A Score You Can Explain

Transparent Scoring from Rubric to Decision with Urteil

The Urteil project

First working edition — September 2026
Built with Urteil 0.2.0

How to use this book

This book develops a scoring system from its first question to an executable decision rule. It begins with a small pilot-grant rubric that can be checked by hand, then follows a larger startup-screening example through fitting, evaluation, and threshold selection. The objective is to make every step inspectable: what an answer means, how many points it contributes, how weights were obtained, and what evidence supports the resulting classification.

All cases, labels, and performance results in this book are synthetic. The startup outcomes come from the package's mathematical generator. They are neither observed investment returns nor recommendations about companies. The grant example is an invented policy for allocating review attention. These examples demonstrate software behavior and an analysis workflow.

The chapters combine conceptual explanations, mathematical definitions, and actual Urteil commands. Tables and the threshold figure are generated during the build; their inputs are saved alongside the book. The book does not need API credentials or remote inference. Its default build needs scikit-learn for the fitted examples, and PDF compilation needs a local TeX installation. EDSL export commands that require EDSL are described separately from the exports executed by the book.

Build and inspect

From the repository root, install into a local environment and build:

```
python3 -m venv .venv
. .venv/bin/activate
python -m pip install -e '.[manual]'
urteil scoring docs manual --output-dir build/manual --pdf
```

Use a new output directory for each build. This preserves previous results for comparison. Omit `-pdf` to generate the LaTeX sources, examples, and results without compiling. Use `-without-fit` to run the manual-point examples without modeling dependencies; the book will explicitly mark fitted results as absent. It will never substitute invented results for a skipped computation.

```
urteil scoring docs manual --output-dir build/manual-core \
  --without-fit --pdf
```

The PDF build uses `pdflatex` with shell escape disabled. Required TeX packages include Latin Modern, geometry, AMS math, booktabs, microtype, listings, PGFPlots, hyperref, and xurl. Three compilation passes resolve the contents and references. TeX logs are retained beside the source.

Read Chapters 1–3 to build a manual scorecard. Chapters 4–6 develop and challenge fitted scores.

Chapter 7 explains the survey handoff, and Chapter 8 describes how to maintain a rule. The appendices map the artifacts and record implementation details that affect interpretation.

Contents

1 Start with an inspectable decision **1**

1.1 Interpretability belongs in the model 1

1.2 A score, an outcome, and an action 2

1.3 What additivity assumes 2

1.4 A working sequence 2

2 Turn a judgment into a rubric **3**

2.1 Define the answer before assigning points 3

2.2 Calibration examples are annotations 4

2.3 Represent categories and simultaneous risks 4

2.4 Inspect the rubric before fitting 4

2.5 Create versus draft 5

3 Build and inspect a manual scorecard **6**

3.1 Combine numeric weights and answer points 6

3.2 Work one case by hand 7

3.3 Make the tradeoffs visible 7

3.4 Missing values are substantive 7

3.5 Publish a readable rule locally 8

4 Learn a small integer rule **9**

4.1 A larger worked example 9

4.2 Create the manual benchmark first 10

4.3 What the fitting backend does 10

4.4 Integerization changes the model 11

4.5 The connection to SLIM and RiskSLIM 11

5 Choose a threshold and test the final rule **12**

5.1 Give each split one job . . . 12
5.2 Count decisions, not just correct answers . . . 12
5.3 Select the cutoff on validation data . . . 13
5.4 Examine the size of the point system . . . 13
5.5 Challenge the measurement process . . . 14

6 Read the results of this build 15
6.1 The prespecified manual benchmark . . . 15
6.2 The fitted rule in this build . . . 15
6.3 One case, contribution by contribution . . . 16
6.4 Evaluate the rule that will actually run . . . 16
6.5 Sensitivity to the point budget . . . 17
6.6 How to interpret a change in accuracy . . . 17
6.7 Use the figures for diagnosis . . . 17

7 Carry the rule into a survey 19
7.1 Two survey modes . . . 19
7.2 Exports executed by this book . . . 19
7.3 Create native EDSL artifacts . . . 20
7.4 Run a local scoring job explicitly . . . 20
7.5 Check parity on the answers you support . . . 20
7.6 Separate assessment from arithmetic . . . 21

8 Maintain a rule people can inspect 22
8.1 Keep a release bundle . . . 22
8.2 Review a disputed case . . . 22
8.3 Respond to changes in the population . . . 23
8.4 Extend the example deliberately . . . 23
8.5 Exercises . . . 23

A Artifacts and command reference 24
A.1 What a build writes . . . 24
A.2 CLI map . . . 25

B Implementation details that affect interpretation 26
B.1 Input coercion . . . 26

B.2 Study ratings and legacy standalone scoring 26
B.3 Standalone validation is limited 27
B.4 Dependencies and reproducibility 27

Bibliography **28**

Chapter 1

Start with an inspectable decision

A reviewer sees a promising proposal. The problem matters, the evidence is incomplete, and the team may lack delivery capacity. A useful scoring system makes those considerations explicit enough that another reviewer can reproduce the assessment and disagree with a particular premise.

Urteil starts from a domain rubric. A rubric names the questions, defines the permitted answers, and supplies examples of how to interpret them. A scorecard combines that rubric with numeric weights or answer-specific points and classification thresholds. The package can construct a scorecard manually or estimate weights from a table of labeled examples. It can then score local answers, print a readable rule, and export survey artifacts.

1.1 Interpretability belongs in the model

Cynthia Rudin argues for inherently interpretable models when their structure can support the decision task, rather than relying on explanations fitted around opaque predictors [1]. An additive scorecard is one such structure: the quantities displayed to the reviewer are the quantities used to calculate the score.

This connection is the package's motivation, not a claim of authorship or endorsement by Cynthia Rudin. The package implements additive scorecards and survey exports. It does not currently implement learned rule lists, a general constrained integer optimizer, or probability calibration. Its optional fitting backends are discussed precisely in Chapter 4.

