — and, per this guide's standing practice, it does so partly by exhibiting wrong managerial readings and correcting them, because the predictive setting mints new misreadings faster than the associational one did, and three of them will be voiced in the scoring meeting by people whose only error is applying last chapter's intuitions to this chapter's numbers.

The first wrong reading will come from whoever liked the vendor deck: “Our in-house model's average miss is fifty-odd dollars. The vendor's model is at 96 percent. Ours is a C-minus student; theirs is an A. Buy the license.” The comparison is meaningless before it is even unfair, and the first problem is vocabulary rather than evidence. “Ninety-six percent accuracy” is undefined for a numeric prediction. Accuracy is a classification statistic — the share of cases assigned to the correct class — and future spend in dollars has no classes to be correct about. If the vendor means a test R^2 of 0.96, they should say so and disclose the evaluation design, because R^2 is a proportion of variance accounted for and not a percentage of predictions that were right; the two sentences sound alike and describe different quantities. It is worth knowing, and worth saying in the meeting, that R^2 computed on data a model was not fitted to is not even bounded below by zero: a model that predicts held-out cases worse than their own mean produces a negative value (scikit-learn developers, 2026d). A statistic that can go negative out of sample is not a percentage of anything.

The second problem is the one Section 8.4 armed you for. Even read charitably as $R^2 = 0.96$ on individual six-month spend, the number is implausible against everything the lab documented about this outcome: a quarter of eligible customers spend nothing, spend is concentrated in a small minority, and the honest baselines land in a known range. The corrected reading, which the deliverable should print in exactly this shape: “Our model's grade was earned under exam conditions we can show you — a declared frame, a sealed test set opened once, two baselines on the same customers. The vendor's number, as presented, was not earned under any conditions we can inspect, and the statistic it names does not exist for this kind of target. Before the two numbers can share a sentence, the vendor must answer five questions” — and the five questions are Table 8.5 read aloud: What is the frame, and what exactly does the number measure? Where is the snapshot, and can you prove no feature crosses it? What was held out, how often was it consulted, and was model complexity chosen against it? What baseline was beaten, by how much, on which metric? What is the error in dollars, on our customers, and what happens to it over time? A vendor with good answers becomes a genuine candidate — build-versus-buy is a real decision, and Section 8.14's third vignette shows the buy side winning honestly. A vendor without them is selling in-sample flattery at an annual license price.

The second wrong reading is gentler and more corrosive: “The model says C004471 will spend 585 dollars, so put her on the list — the model knows.” The model does not know. The score is a conditional average — what customers with C004471's snapshot profile spent, on average, in past outcome windows — wearing the precision of its decimal places, and the test MAE is the printed reminder that individual customers sit tens to hundreds of dollars from their scores routinely. The corrected reading changes the object of confidence from the customer to the list: the model is not trusted to be right about C004471; it is trusted to sort eight thousand customers well enough that the top eight hundred capture more of next season's spend than any competing list — which is precisely the claim the capture exhibit tests and the reason Section 8.6 made capture the decision metric. This distinction — score as verdict versus score as ordering — is the predictive sibling of Chapter 7's coefficient-is-not-a-lever box, and it changes what “the

model was wrong” even means: one customer's shortfall is noise; the list underperforming the spreadsheet is failure.

The third wrong reading is the most expensive, and it is the one the opening brief pre-empted on purpose: “The model's list captures six points more spend than the spreadsheet's, so the Backstage program will generate that much additional revenue.” It will not, and the gap between those two sentences is the difference between a prediction and a treatment effect. The capture figure says that the customers the model selected turned out to spend more than the customers the spreadsheet selected. It says nothing about how much either group would have spent without a lookbook. A program that gifts eight hundred customers who were always going to spend heavily has bought recognition, not growth, and the honest deliverable prices it that way: the model improves who receives the treatment, and only a holdout — a randomly withheld slice of the eligible list, measured six months later — can price what the treatment does. Naming that experiment in the deliverable is not a hedge. It is the analysis pricing its own upgrade, the same closing move Chapter 7's drivers deliverable made.

The deliverable that survives all three misreadings has a fixed anatomy, and the lab produced every part: the frame, one page, seven declarations and the label definition with its four dates, because it is the document every later question lands on; the leaderboard from Code 8.11, cross-validated on training customers only, with both baselines and every candidate's margin; the single test-set table from Code 8.14, with capture leading and MAE and RMSE in dollars beside it; the capture head-to-head from Code 8.16, model list against spreadsheet list, with the dollar gap and the count of names that change, because the meeting is deciding between those two artifacts and this table grades the actual choice; the expected-value sketch, with incremental flagged as the assumption the design cannot certify and the holdout test named as the instrument that could; the vendor verdict as a one-paragraph memo built on the five questions; and the monitoring plan of Section 8.10 — who checks the realized error when the outcome window matures in June, against what benchmark, which input distributions are watched in the meantime, and what number triggers a retrain — because the deliverable is not a scored list but a scored list with a maintenance contract, and only one of those survives its first drift.

