
MACHINE LEARNING FOR MARKET PREDICTION

**Feature Engineering and Backtesting
Without Lookahead Bias in Python.**

Weston Ashgrove

Copyright © 2026 by Weston Ashgrove

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

This book is intended for informational and educational purposes only. While every effort has been made to ensure the accuracy and completeness of the information contained herein, the author and publisher make no representations or warranties with respect to the accuracy or completeness of the contents and specifically disclaim any implied warranties. The modeling techniques, backtesting methodologies, and code examples described in this book are provided for educational purposes and should be independently tested and validated before use in any live trading or investment decision.

Nothing in this book constitutes financial or investment advice. Past backtest performance does not guarantee future results, and any model or strategy described may perform differently under live market conditions than in historical testing. Readers are solely responsible for independently validating any model before relying on it for actual trading or investment decisions.

All trademarks, service marks, and trade names referenced in this book are the property of their respective owners and are used for identification purposes only. Use of these names does not imply endorsement.

First Edition: 2026

What You'll Build

By the end of this book, you will have built a complete, leakage-free research pipeline: starting from raw price and alternative data, through feature engineering that survives real-world deployment, to a labeled, walk-forward-validated model with honest, regime-tested performance metrics. Every stage is instrumented against the three failure modes that kill most trading models before they ever go live: lookahead bias, survivorship bias, and label leakage. The final case study walks the entire pipeline end to end on real data, producing a model whose backtest results you can actually trust, along with a clear handoff point for deployment.

Prerequisites

You should be comfortable writing Python and working with pandas for data manipulation. Basic statistics (mean, standard deviation, correlation) is assumed. No prior machine learning experience is required. Every ML concept used, from cross-validation to SHAP values, is introduced from first principles in the context of market prediction specifically, not as a generic ML tutorial.

How to Use This Book: The Leakage Audit Ledger

Every chapter in this book ends with a Leakage Audit Ledger, a short running record of every place lookahead bias, survivorship bias, or label leakage could have crept into the pipeline at that stage, and exactly how it was closed off.

This exists because leakage rarely arrives as one dramatic mistake. It accumulates in small decisions: a rolling window that peeks one bar too far, a data source that quietly survivorship-biases its universe, a label built from information that would not have been available at prediction time. By the end of the book, the ledger becomes a complete audit trail of the pipeline you built, and a checklist you can apply to any future model, in or outside the exact techniques covered here.

Each entry follows the same short format:

- **Where:** the pipeline stage where the risk was introduced
- **Risk:** what kind of leakage was possible and why
- **Fix:** the specific technique used to close it

By Chapter 10, the ledger doubles as a complete leakage-prevention checklist for your own future projects.

Chapter 2

Sourcing and Cleaning Market Data

Choosing a Data Source

Do not build your feature pipeline before you have confirmed your data source is point in time. Everything else in this chapter assumes that decision has already been made correctly, and getting it wrong here invalidates work you have not done yet.

A data source is point in time when it can tell you, for any historical date, exactly what was knowable as of that date, not what is true now. This distinction matters differently across the three broad categories of market data, and each category fails in its own characteristic way if you choose the wrong source.

Price data is the easiest category to get right and the easiest to get wrong through convenience. Free and low cost price data providers typically deliver adjusted closing prices, adjusted for splits and dividends using the adjustment factor as it stands today, applied backward across the entire history. This is usually acceptable for backtesting price momentum and mean reversion strategies, because the adjustment is mathematically consistent across the full series and does not depend on information a trader could not have had. It becomes a problem the moment you mix adjusted and unadjusted series, or the moment your strategy logic depends on the actual dollar price level rather than the return series, such as a strategy with a fixed dollar stop loss. Know which version of price your provider gives you, and pick one version and use it consistently through the entire pipeline, because a backtest that silently switches between adjusted and raw prices at different stages will produce return

calculations that do not correspond to any trade a real account could have executed.

Volume data carries a subtler risk. Reported daily volume figures are sometimes revised after the fact, particularly around corporate actions, block trades reported late, or data vendor corrections. If your provider gives you a single volume figure per historical date with no revision history, you are implicitly assuming the number never changed, which is usually a reasonable assumption for volume specifically, but it is an assumption you should confirm rather than inherit silently. The bigger volume related risk is liquidity survivorship, where your dataset only reports volume for securities that currently trade with reasonable liquidity, quietly excluding thinly traded or now delisted names from the historical volume record even when their price history is present. A strategy that filters by minimum average volume needs volume data with the same point in time integrity as the price data it is paired with.