For numeric answers $x_1, \ldots, x_p$, the simplest rule is

$$S(x) = b + \sum_{j=1}^{p} w_j x_j, \tag{1.1}$$

where b is a shared intercept and w_j is the weight on criterion j. The contribution $w_j x_j$ can be shown beside the answer. A reader can reconstruct the total without a separate explanation model.

Small integers can make arithmetic easier. They do not, by themselves, make a criterion meaningful. "Team quality: 4" is hard to reproduce unless the rubric says what evidence distinguishes 4 from 3. Interpretability requires understandable inputs as well as a visible calculation.

1.2 A score, an outcome, and an action

Keep three objects distinct. The *outcome* is the thing labeled in training data. The *score* orders cases according to the rule. The *classification* assigns labels to score bands. A subsequent organizational action may use those labels, but the package does not carry out that action.

For the grant example, `advance` means advance to the next review stage. It does not mean that the proposal has been funded. For the startup example, `invest` is a binary label generated by a fictional mathematical process. Calling a field `invest` does not turn a score into an estimate of financial return.

Suppose a rule assigns `review` at 4 points and `advance` at 7. Then

$$c(S) = \begin{cases} \text{negative} & S < 4, \\ \text{review} & 4 \le S < 7, \\ \text{advance} & S \ge 7. \end{cases}$$

Urteil uses inclusive lower bounds and selects the qualifying band with the highest minimum score. The default below all thresholds is the literal label `negative`.

1.3 What additivity assumes

Equation (1.1) makes tradeoffs explicit. With a two-point weight on evidence, one extra evidence unit contributes two points whatever the other answers are. A severe delivery problem can be offset by favorable answers elsewhere if the negative contribution is finite. Those are modeling choices, not mathematical necessities.

If a criterion is a nonnegotiable eligibility condition, use an explicit eligibility step in the surrounding process. Urteil has no native veto field. If the effect of one answer depends on another, a plain additive rule may be inadequate. An explicitly defined interaction feature can express some dependencies, but the user must construct and validate it; the CLI does not discover interactions automatically.

A short model may also have redundant inputs. If “strong traction” and “repeat paid usage” measure almost the same evidence, including both can count that evidence twice. Start with the meaning and provenance of each input before deciding how many points it deserves.

1.4 A working sequence

Begin by specifying the unit of analysis and the decision stage. Write observable questions, give their answer anchors, and score a few cases manually. Agree on the direction and scale of each contribution. Only then fit weights if suitable labels exist. Evaluate the final integer scorecard on data that did not determine it, choose a threshold on separate validation data, and preserve the released artifact.

The rest of this book follows that sequence. Manual points express a proposed policy. Fitted points summarize relationships in labeled data. Either can be transparent; either can be wrong. Transparency gives reviewers a concrete object to examine and revise.

Chapter 2

Turn a judgment into a rubric

Our small example screens applications for a pilot grant. Reviewers must distinguish a specific problem, credible evidence, implementation readiness, and delivery risks. Four questions cover these considerations, using all four question types that Urteil currently supports.

2.1 Define the answer before assigning points

A yes/no question should have a reproducible boundary. A scale should identify what changes between adjacent levels. A multiple-choice question should use mutually exclusive values; a checkbox question should allow more than one applicable condition.

Question	Kind	Stored answers
Clear problem	`yes_no`	0 or 1
Evidence	`linear_scale`	Integers 0 through 4
Readiness	`multiple_choice`	idea, pilot, ready
Risks	`checkbox`	A list of selected values

Use stable names such as `clear_problem`. These names connect rubric questions, CSV columns, answer JSON, and exported survey fields. Use valid Python-style identifiers for compatibility with generated templates. Display text can be more descriptive.

The following commands are the commands executed by this book's build. Paths in all generated command listings are relative to the output directory. To replay them, use a fresh working directory, with Urteil installed. Reading the existing artifacts does not require replaying the commands.

```
urteil scoring rubric create 'pilot grant triage' --outcome advance \
  --output grant/rubric.json

urteil scoring question add grant/rubric.json clear_problem --text \
  'Is the problem specific and supported by evidence?' --kind yes_no \
  --example '0=0::Vague problem statement' --example \
  '1=1::Specific problem with an external source'

urteil scoring question add grant/rubric.json evidence --text \
  'How strong is the evidence for the proposed mechanism?' --kind \
  linear_scale --min-value 0 --max-value 4 --label 0=Absent --label \
```

```
  4=Replicated --example \
  '4=4::Independent replications in a relevant setting'

urteil scoring criterion example add grant/rubric.json evidence --answer \
  2 --score 2 --rationale \
  'One relevant pilot, with unresolved confounding'
```

2.2 Calibration examples are annotations

An example such as `2=2::One relevant pilot` records an answer, an illustrative criterion score, and a rationale. It helps reviewers align their interpretation of an answer. It does not create a training observation, change a coefficient, or install an answer-to-points function. Both fitting and local scoring operate independently of these example annotations.

This distinction matters when the eventual scorecard gives evidence a weight of 2. An evidence answer of 2 contributes 4 points even though its rubric example records `score: 2`. The rubric describes the evidence level; the scorecard determines its contribution. If a genuinely nonlinear answer-to-points mapping is intended, define it using `-point`, as in Chapter 3.

An effective anchor includes the evidence that would justify the answer. For a score of 2, identify the kind of pilot and the unresolved uncertainty. For a score of 4, identify what counts as an independent replication in a relevant setting. The package stores these descriptions but does not check that a reviewer actually observed the evidence.

2.3 Represent categories and simultaneous risks

We next add readiness and delivery risks:

```
urteil scoring question add grant/rubric.json readiness --text \
  'What is the implementation stage?' --kind multiple_choice --option \
  Idea=idea --option Pilot=pilot --option Ready=ready

urteil scoring question add grant/rubric.json risks --text \
  'Which delivery risks are present?' --kind checkbox --option \
  'Staffing gap=staffing' --option 'Missing access=access'
```

The left side of each `label=value` option is display metadata. The right side is the stored answer used by the scorecard. Use `ready`, not `Ready`, when supplying local answer JSON for this example. Mappings are keyed by the stored values and are case sensitive.