8.14 Business Analytics in Practice

This section turns from the fictional case to how predictive models operate — and fail — in professional marketing organizations, where the models are commodities, the pipelines are products, and the failures that matter are almost never statistical. All three vignettes below are composites of recurring professional patterns rather than reports about a single named organization.

8.14.1 Lead Scoring and the Adoption Problem

The first vignette is lead scoring, the business-to-business world's version of this chapter's scored list. The setting recurs across enterprise software firms: marketing builds a propensity model scoring inbound leads on firmographic fit and behavioral signals, validated properly out of

sample, demonstrably better than the alphabetical-and-instinct routing it replaced — and six months later, sales representatives are ignoring the scores. The post-mortems rarely find a modeling failure. They find an adoption failure with a recurring anatomy: nobody translated the score into the representative's decision, so a lead marked 74 arrived with no instruction about what to do before lunch; nobody showed the representatives evidence that the ordering worked, so the first false positive — a high-scored lead who wasted an afternoon — confirmed every prior about ivory-tower analytics; and nobody instrumented the loop, so scores were computed, exported, and quietly unconsulted while the dashboard reported the model live.

The operational artifact that separates the teams who fix this from the teams who repeat it is unglamorous: an adoption metric, defined as precisely as any model metric and reported beside it. The working version is a contact-rate-by-score-decile report — of the leads the model placed in the top decile, what share were actually worked within the service-level window, and how does that share compare with the bottom decile? A model whose top and bottom deciles are worked at the same rate is not in production in any sense that matters, whatever the pipeline logs say. Teams that instrument this discover the gap in weeks rather than quarters, and the fix is usually not a better model but a translation layer: score bands with named actions rather than a continuous number, a weekly list rather than a field in a record, and a published back-test showing representatives what the ordering was worth on leads they remember. The practice lesson is the one Provost and Fawcett (2013) build their expected-value framing around: the model is an input to a decision system, and the system — thresholds in the user's units, feedback that earns trust, telemetry on whether scores change behavior — is what produces value. A perfectly ranked list that no one acts on has an expected value of exactly zero.

8.14.2 Silent Decay After a Pricing Change

The second vignette is silent decay, composited from a pattern retention teams describe often enough to treat as a genre. A subscription commerce firm builds a customer-value model — a spend-horizon frame much like this chapter's — and wires it into onboarding offers, service-tier routing, and paid-acquisition bidding. It works, and because it works, it stops being watched. Fourteen months later, someone notices acquisition economics sagging and pulls the thread: the model has been mis-scoring new customers for two quarters, ever since a pricing-and-packaging change restructured early purchase behavior — the model's most important input. Nothing had errored. The scores still arrived, plausible-looking, on schedule; the model was simply answering a question about a relationship that no longer held.

Note which failure this is, in the vocabulary of Section 8.10. The inputs did shift, so there was data drift. But the expensive part was concept drift: the same early-purchase pattern now implied a different lifetime value, because the packaging change altered what an early purchase meant (Gama et al., 2014). No amount of watching the input distributions alone would have caught the second, which is why the repaired practice monitors both. The operational artifact here is a monitoring view with three panels and one rule. The panels: realized error on each cohort as its outcome window matures, plotted against the benchmark the original test set

established; the distribution of the two or three most important inputs, this month against the training window; and the score distribution itself, which moves when either of the first two does. The rule is the part teams skip: a written trigger naming the number that forces a retrain — realized error exceeding the test benchmark by a stated margin for two consecutive cohorts — plus a standing clause that known structural changes, a repricing or a merchandising overhaul, force revalidation without waiting for the cadence. Sculley et al. (2015) systematized why this is engineering rather than diligence: the model is the smallest box in the system diagram. The lesson compresses to a sentence worth carrying into every deployment conversation: a model without a monitoring plan is not an asset; it is a liability with good first-quarter manners.

8.14.3 Build Versus Buy

The third vignette is build versus buy, composited from diligence processes that recur wherever marketing technology is purchased, because most working analysts will meet this chapter's machinery not as builders but as evaluators of other people's claims. The modern marketing stack sells predictions everywhere: customer-data platforms ship churn and spend scores, CRM suites ship propensity models, ad platforms ship value-based bidding — and the buy side is often right, because a vendor amortizing pipeline engineering across hundreds of clients can genuinely out-build a two-person analytics team. What separates a sound purchase from an expensive slide is the diligence, and the diligence is this chapter run as a questionnaire: the frame (what unit, target, and horizon — a “churn score” with an undisclosed churn definition is not yet a product claim, per Section 8.3's label discipline); the evaluation (out of sample on whose data, and how was complexity chosen); the baseline (better than what — scores that barely beat trailing spend are repackaging your own spreadsheet); the leakage audit (what are the features, and can impossible values be range-checked); and the maintenance contract (who monitors, who retrains, what triggers).