Fundamental data is where source selection has the largest impact on your results, because the gap between "reported today" and "true as of the historical date" is largest here. Company financials get restated. A quarterly earnings figure released initially, then revised in a subsequent filing due to an accounting correction, a merger adjustment, or a regulatory requirement, will often be delivered by data vendors as a single, most current value stamped with the original report date. If you join this to price data using that report date, you are giving your model a version of earnings that the market did not have on that date, which is the fundamental data equivalent of the lookahead bias mechanism from section 1.2, entering through the data source rather than through your own feature code.

python

```
import pandas as pd

# Ordinary vendor join, using report date as the join key
prices = load_price_data()
fundamentals = load_fundamentals() # contains only latest
revised values
merged = prices.merge(
    fundamentals, left_on="date", right_on="report_date",
    how="left"
```

```

)
# merged now contains revised earnings figures joined to price
dates
# that occurred before the revision existed

# Point in time correct join, using actual public release timestamp
fundamentals_pit = load_fundamentals_point_in_time() #
includes release_date
# and preserves each historical revision as a separate row
merged_pit = pd.merge_asof(
    prices.sort_values("date"),
    fundamentals_pit.sort_values("release_date"),
    left_on="date",
    right_on="release_date",
    direction="backward",
)

```

`merge_asof` with `direction="backward"` is doing real work in the second version. For every price date, it pulls the most recent fundamental value whose release date is at or before that price date, which is exactly the information a trader would have actually had. The first version, the ordinary merge on report date, has no concept of when a value became public, and will happily attach a 2024 restated 2021 earnings figure to a 2021 price row.

Choosing a data source therefore comes down to three questions you should be able to answer before you write a single feature. Does this source preserve delisted and removed securities, so you avoid the survivorship bias covered in section 1.3. Does this source distinguish between when a value was reported and when it was actually released to the public, so you can join on release timestamp rather than report date. Does this source version its historical values, so a restated figure does not silently overwrite the value that was actually knowable at the time.

Red flag. A reader submits a fundamentals based strategy sourced from a free API with no documentation of a point in time methodology, and the backtest performance is strong specifically around earnings announcement dates. That combination is common enough to check first, because free fundamental data

feeds are the single most frequent source of the report date versus release date leak described above, and the strategy's apparent skill at trading around earnings is often just the model seeing the final, corrected number early.

None of this means you need the most expensive data provider available. It means the provider you choose needs to answer the three questions above correctly, and if it does not, you need to build the point in time correction yourself, using the join logic shown here, before any feature touches the data. Section 2.2 continues from here, handling the missing data and corporate action adjustments that even a properly sourced dataset will still require.

Handling Missing Data and Corporate Actions

A missing value in a price series is not an inconvenience to be filled in. It is information about why the data is missing, and treating every gap with the same fill method destroys that information before your model ever sees it.

Start with the two structurally different reasons a value can be absent, because the correct handling depends entirely on which one you are facing. The first is a calendar gap, a date where the market was simply closed, a weekend or a holiday, and there is no trading to record. The second is a data gap, a date where the market was open, trading occurred, and your specific data source simply failed to capture or deliver a value for that security on that date. Forward filling across a calendar gap is harmless, because nothing happened during that gap that your model needed to know about. Forward filling across a data gap manufactures a price where none should exist, and if that manufactured flat price sits inside a rolling volatility or momentum window, it quietly compresses the computed volatility or dampens the computed momentum for every window that includes it, in a way that has nothing to do with the actual security's behavior.

python

```
import pandas as pd
```

```
def classify_gaps(prices: pd.Series, trading_calendar:
pd.DatetimeIndex) -> pd.Series:
    full_index = trading_calendar[
        (trading_calendar >= prices.index.min())
        & (trading_calendar <= prices.index.max())
```

```

]
reindexed = prices.reindex(full_index)
missing = reindexed.isna()
# every missing entry here occurred on a day the market was
open,
# so it is a data gap, not a calendar gap
return missing

```

Reindexing against an actual trading calendar rather than a plain date range is the step that makes this classification possible. A plain `pd.date_range` includes weekends, which then look identical to data gaps once you check for missing values. An exchange trading calendar excludes weekends and holidays by construction, so anything still missing after reindexing against it is a genuine data gap that needs a deliberate decision, not an automatic fill.