Checkbox answers in local JSON must be lists, for example `["staffing", "access"]`. A string containing commas is still a single string. Urteil's CSV scenario conversion parses scalar numbers but does not decode JSON lists embedded in cells. Prepare list-valued scenarios explicitly in Python when both risks must be passed to an EDSL scoring survey.

2.4 Inspect the rubric before fitting

```
urteil scoring rubric show grant/rubric.json
```

Inspect the question names, permitted values, examples, and thresholds. In a real project, two reviewers can score the same small set of cases independently, then compare disagreements criterion by criterion. Revise ambiguous wording before fitting: fitting cannot tell whether disagreement came from a vague question or a genuine difference in evidence.

Treat a rubric change as a measurement change. Replacing "has a pilot" with "has a paid pilot" alters the input even if the field name stays the same. Preserve the old rubric with any old scorecards and record which wording produced each batch of answers.

2.5 Create versus draft

`rubric create` starts with no questions. This is the path used here. The separate `draft` command supplies generic questions and a starter threshold:

```
urteil scoring draft "pilot grant triage" --outcome advance \
  --output draft-rubric.json --humanize-output creator.json
```

The draft is a local template; it does not call a language model. Replace its generic signals with operational questions before using it. The optional humanize output is a payload for another tool, not a published survey.

Chapter 3

Build and inspect a manual scorecard

A manual scorecard makes a proposed policy concrete before any statistical fitting. Our grant rule awards two points for a clear problem, two points per evidence level, up to three points for readiness, and deductions for delivery risks. These choices are illustrative. They are easy to dispute because they are visible.

3.1 Combine numeric weights and answer points

For a numeric feature, Urteil multiplies the coerced integer answer by its weight. For a question with an answer-point map p_j, it looks up the selected answer instead. If the answer is a list, it sums the points for each selected item:

$$g_j(a_j) = \begin{cases} p_j(a_j), & \text{mapped scalar answer,} \\ \sum_{v \in a_j} p_j(v), & \text{mapped list answer,} \\ w_j \ \mathrm{int}(\mathrm{coerce}(a_j)), & \text{weighted answer.} \end{cases} \qquad S(a) = b + \sum_j g_j(a_j).$$

If both a map and a weight are supplied for the same question, the map takes precedence. Unknown mapped values contribute zero. Therefore an incomplete mapping can silently omit a category. Supply all intended categories, including explicit zero-point categories.

```
urteil scoring threshold add grant/rubric.json review --minimum-score 4

urteil scoring threshold add grant/rubric.json advance --minimum-score 7

urteil scoring scorecard create grant/rubric.json --output \
  grant/scorecard.json --weight clear_problem=2 --weight evidence=2 \
  --point readiness.idea=0 --point readiness.pilot=1 --point \
  readiness.ready=3 --point risks.staffing=-2 --point risks.access=-3
```

The threshold commands modify the rubric. Creating the scorecard copies those thresholds and embeds the rubric in a new JSON artifact. Later edits to `grant/rubric.json` do not automatically update an existing scorecard. Recreate the card or explicitly update its saved specification.

3.2 Work one case by hand

Consider a proposal with a clear problem, an evidence rating of 2, a ready implementation, and a staffing gap. The build writes the following answer file:

```
{
  "clear_problem": 1,
  "evidence": 2,
  "readiness": "ready",
  "risks": [
    "staffing"
  ]
}
```

Its arithmetic is

$$S = 2 \times 1 + 2 \times 2 + 3 - 2 = 7.$$

The score reaches the inclusive lower bound for `advance`. The local command checks the calculation:

```
urteil scoring score grant/scorecard.json grant/answers.json --output \
  grant/score.json
```

The executed command returns score 7 and classification **advance**. The intercept is zero here. If present, a card-level intercept contributes once to every case. It changes the relationship between a score and a fixed threshold without changing the ordering of cases under a fixed rule.

3.3 Make the tradeoffs visible

A staffing gap costs two points, exactly one evidence level. Moving readiness from pilot to ready adds two points. An additional access risk costs three points and brings this case from 7 to 4, putting it in `review`. An evidence rating of 1 instead of 2 would bring it to 5.

These are useful local sensitivity questions: which plausible answer changes would alter the classification? They do not identify causal interventions. Changing an entry from 1 to 2 on a form does not cause the underlying evidence to improve.

The rule ranges from -5 to 13 over its intended answer space: the maximum is $2 + 8 + 3$, and both risks together contribute -5. Consequently the advance threshold of 7 is reachable. Checking attainable score ranges catches obvious problems such as a threshold that no valid case can meet. The CLI does not perform that check for you.

3.4 Missing values are substantive

The local scorer defaults absent weighted answers to zero. It also coerces `None` to zero. Missing evidence thus looks like zero evidence. Missing risk information can be worse: it can avoid a penalty entirely. In a mapped field, an absent answer is looked up using the default value 0; a mapping containing the key `"0"` can therefore award or deduct points for an omitted field.

Validate completeness before calling the scorer when zero and missing have different meanings. Keep an explicit unknown category if the policy permits unknown answers, assign its points deliberately,

and measure how often it occurs. This book uses complete, valid answers for all generated numerical comparisons.

3.5 Publish a readable rule locally

```
urteil scoring present grant/scorecard.json --output \
  grant/scoring-rule.md
```

The Markdown artifact is useful for reviewing weights and bands beside the JSON. Treat it as a presentation of the saved model; use the saved model to execute the calculation. A human-readable report is most useful when it travels with the input rubric and a few known-answer cases.

Chapter 4

Learn a small integer rule

Manual weights express a chosen scoring policy. When labeled examples exist, fitting offers a second starting point: find weights that predict those labels. Urteil accepts a CSV with one row per case, columns matching rubric question names, and a binary target column. The fitted scorecard embeds the rubric, coefficients converted into integer points, thresholds, backend name, and backend metrics.