The artifact that converts that questionnaire from a conversation into evidence is an acceptance test written into the pilot: the vendor scores a historical snapshot of the client's own customers, chosen by the client, with the outcome window already complete but withheld from the vendor. The client then computes the metrics — capture at the client's own capacity, error in the client's own dollars — against the client's own incumbent rule, using code the client wrote. Performance on someone else's base is drift waiting to be discovered; performance on a vendor-selected sample is a sales artifact; performance on a client-held holdout is the only version that means what the contract will claim. Teams that run this report an unexpected benefit: the vendors worth buying agree to it easily, because the honest ones did the work and would rather demonstrate it than argue about it. The meeting where a vendor cannot name their own snapshot date is a license fee saved.

8.14.4 In Your First Analyst Job

In your first analyst job, these vignettes compress into one expectation: the modeling is the minority of the work. The frame conversations, the label negotiations, the leakage audits, the

baseline politics — the spreadsheet's defenders are stakeholders, not obstacles — the adoption design, the monitoring calendar, the vendor diligence: that is the job's actual shape, and the fitted model sits inside it as one well-understood component. The industry's software has made fitting nearly free. What it has not made free, and what this chapter trained, is knowing whether the number the software prints deserves to be believed — and the analysts who advance are the ones whose belief, granted or withheld, turns out to be worth something.

8.15 Ethics, Acting on Predictions About People

The Business Analytics in Practice section described organizations wiring predictions into decisions; this section examines the wiring as an ethical act, extending the guide's running discussions — data use (Section 1.13), problem framing (Section 2.12), measurement design (Section 3.13), cleaning as editorial power (Section 4.14), honest summarization (Section 5.14), differential treatment of segments (Section 6.16), and causal language (Section 7.15) — to what this chapter adds: decisions made about individual people, automatically and at scale, on the strength of statements about people who resemble them. The chapter's ethics is not a caution about misuse at the margins. Acting on predictions is the technique's entire purpose, and the act carries obligations that the arithmetic, however honest, does not discharge on its own.

Begin with what a score is, because the ethics follows from the epistemology. The model's 585 dollars for customer C004471 is, as Section 8.13 established, a conditional average over historically similar customers, carrying a typical error the test set measured in tens of dollars. Statistically, that is a fine instrument for building a list. Ethically, the shift happens at the moment of action: the Backstage program does not gift a conditional average — it gifts, or declines to gift, a person, and the person receives the treatment their resemblance earned, not the treatment their actual future would have justified. At the individual grain, some of the model's decisions are simply wrong, in both directions.

It is tempting to say that the test MAE tells you how often. It does not, and the precision matters because this is a chapter about not overstating what a number establishes. MAE is unsigned and it is an average magnitude: it reports how far the model misses on held-out customers, on average, in dollars. It does not report a frequency of wrong decisions, it does not say which side of the cutoff any customer fell on, and it does not distinguish a twenty-dollar miss that changed nothing from a two-hundred-dollar miss that displaced someone from the list. What the test MAE is, exactly, is the honest numerical size of the model's individual-level ignorance. That is enough to dissolve the comfortable idea that a validated model has earned the right to be unexamined — and it is worth being equally precise about what validation bought. Validation did not certify the average in any general sense. It estimated average performance on a stated holdout population, at a stated snapshot, under a stated use, with sampling uncertainty attached and no claim about transport to a later period. The decisions, meanwhile, land one at a time.

Three obligations follow, each with a StyleCraft edge. The first is proportionality between the model's demonstrated reliability and the stakes of the automated act. A capture improvement of a few points honestly justifies routing lookbooks; the same model, repurposed to decide which

customers' service complaints get escalated or whose returns get flagged for review, is the same arithmetic carrying stakes it was never validated for — and the repurposing conversation, which will arrive because scores are convenient and licenses are sunk costs, is the analyst's to have. Function creep, not initial deployment, is where scoring programs most often cross their ethical lines.

The second is transparency of the automated targeting itself, in two registers. Internally, the deliverable's frame and monitoring plan are the transparency instrument: anyone operating the program can learn what the score is, what it is not, and how wrong it typically runs — which is why Section 8.13 made the frame page one. Externally, the customers on neither side of the line were asked whether their transaction histories could audition them for gifts, and while a lookbook is benign, the guide's standing test — would you be comfortable explaining the program's mechanics, aloud, to the customers on both sides of the cut? — is worth running precisely because it is easy to pass here. A program that passes it at twenty dollars a slot and fails it when the same scores set service priorities has located its own boundary, and the analyst who ran the test knows where the boundary is before the repurposing meeting starts.

The third obligation is memory that the scores inherit the past. The model learned from behavior shaped by everything StyleCraft has already done — where stores were opened, who was marketed to, which segments got offers — so the scored list partly re-awards yesterday's attention as tomorrow's gifts; the disparate-impact frame introduced with Barocas and Selbst in Section 1.13 applies whenever resemblance-based decisions recycle historical patterns, and the StyleCraft dataset's own designed trap — the newest stores, whose customers score poorly for no reason except the stores' youth — is waiting in exactly this chapter's machinery for any analyst who scores without asking what the training window could not yet know. The eligibility rule of Section 8.3 is where this bites hardest and least visibly: a population defined by having purchased in an eighteen-month window is a population defined, in part, by who was reachable during it.

> **BRIDGE**
>
> Chapter 9 takes the next step: when predictions become yes/no verdicts, fairness across segments and the feedback loops by which models cause what they predict become the central ethical machinery.