For genuine data gaps, forward filling is defensible only for short gaps, one or two sessions, and only for the price level itself, never for derived features computed over a window that spans the gap. A five day gap in a small cap name is not noise to be smoothed over. It is frequently a trading halt, and a trading halt is itself informative, often preceding a material announcement. Filling through it with the last known price tells your model that nothing happened during a period when something significant likely did.

python

```

def safe_forward_fill(prices: pd.Series, max_gap: int = 2) ->
pd.Series:
    filled = prices.ffill(limit=max_gap)
    still_missing = filled.isna()
    if still_missing.any():
        # gaps longer than max_gap are left as NaN, not filled,
        # forcing an explicit downstream decision rather than a silent
one
        pass
    return filled

```

Leaving longer gaps as NaN rather than filling them is intentional. A rolling window function that encounters NaN values will propagate them rather than silently computing a window statistic over fabricated flat prices, and that

propagation is a feature, not a bug, because it forces you to decide explicitly whether a security with a long trading halt belongs in your universe on that date at all.

Corporate actions are the second major source of gaps and distortions, and they fail in a specific, recognizable way when handled incorrectly. A two for one stock split doubles the share count and halves the price, with no change in the company's actual value. A price series that is not split adjusted will show what looks like a fifty percent single day drop on the split date. Any momentum or return based feature computed across that date will register a catastrophic loss that never actually happened to anyone holding the stock. Dividends create a smaller version of the same problem, a price drop on the ex dividend date that reflects the value paid out to shareholders rather than a decline in company value.

The fix is split and dividend adjustment, but section 2.1 already flagged the risk buried inside this fix. Adjustment factors are typically computed backward from the present, meaning the adjustment applied to a price from five years ago incorporates every split and dividend that has happened since, including ones that had not occurred yet as of that historical date. This does not create the same kind of lookahead bias as a fundamentals leak, because the adjustment is a pure mathematical transformation of price level and does not inject outside information about future company performance. It does mean you cannot treat an adjusted price level as if it were the literal price a trader would have seen on a screen on that historical date, and any feature that depends on the actual nominal price level, such as a fixed dollar stop or a round number effect, needs the unadjusted price series specifically, kept and labeled separately from the adjusted one used for return calculations.

python

```
def apply_split_adjustment(
    prices: pd.DataFrame, splits: pd.DataFrame
) -> pd.DataFrame:
    adjusted = prices.copy()
    cumulative_factor = 1.0
    for date, ratio in splits.sort_index(ascending=False).iteruples():
        adjusted.loc[:date, "close"] /= cumulative_factor
        cumulative_factor *= ratio
    return adjusted
```

Working backward from the most recent split, as this function does, is the standard mechanism, and the resulting series is correct for computing returns across the full history. Keep the unadjusted close column alongside the adjusted one in your processed dataset rather than discarding it, because you will need both, and reconstructing unadjusted prices later from an already adjusted file is unreliable.

Red flag. A reader submits a strategy where a single day contributes an outsized share of total backtested return or loss, and that day coincides with a known split or large dividend date for the security involved. This is one of the fastest checks available during a review, because an unadjusted price series will produce exactly this signature, a single implausible one day move that dwarfs every other day in the sample, and it is a data handling error rather than a market event almost every time it appears.

The underlying principle across both gaps and corporate actions is the same one from section 1.2. Every transformation you apply to raw data changes what the resulting values actually represent, and if you do not know precisely what a given number represents, whether it is a genuine missing session or a manufactured fill, an adjusted price or a raw one, you cannot know what your model is actually learning from it. Section 2.3 extends this same discipline to a related problem, aligning timestamps across multiple data sources that were not built to agree with each other.

Aligning Timestamps Across Data Sources

Two data sources that both claim to describe the same date rarely describe the same moment in time, and merging them on the date alone assumes they do. The mechanism behind this is that different data types get finalized at different points in the trading day, and different vendors stamp their records using different conventions for what "the date" means. A daily close price for a US equity is not final until the market closes at 4pm Eastern. A news sentiment score attached to an article published at 9am is available hours before that close. An economic data release, such as a jobs report, is published at a fixed time that may fall before, during, or after the trading session depending on the exchange and the release calendar. If you join all three of these to the same calendar date and treat them as simultaneously available, you have built in an assumption about information timing that is false for at least one of the three, and the direction of the error usually favors the model, because same day information

typically correlates with same day price moves for reasons that have nothing to do with genuine predictive power.