4.1 A larger worked example

The bundled startup generator has ten features. Founder-market fit and technical moat are binary. The remaining eight features use integers from 0 to 4. Three are risks: regulatory risk, capital intensity, and competitive crowding. A higher risk answer should contribute negatively if the fitted rule recovers the generator's direction.

Using F for founder-market fit, M for technical moat, P for customer pain, T for traction, Z for market size, E for sales efficiency, G for gross-margin potential, and R, C, K for the risks, the generator computes

$$\begin{aligned} L &= 5F + 4M + 2P + 3T + 2Z + 2E + G \\ &\quad - 3R - 2C - K + \varepsilon, \qquad \varepsilon \sim N(0, 2.5^2), \\ Y &= \mathbf{1}\{L \geq 15\}. \end{aligned}$$

The features are sampled separately using fixed probabilities in `urteil.scoring.examples`. Binary probabilities are 0.42 for F and 0.35 for M; each scale has its own categorical probabilities. The independent-feature construction is convenient for teaching and omits many dependencies likely to exist in real data.

The additive generator intentionally gives an additive model favorable conditions. Noise still creates overlapping classes. Good performance on this example demonstrates recovery of an invented relationship, not predictive validity for startup outcomes.

The book makes three independent seeded draws from this generator:

```
urteil scoring example startup-investment --output-dir train --rows 240 \
  --seed 7

urteil scoring example startup-investment --output-dir validation --rows \
  240 --seed 19
```

```
urteil scoring example startup-investment --output-dir test --rows 480 \
  --seed 23
```

The training draw has 240 cases with seed 7; validation has 240 with seed 19; test has 480 with seed 23. The CSVs, including the label column, are saved in separate directories. Their generated five-row sample files are previews of each split, not additional holdouts.

4.2 Create the manual benchmark first

The startup manual rule uses the following prespecified weights and the example rubric's bands at 8 and 16 points:

```
urteil scoring scorecard create train/rubric.json --output \
  startup-manual.json --weight founder_market_fit=3 --weight \
  technical_moat=2 --weight customer_pain=2 --weight traction_quality=3 \
  --weight market_size=1 --weight sales_efficiency=2 --weight \
  gross_margin_potential=1 --weight regulatory_risk=-2 --weight \
  capital_intensity=-2 --weight competitive_crowding=-1
```

This rule is a teaching benchmark, not the result of expert elicitation or an optimization procedure. It is evaluated using exactly the same test cases as the fitted rule.

4.3 What the fitting backend does

The normal `fit` command defaults to `-backend auto`. It attempts `imodels.SLIMClassifier`. If that attempt raises an exception, it falls back to the scikit-learn backend and records the exception text in `metrics.fallback_reason`. An explicit `-backend imodels` propagates the error instead of switching implementations.

This book explicitly selects `-backend sklearn` so that an optional imodels installation does not change the path taken. The command below is executed in the default build; a `-without-fit` build omits the fitted experiment.

```
urteil scoring fit train/rubric.json train/synthetic_startups.csv \
  --target invest --backend sklearn --max-points 10 \
  --output startup-fitted.json
```

The scikit-learn path constructs an L1-penalized logistic regression with `solver="liblinear"` and `C=0.5`. Logistic regression models a binary conditional probability through

$$\Pr(Y = 1 \mid x) = \frac{1}{1 + \exp[-(\beta_0 + x^\top \beta)]}.$$

An L1 penalty discourages large absolute coefficients and can set some coefficients to zero. This can reduce the number of active features, but Urteil exposes no hard constraint on the number of nonzero weights, no sign constraints, and no automatic model selection.

4.4 Integerization changes the model

After either backend returns coefficients, Urteil rescales them. If $A = \max_j |\beta_j| > 0$ and M is `-max-points`, then

$$w_j = \text{round}\left(M\frac{\beta_j}{A}\right), \qquad b = \text{round}(\beta_0). \tag{4.1}$$

All coefficients become zero when $A = 0$. Python's rounding is used. Choose a positive integer point budget; the current CLI accepts integer values without enforcing positivity.

The intercept is rounded *without* the coefficient scale factor. Thresholds are copied from the rubric; when the rubric has none, fitting inserts a positive band at zero. As a result, this transformation does not in general preserve the logistic classifier's boundary. Even uniformly rescaling all coefficients and the intercept would require a consistent threshold transformation; separate rounding adds further differences.

Nor is M a bound on the total score. A weight of 10 on a 0–4 scale can contribute 40 points. Features with different units have different attainable contributions even when their weights match. Use both the weight and the answer range when explaining influence.

The returned `training_accuracy` on the sklearn path describes the logistic estimator before integerization. It does not describe the final exported bands. The final card must be scored and evaluated directly, which is the subject of the next chapter.

4.5 The connection to SLIM and RiskSLIM

SLIM formulates sparse integer scoring as an optimization problem with explicit accuracy and sparsity objectives [2]. RiskSLIM develops optimized risk scores with a calibration objective [3]. Those methods motivate small, constrained, understandable models. They should not be equated with rounding a logistic regression.

Urteil's backend adapter extracts coefficients and then applies Equation (4.1), including to coefficients obtained through imodels. This book makes no claim that the resulting card inherits an integer solver's optimality guarantees. Inspect the saved backend and metrics, and validate the exported rule. Urteil does not export a calibrated probability; applying a sigmoid directly to its point total has no calibrated interpretation.

Chapter 5

Choose a threshold and test the final rule

A fitted rule is unfinished until we know how its decisions behave on cases that did not determine its coefficients or threshold. This chapter defines the evaluation executed by the book. These helpers live in `urteil.scoring.manual`; the general CLI does not currently provide separate evaluation or threshold-tuning commands.

5.1 Give each split one job

The training split supplies coefficients. The validation split chooses a single binary invest threshold. The test split measures performance after that threshold has been fixed. No test labels enter fitting or threshold selection.