What the discipline asks of the analyst, concretely, is a fourth section on the deliverable's last page — after the frame, the leaderboard, and the monitoring plan: the acting-on-it note. Three sentences suffice for the Backstage program: what the score may be used for (ranking customers for this program's treatments); what it may not be used for without revalidation (any decision with materially higher stakes, any population outside the eligibility rule, and — if the leaderboard selected a candidate that ranks well while calibrating poorly — any use of the score as a dollar estimate); and who reviews the answer when a use-case question arrives. The note costs a paragraph and does a specific job: it converts the analyst's private judgment into the program's standing policy, which is the difference between ethics that depends on the analyst

being in the room and ethics that survives their promotion. The analyst of record signs the scores; the note is what the signature means.

CONCEPT

The Average Is Estimated; the Act Lands on a Person

Every predictive deliverable in this guide carries the distinction in writing: what the evaluation estimated (ordering quality and typical error, on a stated population, at a stated snapshot, for a stated use) and what it did not (the correctness of any individual decision, the model's fitness for stakes and populations it was never graded on, or its performance at a later date after the world has moved). Before wiring a score into an automated act, name the act's stakes, check them against the evaluation, and write the permitted-use note.

The test MAE is not a disclaimer to bury and not a count of mistakes; it is the honest size of the model's individual-level ignorance, and the program design — reversible treatments, human review where stakes rise, revalidation before repurposing — is where that ignorance is either respected or laundered.

Source: Course concept developed for this guide, informed by Barocas and Selbst (2016) and Provost and Fawcett (2013).

8.16 Chapter Summary

This chapter crossed the line Chapter 7 drew and the guide had been approaching since Part II began: from models whose coefficients are the deliverable to models whose outputs are, and from grading analyses on their honesty about the past to grading them on their accuracy about a future they have not seen. The main point is a change of posture. Explanatory work earns trust through audited structure — signs, magnitudes, units, diagnostics. Predictive work earns trust structurally: a frame declared before fitting, a wall in time between what the model may know and what it must predict, an opponent named before the race, complexity chosen inside the training data, and a grade computed once, on sealed customers, in the decision's own currency. Every impressive number produced outside that structure is flattery of one kind or another, and the chapter's standing red flag inverted the beginner's instinct in a sentence: in prediction, the result that looks too good almost never is.

The machinery arrived in three movements. The framing movement built the predictive frame — unit, target, features, horizon, and, in this chapter's version, the metrics and baselines declared alongside them — and the label as a manufactured measurement: a snapshot date, a feature window and an outcome window with both ends enforced, an eligibility rule evaluable at the snapshot, and the insistence that a zero is a label rather than a gap. The modeling population was distinguished from the deployment population, and the customers the rule excludes were named as an activation problem rather than absorbed silently. Leakage was named as the canonical error and cataloged by route — temporal, target-derived, outcome-window contamination, split contamination, population — with the audit, not the software, as its only

reliable detector, and with the split-contamination rule stated precisely enough to be usable: what contaminates a split is a step that estimates something from data, not any step performed before it.

The evaluation movement built the honest exam. Train/test splits defeat in-sample flattery, and the chapter was careful about what they buy: a random customer holdout estimates cross-sectional generalization at the historical snapshot and does not measure time-forward transport to a later scoring date. MAE and RMSE were read in dollars and separated from the decision metric, because the Backstage program acts through a fixed ranked cutoff and a model can improve its typical miss while worsening the ordering at the line — so top-decile capture leads the frame and MAE leads the calibration reporting. Baselines were operationalized from Section 2.7's interpretive discipline into working opponents, with the incumbent's sorted spreadsheet dignified as the baseline with teeth and the direction of the error metrics stated plainly, since a model far below the baselines on MAE is the suspicious one and a model far above them is the broken one. Overfitting and underfitting were read jointly from the training-versus-validation signature, with the guarantee attached to training RMSE rather than to training MAE, and with the limit on honest performance described as a documented picture of the outcome's volatility rather than a computed noise floor the chapter had not earned. Cross-validation stopped being a concept and became the instrument: one fold generator, one leaderboard, every candidate and both baselines scored by the same call, the test set sealed throughout.

The deployment movement carried the score into the world: rank and cut under a fixed budget with a declared tiebreaker, the capture head-to-head against the incumbent list, expected-value framing at concept level with the word incremental flagged as unearned, and the deployed model's real life stated in three terms rather than one — data drift in the inputs, concept drift in the relationship, performance decay as the observable consequence — each with its own monitoring implication and its own trigger. The AI section named the assistant-era failures — the convenient-table leak, the celebrated impossible fit, the missing opponent, the metric that does not match the act — and installed the five-point audit that the labs then ran three ways: on the honest pipeline, on a deliberately manufactured leak whose smoking gun was a single impossible range check, and on an assistant's unframed enthusiasm. Sixteen numbered cells carried the argument into executable form: the miniature baselines with their one-slot capture, a snapshot-safe feature builder that takes its window as an argument, both windows enforced at both ends, the eligible population reconciled against the customer file, the labeled table audited before anything was fitted, the split frozen by customer identity, five candidates registered with every learned step inside a pipeline and a selection rule written before the race, the cross-validated leaderboard with its paired fold margins, the flexibility dial turned without touching the test set, the rule executed unedited to name a finalist, that finalist opened against the sealed customers once, the leak manufactured on the same customers and caught by arithmetic, and the two ranked lists compared deterministically. The interpretation, practice, and ethics sections carried the results into the meeting room, where the vendor's 96 percent met the five questions and a vocabulary correction, the score was demoted from verdict to ordering, the capture gap

was distinguished from a treatment effect, and the act of gifting eight hundred people on the strength of their resemblance to other people acquired its permitted-use note.