Consider a strategy using overnight news sentiment to predict the next trading day's return. If your sentiment feed timestamps an article with the calendar date of publication and your price feed timestamps a daily bar with the same calendar date for the session that closed that day, joining on date looks correct and is actually wrong for any article published after the prior session's close but before the current session's close. That article is legitimately available before your target session opens, and using it is fine. The problem is the reverse case, an article published intraday, after the price move it appears to be predicting has already partly happened. A same date join cannot distinguish between "sentiment published at market open, predicting the day's return" and "sentiment published at market close, reacting to the day's return." Both get treated identically, and the second case is not lookahead bias in the strict sense from section 1.2, because the sentiment did genuinely exist by end of day, but it is a timing error that inflates your apparent predictive power by including reactive information alongside predictive information.

python

```
import pandas as pd
```

```
# Naive join, both sources reduced to a bare calendar date
```

```
prices["date"] = prices["timestamp"].dt.date
```

```
sentiment["date"] = sentiment["published_at"].dt.date
```

```
merged = prices.merge(sentiment, on="date", how="left")
```

```
# an article published at 3:55pm is now treated identically
```

```
# to one published at 8:00am, even though only one of them
```

```
# was available before the session's price action occurred
```

```
# Timestamp aware join, using a defined decision point
```

```
def align_to_decision_time(
```

```
    prices: pd.DataFrame, sentiment: pd.DataFrame, session_open:
    str = "09:30"
```

```
) -> pd.DataFrame:
```

```
    prices = prices.sort_values("timestamp")
```

```

sentiment = sentiment.sort_values("published_at")
prices["decision_cutoff"] = (
    prices["timestamp"].dt.normalize()
    + pd.Timedelta(session_open + ":00")
)
merged = pd.merge_asof(
    prices,
    sentiment,
    left_on="decision_cutoff",
    right_on="published_at",
    direction="backward",
)
return merged

```

merge_asof with an explicit decision cutoff replaces the vague notion of "same date" with a specific moment, the session open, and pulls only sentiment published before that moment. Any article published after the cutoff is correctly excluded from that day's feature and instead becomes available for the following session. This is the same mechanism used in section 2.1 to align fundamentals by release date rather than report date. The pattern generalizes: whenever you combine two time stamped sources, define a single decision point per row, and pull from each source only the most recent value available strictly before that point.

Time zones compound this problem in a way that is easy to overlook because the bug does not announce itself. A vendor delivering intraday data in UTC, joined against a price feed assumed to be in US Eastern time without an explicit conversion, will misalign every row by four or five hours depending on the time of year, due to daylight saving transitions. A four hour misalignment does not merely blur the boundary between predictive and reactive information, the way the sentiment example above does. It can push an entire trading session's data onto the wrong calendar date on either side of the error, corrupting the join at a much coarser level.

python

```

def normalize_timezone(df: pd.DataFrame, source_tz: str,
target_tz: str = "America/New_York") -> pd.DataFrame:

```

```
df = df.copy()
if df["timestamp"].dt.tz is None:
    df["timestamp"] = df["timestamp"].dt.tz_localize(source_tz)
df["timestamp"] = df["timestamp"].dt.tz_convert(target_tz)
return df
```

Convert every source to a single, explicit target time zone before any date based grouping or joining happens, and never rely on an unlabeled, timezone naive timestamp column, because a naive timestamp forces you to assume its zone, and that assumption is exactly the kind of silent decision that produces the errors described above.

Different sampling frequencies raise a related but distinct issue. A weekly economic indicator, a daily price series, and an intraday order book feature cannot be joined by matching frequencies without a deliberate resampling decision, and the direction of that resampling matters. Downsampling the daily price series to weekly, to match a weekly indicator, is generally safe if the weekly bucket is labeled by its start date and the resample only aggregates data from within that week. Upsampling a weekly indicator to daily, to match daily prices, requires forward filling the weekly value across the days that follow its release, using the same `merge_asof` backward direction shown above, and never using a fill method that could pull a value backward from a release date to days that preceded it.

Red flag. A reader submits a multi source strategy, typically combining price with sentiment, news, or macroeconomic data, where the backtest achieves its strongest performance specifically on days with the largest price moves.

Genuine predictive features tend to add modest, consistent value across ordinary trading days. A feature that only appears to work on the days with the biggest moves is more often explained by same day reactive information leaking in through an imprecise timestamp join than by a model that has learned to specifically anticipate large moves.

The governing rule across time zones, sampling frequency, and release timing is the same rule from section 2.1 applied at the join step rather than the source selection step. Know the exact moment each value became available, define a single decision cutoff for each row of your dataset, and join every source to that cutoff explicitly rather than to a shared but ambiguous calendar date. Section 2.4 now puts sections 2.1 through 2.3 together in a single worked example,

building one clean, point in time correct price dataset from raw, unaligned inputs.