Real data require a split appropriate to the intended use. If a company has multiple observations, put its observations together rather than letting the same company appear on both sides of a split. If deployment is prospective, a time-based holdout can better reflect the forecasting task. The book's independent draws are justified by its artificial generator; they are not a universal splitting recipe.

5.2 Count decisions, not just correct answers

For evaluation we define a positive prediction as classification exactly equal to `invest`. In the original three-band card, both `negative` and `investigate` are therefore negative predictions for this binary exercise. We are not evaluating whether the intermediate review band is operationally useful.

	Observed $Y = 1$	Observed $Y = 0$
Predict invest	True positive (TP)	False positive (FP)
Predict otherwise	False negative (FN)	True negative (TN)

The generated results retain all four counts, along with

$$\text{accuracy} = \frac{TP + TN}{n}, \qquad \text{precision} = \frac{TP}{TP + FP}, \qquad \text{recall} = \frac{TP}{TP + FN}.$$

Precision or recall is recorded as JSON null if its denominator is zero. The predicted positive rate, $(TP + FP)/n$, describes how many cases the rule advances. Accuracy alone can conceal a large workload or poor coverage of the positive class.

The evaluation uses `score_answers` and `classify` on the saved scorecard, not predictions from the temporary logistic estimator. It therefore includes integer rounding, the intercept, the specified bands, and the scorer's actual arithmetic.

5.3 Select the cutoff on validation data

For a candidate threshold t, the book forms a binary card with one band, `invest`, at t. It minimizes the validation error count

$$\mathcal{L}(t) = FP(t) + FN(t).$$

The candidates are the distinct observed validation scores and one point above their maximum. These cover every distinct binary partition available at an inclusive score threshold, including all-positive and all-negative predictions. Equal losses are resolved in favor of the larger threshold, a prespecified preference for fewer positive classifications among tied solutions.

The selected threshold is written into `startup-tuned.json`. This is a separate artifact; `startup-fitted.json` retains the original rubric bands. The sweep, objective, and tie-breaking policy are recorded in `threshold-sweep.json`.

The following Python API example shows the same operation performed by the builder. Run it from the output directory after a fitted build:

```
from dataclasses import replace
from pathlib import Path
from urteil scoring import Scorecard, Threshold
from urteil.scoring.fit import read_csv_rows
from urteil.scoring.io import read_json, write_json
from urteil.scoring.manual import select_threshold

card = Scorecard.from_dict(read_json(Path("startup-fitted.json")))
validation = read_csv_rows(Path("validation/synthetic_startups.csv"))
cutoff, curve = select_threshold(card, validation)
tuned = replace(card, thresholds=[Threshold("invest", cutoff)])
write_json(Path("startup-tuned.json"), tuned.to_dict())
```

Equal costs make the exercise easy to audit. If missing a positive case costs more than reviewing a negative case, choose and document a different loss, such as $c_{FP}FP + c_{FN}FN$, before inspecting the final test results. Urteil does not infer these costs from a rubric's outcome name.

5.4 Examine the size of the point system

The book also fits point budgets of 3 and 5 alongside its main budget of 10. Each alternative chooses its own threshold using validation data. These prespecified comparisons show how smaller point systems change sparsity and test performance. They do not use the test set to select a winner.

A true model-selection workflow would compare point budgets and thresholds on validation data, freeze the choice, and evaluate that single selection on untouched test data. If the book's test table prompts another modeling change, obtain a fresh holdout before reporting a final performance estimate.

5.5 Challenge the measurement process

The synthetic experiment has complete, consistently encoded inputs. A real deployment needs more than this clean-data exercise. Review cases near the chosen threshold and identify which uncertain answers would change the classification. Check whether omitted fields are being interpreted as zero. Compare error patterns across relevant case types and across batches of reviewers.

Inspect mislabeled outcomes as well as model errors. If historical labels record a committee's choices, the model predicts those choices. It does not automatically estimate eventual success. A label observed only for selected cases can omit the very cases the new process is meant to reconsider.

The book reports descriptive results for one fixed set of draws. It does not calculate confidence intervals, assess performance under distribution shift, or establish a fair or optimal decision policy. Those are additional analysis tasks, not properties supplied by a small integer representation.

Chapter 6

Read the results of this build

The tables in this chapter are generated from the saved artifacts. They are not values copied from a previous run. Package versions and hashes are recorded in `manifest.json`; the input draws are fixed by the seeds in Chapter 4. Backend and numerical-library changes can still alter fitted coefficients or borderline predictions.

6.1 The prespecified manual benchmark

This table uses the 480-case test draw and the manual startup rule. The positive event is the synthetic `invest` label; the prediction is classification equal to `invest`.

Manual rule, test data	Value
rows	480
tp	125
fp	4
fn	92
tn	259
accuracy	0.800
precision	0.969
recall	0.576
positive_rate	0.269

6.2 The fitted rule in this build

Backend: `sklearn.LogisticRegression.l1`. The logistic model's training accuracy is 0.921. The following tables evaluate the integer rule separately.

Feature	Manual	Fitted
founder market fit	3	7
technical moat	2	4
customer pain	2	4
traction quality	3	8
market size	1	4
sales efficiency	2	5
gross margin potential	1	0
regulatory risk	-2	-10
capital intensity	-2	-6
competitive crowding	-1	-4

6.3 One case, contribution by contribution

Feature	Answer	Weight	Points
founder market fit	1	7	7
technical moat	1	4	4
customer pain	4	4	16
traction quality	3	8	24
market size	4	4	16
sales efficiency	3	5	15
gross margin potential	3	0	0
regulatory risk	1	-10	-10
capital intensity	1	-6	-6
competitive crowding	2	-4	-8
Intercept			-2
Total			56

```
urteil scoring score startup-fitted.json train/sample_answers.json \
  --output startup-sample-score.json
```

6.4 Evaluate the rule that will actually run

Rule / data	TP	FP	FN	TN	Accuracy
Manual / test	125	4	92	259	0.800
Fitted / train	100	6	18	116	0.900
Fitted / test	174	27	43	236	0.854
Tuned / test	181	31	36	232	0.860

Validation selected threshold 15 using equal costs for false positives and false negatives. The tuned rule has a single invest band; all lower scores are negative. The original rules count only the invest band as a positive prediction.