Looking ahead, the chapter's target was a dollar amount, and dollars were a mercy: a numeric miss has a size, and MAE could price it. Most of marketing's predictive questions refuse the mercy. Will this customer churn; will she respond; will he click, convert, return — yes or no, where a prediction is not close or far but right or wrong, and where being wrong has two faces with two different prices. The next chapter takes up classification, propensity, and churn: the logistic machinery that turns features into probabilities, the confusion matrix that gives the two error types their business names, the lift and gains machinery this chapter borrowed one exhibit from, and the threshold decision — set not by statistics but by the asymmetric costs Section 2.7 planted and this chapter previewed — where the analyst's model finally negotiates directly with the CFO's economics.

8.17 Exercises for Practice and Homework

The following exercises practice the chapter's main habits: declare the frame before touching data, define labels in time with both ends of both windows enforced, trace every feature for leakage, choose complexity inside the training data, grade models only out of sample and only against named baselines, read the training-versus-validation signature, translate scores into decisions with their economics attached, and treat too-good results as audits waiting to be run. They are organized into two groups. Core chapter practice is the required path and should be completed by every student, and it holds the two homework submissions from which your instructor will assign a subset; Assignment #3, Drivers and Predictive Modeling, draws on the homework sets of Chapter 7 and this chapter. In-class activities are prepared for discussion rather than submitted. Each exercise also carries its assignment label — required practice, homework submission, or in-class discussion — so that instructors can assign selectively.

8.17.1 Core Chapter Practice

Exercise 8.1 Concept Check (Required Practice)

Answer each in two or three sentences, in your own words.

1. State what changes when a model's outputs, rather than its coefficients, become the product, and name one practice that is acceptable in explanatory work and a failure in predictive work.
2. Define the four commitments of the predictive frame, and explain why the decision metric and the baselines must be declared before fitting rather than after.
3. Explain why “future spend” is a manufactured label rather than an existing column, and state why each of the four declared dates is load-bearing rather than decorative.
4. State this chapter's eligibility rule, explain why both of its conditions can be evaluated at the snapshot, and name the population it excludes and what should happen to that population.

5. Define leakage and explain why it cannot be detected by fitting statistics or by a train/test split, only by audit. What is its most reliable symptom?
6. A colleague says that any transformation applied before the split contaminates it. Give one transformation for which this is true and one for which it is false, and state the test that distinguishes them.
7. An analyst reports that their model's training MAE is 31 dollars and its five-fold cross-validated MAE is 78 dollars. Diagnose, using Table 8.4's vocabulary, and prescribe.
8. Why does this guide compute the mean baseline's prediction from the training labels only, and which rule from Section 8.5 is that the smallest example of?
9. RMSE substantially exceeds MAE on the same predictions. What does the gap announce, and which StyleCraft customers — by Chapter 5's findings — are almost certainly responsible?
10. Explain why capture and MAE can disagree about which candidate is better, and state what the deliverable should do when they do.
11. State the three clauses of the model-comparison discipline, and for each, describe the loophole it closes.
12. Distinguish data drift, concept drift, and performance decay, and state which of the three can be detected without waiting for outcomes to mature.

Exercise 8.2 Draw the Frame (Required Practice)

For each brief below, write a complete predictive frame — unit of prediction, target with a fully defined label (snapshot, feature window with both ends, outcome window with both ends, filters, eligibility), horizon, decision metric, calibration metrics, and the strongest honest baseline — and then name the single most likely leakage route from Table 8.2 and how your frame forecloses it.

1. The CRM manager wants to time win-back email for customers likely to lapse: predict each customer's days until next purchase.
2. Merchandising wants to know, at the end of each customer's first month, what they will spend over their first year, to set onboarding investment.
3. The email team wants to predict, per customer, next quarter's email-driven revenue — and someone proposes using each customer's full-history open rate as a feature. Address the proposal explicitly.
4. The retention team wants to score every customer monthly, including customers acquired last week. State what your frame can and cannot do for the newest customers, using the representation argument of Section 8.3 rather than a claim about technical impossibility.

