DATA SCIENCE & ML

Feature Engineering for Machine Learning

Crafting, selecting and serving features that move metrics

Houssam Kodad

PDF · DATAFORGE BOOKS

Questions about this sample or the full edition: **support@dataforgebooks.com**

CONTENTS

Table of Contents

01 **Why Features Decide the Outcome** 1

Model capacity vs signal
The notebook-to-production gap
A workflow for the book

02 **Numerical Features Done Right** 27

Scaling and transforms
Binning and outliers
Interactions and ratios

03 **Encoding Categorical Variables** 53

One-hot vs ordinal
Target and frequency encoding
High-cardinality strategies

04 **Time, Dates and Window Features** 79

Calendar and cyclical features
Lag and rolling-window features
Avoiding lookahead bias

05 **Text and Embeddings as Features** 105

Bag-of-words to TF-IDF
Pretrained embeddings
Dimensionality reduction

06 **The Leakage Traps** 131

Target leakage in practice
Leakage through preprocessing
Leak-proof cross-validation

07 **Selecting What Matters** 157

Filter, wrapper and embedded methods
Permutation importance
Pruning redundant features

08 **Training/Serving Parity** 183

Why features drift apart
Shared transformation code
Point-in-time correctness

09 **Feature Stores in Production** 209

Offline and online stores
Backfills and freshness
Governance and reuse

CHAPTER 01

Why Features Decide the Outcome

Practitioners learn a humbling lesson early in their careers: a better feature beats a fancier model far more often than the reverse. The model architecture you choose matters, but it can only amplify the signal already present in your features. If the signal is not there, no amount of tuning, no deeper network, no cleverer ensemble will conjure it from nothing.

This chapter sets up the discipline that the rest of the book develops. We examine the relationship between a model's capacity and the signal in its inputs, name the failure that quietly breaks more production systems than any other — training/serving skew — and lay out the workflow we follow throughout. The emphasis is relentlessly practical: features that are predictive, reproducible, and actually available at the moment a prediction must be made.

That last constraint is where most notebook-grade feature engineering falls apart, and it is why this book exists. A feature that is trivial to compute over a full historical dataset can be impossible or wrong to compute for a single live request. Keeping those two worlds consistent is a central theme, and we introduce it here so it colours everything that follows.

Model Capacity vs Signal

Every model has capacity: the complexity of the patterns it can represent. A linear model has little; a deep gradient-boosted ensemble has a great deal. But capacity only helps if there is signal to learn. Pour more capacity onto features that carry no predictive information and you do not get accuracy — you get overfitting, a model that memorises the noise in your training set and falls apart on new data.

Feature engineering is the work of getting signal into a form the model can actually use. Sometimes that means creating a ratio or interaction the model could never derive on its own from raw columns. Sometimes it means encoding a category so its structure becomes visible. The model is the easy part once the features carry the signal — which is precisely why this book spends its pages on the features.

The Notebook-to-Production Gap

The feature that wins in your notebook can fail in production for one insidious reason: it was computed differently at training time than it is at serving time. A rolling average that accidentally included future rows, a category encoding fit on the entire dataset, a join that is point-in-time correct offline but not online — these are all training/serving skew, and they are the most common cause of

models that look brilliant offline and disappoint in the wild.

Closing this gap is a central theme of the book, and the fix is architectural rather than clever. Share the transformation code between training and serving so there is one definition of each feature. Enforce point-in-time correctness so a feature only ever uses data that was available before the event it describes. And treat features as a versioned, governed asset rather than as glue code buried in a notebook nobody will read again.

```
# Leakage hiding in plain sight: this average peeks at the future.
df["avg_spend"] = df.groupby("user")["spend"].transform("mean")

# Point-in-time correct: only data available before each event.
df = df.sort_values("ts")
df["avg_spend"] = (df.groupby("user")["spend"]
                     .apply(lambda s: s.shift().expanding().mean()))
```

Numerical Features

Numerical columns look the simplest and hide the most subtlety. Scaling matters for models sensitive to magnitude, and the choice between standardisation and normalisation depends on the algorithm and the presence of outliers. Skewed distributions — income, counts, durations — often benefit from a log or other transform that spreads their mass so the model can see structure that was crushed into one corner.

Beyond transforms, the real value often comes from combinations. A ratio of two columns, a difference, an interaction term, or a binned version of a continuous variable can expose a relationship the model would struggle to find unaided. The judgement of which derived numerical features are worth creating — and which merely add noise — is a skill we develop with worked examples throughout the book.

Encoding Categorical Variables

Categorical features are where leakage and cardinality problems love to hide. One-hot encoding is safe and interpretable but explodes in width when a column has thousands of values. Ordinal encoding is compact but imposes a false ordering. Target encoding — replacing a category with a statistic of the target — is powerful and compact, and also the single most common source of subtle leakage in real projects.

The danger with target encoding is that, done naively, it lets each row peek at its own label through the category statistic, inflating offline scores and collapsing in production. Done correctly, with out-of-fold computation, it is one of the most effective tools for high-cardinality features. We treat encoding carefully because getting it wrong is so easy and so quietly destructive.

```
# Target encoding must be computed out-of-fold to avoid leakage:
# each row's encoding is learned from OTHER rows, never itself.
from sklearn.model_selection import KFold

enc = np.zeros(len(df))
for tr, va in KFold(5, shuffle=True, random_state=0).split(df):
    means = df.iloc[tr].groupby("category")["target"].mean()
    enc[va] = df.iloc[va]["category"].map(means)
df["category_te"] = enc
```

Time, Dates and Window Features

Temporal data is a goldmine and a minefield. From a single timestamp you can derive calendar features — hour of day, day of week, month, holiday flags — that capture seasonality the raw value hides. Cyclical encodings let a model understand that hour 23 is close to hour 0. These features are cheap and frequently among the most predictive in the whole set.

The minefield is lookahead bias. Lag features and rolling-window aggregates — the average of the last seven days, the count in the previous hour — are enormously useful, but every one of them must be computed using only data from strictly before the prediction point. A window that includes the current or future rows leaks the answer. This is the temporal face of the training/serving gap, and we handle it with great care.

Selecting What Matters

More features are not better features. Redundant and irrelevant inputs slow training, inflate variance, and make models harder to maintain and explain. Selection methods fall into three families: filters that score features individually, wrappers that evaluate subsets by training models, and embedded methods where the model itself performs selection as it fits.

In practice the most reliable signal comes from permutation importance — measuring how much performance drops when a feature's values are shuffled — and from simply removing features and watching validation error. The goal is a compact set of features that each earn their place, because a model you can understand and a feature set you can maintain are worth more than a marginal, fragile gain from a hundred opaque inputs.

Toward a Feature Store

Everything in this chapter points at one organisational idea: features are an asset worth managing centrally. A feature store is infrastructure that computes features once, serves them consistently to both training and inference, and lets teams discover and reuse each other's work instead of reinventing it. It is the systematic cure for training/serving skew.

We do not begin with the feature store, because you should understand the problems it solves before adopting it. But it is the destination, and the patterns we build in the coming chapters — shared transformations, point-in-time correctness, versioned definitions — are exactly what a feature store formalises. Keep it in mind as the place this all leads.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 248 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com