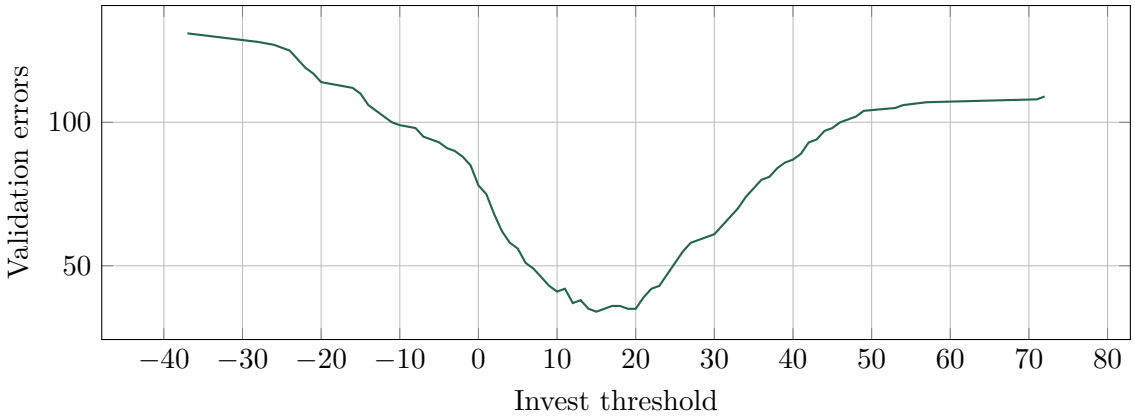

Figure 6.1: Threshold choice on the validation split only. Test labels do not enter this curve.

6.5 Sensitivity to the point budget

Max points	Nonzero weights	Threshold	Test accuracy
3	9	2	0.852
5	9	7	0.852
10	9	15	0.860

The three budgets were specified in advance. These test comparisons illustrate sensitivity; they do not select a winning budget. Choosing one from this table would require a fresh final test set.

6.6 How to interpret a change in accuracy

The manual and fitted scorecards have different origins. The manual rule uses illustrative weights. The fitted rule learns from the training draw, then inherits thresholds expressed on a different point scale. The tuned rule keeps its fitted weights and replaces its bands with a validation-selected binary threshold. Compare them with those distinctions in mind.

A large improvement after threshold selection need not mean the underlying ranking changed. The tuned card has exactly the same point score as the fitted card; only the classification cutoff differs. Conversely, a change in the point budget can alter both score ties and rankings because coefficients are rounded.

Inspect false positives and false negatives in `test-predictions.csv`. It records the row index in the test CSV, true label, manual score and class, fitted score and class, and tuned class. The numeric score is the same for the last two rules. Together with `test/synthetic_startups.csv`, these rows let a reader reconstruct individual errors.

6.7 Use the figures for diagnosis

In a fitted build, Urteil's `visualize` command also writes three SVG files under `figures`: weights, score distribution, and feature correlations. Those are companion artifacts for browser inspection.

The PDF uses a TeX-rendered threshold curve, so it does not need an SVG conversion tool or shell escape.

The visualizations use the training CSV. They describe that sample and its fitted rule. A coefficient plot is not a ranking of causal effects, and a training score histogram is not a held-out performance measure. The confusion counts above are the direct checks of the final classifications.

Chapter 7

Carry the rule into a survey

Urteil's local JSON scorer answers a narrow question: given these answers, what are the points and the class? Survey export carries the same intended rule into an EDSL artifact. It also introduces a distinction between collecting answers and scoring already structured cases.

7.1 Two survey modes

A *scoring* survey contains two compute questions. Their names are `rudin_score` and `rudin_classification`; both use EDSL's `QuestionCompute` type. The survey reads rubric fields directly from a scenario. It is appropriate when someone has already encoded the answers.

A *respondent* survey asks the rubric questions first, then computes the score and class from those answers. It is appropriate when the answers must be collected through the survey. The person or model supplying an evidence judgment is still responsible for its meaning; a compute question only executes the scoring arithmetic.

In current Urteil, `-mode` affects only `-format ep`. The `spec`, `edsl-py`, and `edsl-json` formats produce respondent-style question sequences regardless of that flag.

7.2 Exports executed by this book

These commands create a dependency-free JSON specification and Python source:

```
urteil scoring export-survey grant/scorecard.json --format spec --output \
  grant/survey-spec.json

urteil scoring export-survey grant/scorecard.json --format edsl-py \
  --output grant/respondent_survey.py
```

The JSON spec contains the question descriptions and compute templates. It is not itself a serialized EDSL Survey. The Python file imports EDSL when a user executes it. Writing that source does not require EDSL and does not run a survey. This is why the book can build offline with only scikit-learn as an optional Python dependency.

7.3 Create native EDSL artifacts

The following commands are optional handoff examples, not commands executed by the default book build. Install EDSL in your active environment first:

```
python -m pip install 'urteil'
urteil scoring export-survey grant/scorecard.json \
  --format ep --mode respondent --output grant/respondent.ep
urteil scoring export-survey startup-manual.json \
  --format ep --mode scoring --output startup-scoring.ep
urteil scoring export-scenarios train/sample_startups.csv \
  --output startup-scenarios.ep
```

The `.ep` files contain serialized EDSL objects. The first survey asks grant questions; the second scores numeric startup scenario fields. The sample scenarios are the first five training cases, suitable for an integration check, not a statistical evaluation.

Urteil's classification compute template recomputes the total from the inputs. It does not depend on the preceding compute answer being parsed as a particular numeric type. Both compute questions use the same rule specification.