Exercise 8.3 Baseline Arithmetic on the Miniature (Required Practice)

Using only the ten-customer miniature of Lab 8.1 Part A (no code): verify the mean baseline's MAE of 53.84 by computing all ten absolute errors and summing; verify the last-value baseline's MAE of 84.80 the same way; identify the four customers responsible for the incumbent rule's defeat and state, for each, the behavioral profile from Chapters 5 and 6 that explains the miss; explain why the RMSE gap between the two baselines (161.80 versus 87.42) is proportionally

wider than the MAE gap, and what that says about how the incumbent fails. Then do the decision arithmetic: with two slots rather than one, list which two customers each rule selects and what share of the ten customers' outcome-window spend each pair captures. Close with one sentence on why the capture comparison is a harsher verdict on the incumbent than the MAE comparison, and one on what a model must do to beat both baselines at once.

Exercise 8.4 Spot the Leak (Required Practice)

Each pipeline below contains at least one leakage route from Table 8.2, or none. Name the route (or certify the pipeline clean), cite the table row, and state the repair.

1. To predict spend_next6m from a December 31 snapshot, an analyst includes loyalty_tier as of the June 30 database extract, reasoning that tier changes slowly.
2. A churn model's training population is “customers with at least three orders in the 24-month window,” with churn labeled in the final six months.
3. An analyst standardizes all features on the full labeled table, then splits 80/20, fits, and reports test MAE against a mean baseline computed from training labels.
4. An analyst expands two predictors into third-degree polynomial terms on the full labeled table, then splits 80/20 and fits. Be careful with this one, and justify your verdict with the test from Section 8.4 rather than with the position of the split.
5. To predict first-year spend at the end of month one, the feature list includes number of orders in the first 30 days and average order value over the customer's lifetime.
6. A team refits Chapter 6's segmentation on all labeled customers, adds the resulting segment indicators as features, then splits and cross-validates the model. Name what leaked, and state the two structural repairs Section 8.9 offers.
7. An assistant's pipeline for the Backstage frame reports test R^2 of 0.95, and its feature importance list is led by a column named monetary.

Exercise 8.5 The Full Predictive Deliverable (Homework Submission)

Complete Labs 8.1 and 8.2 on the certified files and assemble the scoring deliverable per Section 8.13: the one-page frame with its seven declarations and four dates; the reconciled population counts, naming the excluded groups; the cross-validated leaderboard with both baselines and every candidate's margin, computed on the training customers only; the single test-set table with capture leading and MAE and RMSE beside it; the capture head-to-head against the incumbent list with the dollar gap and the count of names that change; the expected-value sketch with the incremental caveat stated and the holdout named; the vendor-verdict memo built on Table 8.5's five questions; the monitoring plan with its inputs, its outcome benchmark, and its retraining trigger; and the permitted-use note per Section 8.15. Submissions are graded on the frame's completeness, the order of operations evidenced in the notebook, and the audit evidence as heavily as on the code.

Exercise 8.6 AI Pipeline Audit (Homework Submission)

Give an AI assistant the raw StyleCraft tables and this deliberately loose prompt: “Build me the best possible model to predict customer spend — maximize accuracy.” Then audit the response with Table 8.5's five points, in order and in writing: reconstruct the frame the assistant implicitly chose and compare it to Table 8.1; trace every feature to its window, running the recency range check if applicable; check the split and preprocessing hygiene, including whether any complexity or tuning decision was made by consulting a test set; determine what baseline, if any, the pipeline was compared against, and compute the missing ones yourself on the same customers; and grade the reported performance for plausibility against the volatility figures you recorded in Code 8.6, stating whether the assistant's own narration celebrated or questioned it. Separately, list every performance claim in the reply and mark each as earned under exam conditions or not, flagging any use of classification vocabulary for a numeric target. Document the full exchange per the AI-use documentation template in Appendix D, and conclude with two sentences on which audit point caught the most serious problem.

8.17.2 In-Class Activities

Exercise 8.7 The Vendor Meeting (In-Class Discussion)

Your team will role-play the diligence meeting for the vendor's predictive-spend module. Prepare: the five questions of Table 8.5 translated into vendor-facing language; a one-sentence correction of the “96 percent accuracy” claim that a non-technical buyer could repeat, distinguishing accuracy from R^2 and noting what an out-of-sample R^2 can do that a percentage cannot; for each question, the answer an honest vendor would give and the answer that should end the meeting; the acceptance test you would write into the pilot, per Section 8.14's third vignette, naming who holds the outcome window and who computes the metrics; a position on what evidence would justify the license fee over the in-house model, expressed as a capture margin in dollars; and one question about the maintenance contract that Section 8.14's second vignette proves is not optional. Half the class prepares the vendor's side; the honest version of that brief is harder, and more instructive, than the buyer's.

Exercise 8.8 Find the Flaw in the Predictive Pipeline (In-Class Discussion)

Each scenario below contains at least one flaw from this chapter. Name it, cite the section, and state the repair.

1. A team celebrates a spend model whose test MAE beats the mean baseline by 40 percent. No other baseline appears in the deck, and the CRM team's trailing-spend rule is not mentioned.
2. An analyst fits polynomial models of degree 1 through 9, plots training and test error for each, picks the degree where test error is lowest, and reports that degree's test error as the honest out-of-sample estimate.
3. An analyst tunes a model through thirty rounds of adjust-and-check against the test set, then reports the final test MAE as the honest out-of-sample estimate.