7.4 Run a local scoring job explicitly

For the numeric startup CSV, Urteil can write a ready-to-run Python job:

```
urteil scoring export-job startup-manual.json train/sample_startups.csv \
  --format edsl-py --output startup-job.py
python startup-job.py
```

The generated job loads scenarios, builds a compute-only survey, and invokes EDSL with remote inference and remote cache disabled. This optional runtime path requires a compatible EDSL installation. The book's builder does not execute it. Exporting a respondent survey does not collect answers from people or language models, and it does not publish anything.

7.5 Check parity on the answers you support

Before using a survey in a larger workflow, compare its outputs to local scoring for known cases, including values exactly at thresholds. Use canonical numeric 0/1 answers for yes/no questions and integer scale values. The local scorer recognizes strings such as `"yes"`, whereas exported templates use an integer filter; noncanonical strings need not agree across runtimes.

With no thresholds, both the local scorer and exported compute templates return `negative`. Supply explicit thresholds to define useful decision bands.

A short parity check should include a low-score case, a case at each threshold, and a case with multiple selected risks if checkbox questions are used. Compare labels as well as totals. Treat a difference as an integration error to resolve, not as harmless formatting.

7.6 Separate assessment from arithmetic

For an application narrative, one workflow might ask a reviewer to assign rubric answers, then apply the scorecard. Replacing the reviewer with a language model changes the assessment process, even if the point calculation stays fixed. Evaluate those extracted answers independently against the intended rubric. The synthetic examples in this book exercise arithmetic and fitting; they do not establish the accuracy of automated narrative assessment.

Chapter 8

Maintain a rule people can inspect

A scorecard becomes useful when it can be rerun, questioned, and changed without losing the meaning of earlier decisions. Preserve the measurement specification and decision rule together. The embedded rubric in Urteil's scorecard helps, but it does not capture the entire surrounding process.

8.1 Keep a release bundle

A practical release contains the scorecard JSON, its readable scoring report, a few complete input cases with expected outputs, and a short explanation of the outcome and thresholds. If weights were fitted, also preserve the training-data provenance, the holdout design, the validation objective, the backend version, and the evaluation results. Document any manual changes made after fitting.

Urteil's book provides a small example of such a bundle. `commands.json` records actual CLI invocations and JSON envelopes. `results.json` records numeric summaries. `manifest.json` records versions and hashes of generated inputs, sources, and result artifacts before PDF compilation. The manifest is a build record, not a full software lockfile or a cryptographic attestation of who ran the study.

8.2 Review a disputed case

Begin with the case's stored answers. Is the disagreement about the evidence, the rubric's interpretation, the point tradeoff, or the cutoff? Each suggests a different correction. A mistaken answer should be corrected as a data issue. An ambiguous question may need new wording and renewed reviewer calibration. A point change changes the policy or model and should trigger another evaluation.

Retain the original result when recording a correction, together with the corrected inputs and rule version. Otherwise it becomes impossible to explain why a later rerun produces a different classification. The `urteil scoring apply` command preserves each scoring application under `.urteil/scores/`, including its rule, source ratings, contributions, and manifest. Standalone JSON scoring still leaves history management to the caller.

8.3 Respond to changes in the population

Even a correctly implemented rule can become less useful when inputs or outcomes change. Compare answer distributions, the frequency of unknowns, the rate of each classification, and observed errors over time. A larger positive rate could reflect better cases, changed reviewer behavior, or a broken data mapping.

Threshold adjustment can change workload without repairing an obsolete ranking. Refitting can change rankings without repairing a poorly measured input. Inspect which part of the process changed before choosing the remedy.

8.4 Extend the example deliberately

To adapt this book to another domain, first replace the fictional outcome and rubric. Decide whether weights express a proposed policy or predict an observed label. Build a small manual rule with known-answer cases. For fitted models, prepare numeric features consistently, hold out appropriate cases, and evaluate the actual integer card. Explicitly choose each classification threshold on the intended point scale.

Several useful capabilities remain outside Urteil's current implementation: automated missing-data handling, hard sparsity or sign constraints, learned interactions, probability calibration, and subgroup evaluation. Saved-source scoring supplies an audit trail, but this does not validate the substantive criterion judgments. The existing JSON and Python boundaries let a project add those analyses, but the resulting claims must be supported by the added work.

8.5 Exercises

1. Change the grant evidence weight from 2 to 1 in a new scorecard. Recompute the worked case and the maximum possible score. Does the advance band still reflect the intended policy?

2. Give an explicit point mapping to evidence so that levels 0 and 1 both contribute zero. Verify that the mapping overrides the numeric weight. Explain why this differs from changing a calibration example.

3. For the startup study, minimize $FP + 2FN$ on validation data. Compare its selected threshold and predicted positive rate with the equal-cost rule. Keep test labels out of the selection step.

4. Remove the risk fields from a copy of the grant answers. Explain the change in score. Design an input-completeness check that prevents an unknown risk from being interpreted as absent.

5. Repeat the startup experiment with a new test seed while keeping the fitted weights and validation-selected threshold fixed. Compare error counts rather than choosing whichever seed gives the most favorable accuracy.

Appendix A

Artifacts and command reference

A.1 What a build writes

All paths below are relative to the directory passed to `-output-dir`.

`manual.tex`, chapter files Editable LaTeX source. Recompile from this directory with `pdflatex -no-shell-escape manual.tex` three times after editing prose.

`manual.pdf` Compiled book, present when `-pdf` succeeds.

`generated/` Executed command listings, computed tables, and version information included by the LaTeX source.

`grant/` Incrementally constructed rubric, manual scorecard, sample answers and score, Markdown presentation, JSON survey spec, and respondent survey Python source.

`train/`, `validation/`, `test/` Independent seeded startup draws, rubric, sample answers, and preview cases.

`startup-manual.json` Prespecified startup benchmark with bands at 8 and 16.

`startup-fitted.json` Fitted integer card with the original bands, present in a fitted build.

`startup-tuned.json` Same fitted weights with one validation-selected invest threshold.

`startup-points-*.json` Alternative budgets and their separately saved threshold-adjusted cards.