4. A model's test MAE of 51 dollars is reported as “the model is wrong by 51 dollars about every customer, so it cannot be trusted for individual decisions” — and the program is cancelled.
5. A deck reports that the model's list captures 6 percentage points more spend than the incumbent's and concludes that the program will generate that much additional revenue.
6. A pipeline predicts six-month spend with test R^2 of 0.97; the lead analyst schedules a celebration and a production launch for the same week.
7. A deployed value model performs beautifully for three quarters; monitoring consists of confirming the scores arrive on schedule. A repricing launches in month ten.
8. A comparison table shows Model A winning on MAE and Model B winning on RMSE; the report recommends Model B because “it wins on the more sophisticated metric.” The frame declared capture first and MAE second.
9. Two analysts run the same notebook on the same data and produce eight-hundred-name lists that differ by fourteen customers. Nothing in either notebook errored.

8.18 Glossary of Terms

This glossary includes only the terms introduced in this chapter. Each definition is tied to the sources used in the chapter rather than added for decoration.

Baseline model. A deliberately simple prediction rule — mean, majority class, or last value — evaluated on the same folds, the same test set, and the same metrics as a candidate model, whose margin of defeat is the measure of the model's demonstrated value (adapted from Provost & Fawcett, 2013).

Bias–variance tradeoff. The intuition that prediction error decomposes into systematic error from oversimplification (bias) and instability from fitting sample-specific noise (variance), with model flexibility trading one for the other and honest performance minimized between the extremes (adapted from James et al., 2021; Domingos, 2012).

Concept drift. Change over time in the relationship between a model's inputs and its target, so that the learned mapping no longer describes how the outcome is generated; invisible in the input distributions and detectable only as outcomes mature (adapted from Gama et al., 2014).

Cross-validation. Estimation of out-of-sample performance from training data alone, by k-fold rotation of a held-out fold and averaging of the k held-out scores; the standard instrument for development-time decisions, including model complexity, while the test set stays sealed (adapted from James et al., 2021; scikit-learn developers, 2026b).

Data drift. Post-deployment change in the distribution or quality of a model's input features, so that scored cases increasingly differ from training cases; detectable by watching the inputs alone. Covariate shift is its special case, in which the predictor distribution

changes while the conditional relationship to the outcome remains stable (adapted from Gama et al., 2014).

Expected value framing. The translation of predictions into actions by weighing each action's possible outcomes by their probabilities and subtracting its costs, so that score thresholds become economic lines rather than arbitrary ranks (adapted from Provost & Fawcett, 2013).

Feature window. The declared span of time, bounded at both ends and closing at the snapshot date, from which a predictive model's features may be computed; the past side of the label's wall in time (adapted from Kuhn & Johnson, 2013).

Horizon. The forward distance between the snapshot date and the end of the outcome window: how far into the future the model's claim extends, constraining validation timelines and retraining cadence (adapted from Provost & Fawcett, 2013).

In-sample flattery. This guide's name for the systematic optimism of performance measured on the data a model was fitted to, which grows with model flexibility and cannot be estimated from the training data itself (adapted from James et al., 2021).

Label. The constructed, recorded value of the target for a historical case, manufactured by choosing a snapshot date, a feature window, an outcome window, and eligibility rules; the answers a predictive model trains on (adapted from Provost & Fawcett, 2013; Kuhn & Johnson, 2013).

Leakage. The presence in training features or evaluation procedure of information unavailable at real prediction time — from the future, from the target's construction, or from the evaluation data — inflating measured performance that collapses on deployment (adapted from Kaufman et al., 2012).

MAE (mean absolute error). The average absolute difference between actual and predicted values, in the target's native units, weighting all misses equally; an unsigned average magnitude rather than a count of wrong decisions, and this guide's lead calibration metric (adapted from Willmott & Matsuura, 2005).

Outcome window. The declared span of time after the snapshot date, bounded at both ends, over which the label is measured; the future side of the wall, which no feature may touch (adapted from Kuhn & Johnson, 2013).

Overfitting. A model's capture of training-data-specific noise that does not generalize, diagnosed by training performance substantially exceeding held-out performance and worsening with added flexibility (adapted from James et al., 2021; Domingos, 2012).

Performance decay. The erosion of a deployed model's predictive quality over time as data drift, concept drift, or both accumulate, observed by monitoring realized error as fresh outcomes mature (adapted from Gama et al., 2014; Sculley et al., 2015).

Pipeline. An object chaining every learned step of a model — scaling, imputation, clustering, the estimator itself — so that all of them are refitted together on each training fold; the structural defense against split contamination (adapted from scikit-learn developers, 2026a).

Predictive frame. The four declared commitments of a predictive model — unit of prediction, target, features, and horizon — written before fitting, together with the decision metric, the calibration metrics, and the baselines, as the predictive extension of the analytic specification (adapted from Provost & Fawcett, 2013; Kuhn & Johnson, 2013).

Predictive modeling. The construction of models whose purpose is accurate output for individual unseen cases, evaluated out of sample, as distinct from explanatory modeling graded on the honesty of its coefficients (adapted from Shmueli, 2010; Provost & Fawcett, 2013).