`threshold-sweep.json` Candidate cutoffs and validation metrics for the main ten-point card.

`test-predictions.csv` Case-level test predictions for the manual, original fitted, and tuned rules.

`results.json` Machine-readable summaries used to render this book's results.

`commands.json` Actual Urteil argument vectors and returned JSON envelopes. Python-side threshold selection is recorded separately in the sweep and results artifacts.

`manifest.json` Python and modeling-library versions, split seeds and sizes, and SHA-256 hashes of pre-compilation build artifacts.

figures/ Training-data SVG diagnostics from `urteil scoring visualize` in a fitted build.

Fitted artifacts are absent when `-without-fit` is selected. The generated text says so explicitly. A compilation error retains sources and results together with the TeX logs for inspection.

A.2 CLI map

Canonical `urteil scoring` commands return Urteil JSON envelopes with `ok`, a command name, and data or errors. The deprecated `rudin` compatibility command retains its legacy `status/data` envelope. The top-level `-help` and command-specific help list all flags. Errors in command execution return a nonzero exit status; argument parsing is handled by argparse.

`rubric create`, `rubric show` Start an empty rubric or inspect one.

`question add` Append a typed question; duplicate question names are rejected.

`criterion example add` Append an answer/score/rationale annotation to an existing question.

`threshold add` Append a label and inclusive minimum score to a rubric. Use distinct minimums for unambiguous bands.

`scorecard create` Build a manual card with `-weight`, `-point`, and optionally `-intercept`.

`fit` Learn weights from a numeric-feature CSV and binary target. Optional modeling dependencies are required.

`score`, `present` Score one answer object or render a Markdown rule.

`export-survey` Write a spec, Python source, EDSL JSON, or an EDSL `.ep` Survey.

`export-scenarios`, `export-job` Prepare a ScenarioList or scoring-job source/JSON.

`visualize` Write SVG weight, score-distribution, and correlation diagnostics.

`example startup-investment` Generate the synthetic rubric, data, and sample files with configurable row count and seed.

`configure`, `apply` Turn a codebook into Urteil rating questions or score a saved label source with an audit trail.

`docs manual` Run this worked study and export its book; optionally compile a PDF.

Appendix B

Implementation details that affect interpretation

B.1 Input coercion

Local weighted scoring converts booleans to 0/1, recognizes common yes/no strings, reads numeric strings, treats an unrecognized nonempty string as 1, and treats a list as its length. It then converts the value to an integer before multiplication. A scale answer of 2.9 therefore contributes as 2, not 2.9. These permissive conversions are not a substitute for input validation.

Fitting uses the shared coercion helper but does not apply that final integer conversion to its feature matrix. Fractional inputs can therefore produce a discrepancy between what the backend fits and what the local card scores. The worked study avoids this issue by using integer inputs throughout.

A categorical string is not automatically one-hot encoded by fitting. Unknown text commonly becomes 1; a checkbox CSV cell containing text is not converted into a list of categories. If categories must be learned, construct explicit numeric indicator columns and corresponding rubric questions before fitting. Manual answer-point maps are the native way to assign category-specific contributions without fitting.

B.2 Study ratings and legacy standalone scoring

Use `urteil scoring configure rubric.json -fields memo` in an initialized Urteil project to include criterion definitions and anchors in LLM prompts. Collect ratings through the ordinary sample, plan, execute, and record workflow. Then `urteil scoring apply scorecard.json -source SOURCE` computes scores locally, rejects missing or invalid required ratings, and archives contributions. Verified codebook or prompt changes require new ratings; new weights or thresholds can reuse the original ratings. Imported sources without a verified codebook carry an explicit warning. Read `urteil docs show scoring` for the full protocol.

B.3 Standalone validation is limited

The CLI checks some construction errors, including duplicate question names and manual weights referencing unknown questions. It does not enforce a full schema on every loaded JSON object, guarantee complete CSV feature columns, reject all out-of-range answers, or verify binary targets comprehensively. Missing feature columns during fitting default to zero. Check data before modeling.

Use one weight per question, one map per question, and unique threshold cutoffs. Hand-edited JSON with duplicate entries can behave differently across scoring and export paths. The book uses canonical artifacts produced by the CLI.

Manual `-weight` values and the card intercept are integers. Answer-specific points may be fractional, so a manual card can return a noninteger score despite the integer return annotation in the current scorer. The book's examples use integer points only. A legacy `FeatureWeight.intercept` field is serialized but is not used by scoring; use the card-level intercept.

B.4 Dependencies and reproducibility

Core rubric construction, manual scoring, specs, and source-code export need only Python. The `manual` extra adds scikit-learn for this book. The `modeling` extra adds both scikit-learn and imodels for general fitting. EDSL is a base dependency of urteil, used for native object construction and serialization; local scoring performs no model calls. PDF compilation is a separate system dependency.

The book fixes seeds and backend selection. It records package versions but does not pin the solver's full execution environment. Small numerical changes may affect rounding or scores close to a boundary. Preserve an environment lockfile in projects requiring exact regeneration, and compare saved weights and predictions when updating dependencies.

To review the implementation behind a claim, begin with `rubric.py` and `model.py` for the data structures, `fit.py` for fitting and integerization, `scoring.py` for local decisions, `export.py` for survey templates, and `manual.py` for this book's orchestration and evaluation. These are modules in the installed `urteil.scoring` package.

Bibliography

[1] Cynthia Rudin (2019). *Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead.* Nature Machine Intelligence, 1, 206–215. https://doi.org/10.1038/s42256-019-0048-x.

[2] Berk Ustun and Cynthia Rudin (2016). *Supersparse linear integer models for optimized medical scoring systems.* Machine Learning, 102, 349–391. https://doi.org/10.1007/s10994-015-5528-6. Author manuscript: https://arxiv.org/abs/1502.04269.

[3] Berk Ustun and Cynthia Rudin (2019). *Learning Optimized Risk Scores.* Journal of Machine Learning Research, 20(150), 1–75. https://jmlr.org/papers/v20/18-615.html.