RMSE (root mean squared error). The square root of the average squared difference between actual and predicted values, in native units but disproportionately sensitive to large misses; its excess over MAE announces error concentration (adapted from Willmott & Matsuura, 2005).

Snapshot date. The declared moment that splits time in a label's construction: features may use only information available on or before it; the label is measured only after it (adapted from Kuhn & Johnson, 2013).

Split contamination. The leakage route in which a step that estimates something from data is fitted before the split, letting held-out cases influence the values the model trains on; distinguished from transformations that estimate nothing and are therefore unaffected by the order of operations (adapted from Kaufman et al., 2012; scikit-learn developers, 2026a).

Test set (holdout). The randomly held-out portion of labeled cases, sealed during development and consulted once for the selected model, whose performance estimates cross-sectional out-of-sample error at the historical snapshot (adapted from James et al., 2021).

Capture (at a selection share). The share of a population's actual outcome-window spend held by the customers a model ranks inside the program's capacity — the capacity divided by the population being ranked. It is the decision metric of a fixed-capacity ranked targeting program, and the metric on which a model's list is compared with an incumbent's; where the share is a tenth it is the familiar top-decile case. A constant score carries no ranking content, so its capture is defined only once ties are broken on a declared rule, and a random tiebreak makes that figure the selection share itself (adapted from Provost & Fawcett, 2013).

Training set. The portion of labeled cases from which models, every learned preprocessing step, and every development decision including model complexity are derived (adapted from James et al., 2021).

Underfitting. A model's failure to capture structure that does generalize, diagnosed by poor performance on training and held-out data alike (adapted from James et al., 2021).

8.19 Further Readings

Students who want additional background may begin with the following readings. The leakage and framing sources are listed first, methods second, software and deployment last.

- Kaufman et al. (2012) for the definitive treatment of leakage — its formulation, its taxonomy, and the case studies from data-mining competitions where it repeatedly crowned false winners; the extended companion to Section 8.4.
- Provost and Fawcett (2013), especially the chapters on model evaluation and the expected-value framework, for the business-first treatment of everything from baselines to score-to-decision translation; the closest published relative of this chapter's approach.
- Domingos (2012) for twelve compact pages on what practitioners most often get wrong — overfitting, the futility of accuracy without generalization, and why more sophisticated is not a direction; the fastest inoculation in the field's literature.
- James et al. (2021), Chapters 2 and 5, for the standard accessible development of the bias–variance tradeoff, validation, and cross-validation, one level of formality above this chapter — and, for the implementation of exactly those ideas, the two pages of official scikit-learn documentation these labs are built on (scikit-learn developers, 2026a, 2026b): the common-pitfalls page on preprocessing leakage and pipelines, and the cross-validation page on using folds for development while a test set is reserved for the final grade.
- Willmott and Matsuura (2005) for the short, pointed argument on MAE versus RMSE and what each actually measures; useful ammunition for the metric-choice conversation.
- Sculley et al. (2015) for the famous statement that the model is the smallest box in the system diagram — technical debt, entanglement, and why deployment is where predictive work is won and lost; the companion to Section 8.10.

8.20 References

Barocas, S., & Selbst, A. D. (2016). Big data's disparate impact. *California Law Review, 104*(3), 671–732. https://doi.org/10.15779/Z38BG31

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM, 55*(10), 78–87. https://doi.org/10.1145/2347736.2347755

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., & Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Computing Surveys, 46*(4), Article 44. https://doi.org/10.1145/2523813

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An introduction to statistical learning: With applications in R* (2nd ed.). Springer. https://doi.org/10.1007/978-1-0716-1418-1

Kaufman, S., Rosset, S., Perlich, C., & Stitelman, O. (2012). Leakage in data mining: Formulation, methodology, and avoidance. *ACM Transactions on Knowledge Discovery from Data, 6*(4), Article 15. https://doi.org/10.1145/2382577.2382579

Kuhn, M., & Johnson, K. (2013). *Applied predictive modeling*. Springer. https://doi.org/10.1007/978-1-4614-6849-3

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.

Provost, F., & Fawcett, T. (2013). *Data science for business: What you need to know about data mining and data-analytic thinking*. O'Reilly Media.

scikit-learn developers. (2026a). *Common pitfalls and recommended practices* [Software documentation]. https://scikit-learn.org/stable/common_pitfalls.html

scikit-learn developers. (2026b). *Cross-validation: Evaluating estimator performance* [Software documentation]. https://scikit-learn.org/stable/modules/cross_validation.html

scikit-learn developers. (2026c). *PolynomialFeatures* [Software documentation]. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

scikit-learn developers. (2026d). *r2_score* [Software documentation]. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems 28* (pp. 2503–2511). Curran Associates.

Shmueli, G. (2010). To explain or to predict? *Statistical Science, 25*(3), 289–310. https://doi.org/10.1214/10-STS330

Willmott, C. J., & Matsuura, K. (2005). Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research, 30*(1), 79–82. https://doi.org/10.3354/cr030079