Worked Example: Building a Clean Price Dataset

You now have three separate disciplines, point in time sourcing from section 2.1, gap and corporate action handling from section 2.2, and timestamp alignment from section 2.3. None of them is difficult in isolation. What actually breaks in practice is the order you apply them in, and this section exists because that order is not obvious until you have gotten it wrong once.

The task here is to build a single clean dataset for one security, combining raw daily price data, a corporate actions file, and a raw fundamentals feed, ending with a dataframe where every row is safe to hand directly to a feature construction function with no further correction needed.

Start with the raw inputs exactly as a typical vendor would deliver them, before any of the fixes from this chapter have been applied.

python

```
import pandas as pd

raw_prices = pd.read_parquet("data/raw/prices.parquet")
# columns: date, close, volume — unadjusted, gaps present,
weekends absent

splits = pd.read_parquet("data/raw/splits.parquet")
# columns: date, ratio

fundamentals_raw =
pd.read_parquet("data/raw/fundamentals.parquet")
# columns: report_date, release_date, eps, revenue
# release_date is the date the figure was actually made public,
# report_date is the fiscal period the figure describes
```

Step one is reindexing against an actual trading calendar, which is the prerequisite for correctly classifying gaps, established in section 2.2.

python

```
import pandas_market_calendars as mcal
```

```
calendar = mcal.get_calendar("NYSE")
```

```
schedule = calendar.schedule(
```

```
    start_date=raw_prices["date"].min(),
```

```
    end_date=raw_prices["date"].max()
```

```
)
```

```
trading_days = schedule.index
```

```
full_index = pd.DataFrame({"date": trading_days})
```

```
prices = full_index.merge(raw_prices, on="date", how="left")
```

Every row in prices now corresponds to a genuine trading session. Any close value still missing after this merge is a data gap, not a calendar gap, and section 2.2's distinction now applies cleanly because weekends and holidays were never in the frame to begin with.

Step two handles those data gaps explicitly rather than filling them by default.

python

```
prices = prices.sort_values("date").reset_index(drop=True)
```

```
prices["close"] = prices["close"].ffill(limit=2)
```

```
remaining_gaps = prices["close"].isna().sum()
```

```
if remaining_gaps > 0:
```

```
    # gaps longer than two sessions are left as NaN on purpose,
```

```
    # forcing an explicit decision later about whether this security
```

```
    # belongs in the universe on those dates
```

```
    print(f"{remaining_gaps} sessions left unfilled, longer than 2  
day limit")
```

Step three applies the split adjustment, working backward from the most recent split as shown in section 2.2, and critically, it keeps the unadjusted close alongside the adjusted one rather than overwriting it.

python

```
prices["close_unadjusted"] = prices["close"]
```

```
prices["close_adjusted"] = prices["close"]
```

```
cumulative_factor = 1.0
```

```
for date, ratio in splits.sort_values("date",  
ascending=False).itertuples(index=False):
```

```
    mask = prices["date"] <= date
```

```
    prices.loc[mask, "close_adjusted"] /= cumulative_factor
```

```
    cumulative_factor *= ratio
```

```
prices = prices.drop(columns=["close"])
```

Both columns now exist independently. Return based features will be built from `close_adjusted`. Any feature that depends on the actual nominal price level a trader would have seen on the screen that day uses `close_unadjusted`, and the two are never mixed within a single calculation.

Step four joins the fundamentals data using release date rather than report date, applying the `merge_asof` pattern from section 2.1, so that each price row only receives the most recent fundamental figure that had actually been made public as of that date.

python

```
fundamentals_raw =
```

```
fundamentals_raw.sort_values("release_date")
```

```
prices = prices.sort_values("date")
```

```
prices = pd.merge_asof(  
    prices,
```

```
    fundamentals_raw,
```

```
    left_on="date",
```

```
    right_on="release_date",
```

```
    direction="backward",
```

```
)
```

A price row from before the company's first ever fundamentals release will correctly receive NaN for eps and revenue, rather than being backfilled with a

figure that did not yet exist, which is the behavior a plain merge on date would silently produce instead.

Step five, the final step, adds an explicit availability check that runs against the finished dataset before it is written to the processed data directory, catching any row where a fundamental value's release date falls after the price date it was joined to, which would indicate a bug in the merge rather than a legitimate value.

python

```
violations = prices[prices["release_date"] > prices["date"]]
assert violations.empty, f"{len(violations)} rows contain future
fundamentals"
```

```
prices.to_parquet("data/processed/clean_prices_v1.parquet")
```

This assertion is not defensive programming for its own sake. It is a direct, automated test of the exact failure mode described in sections 1.2 and 2.1, and running it as a standing check on every processed dataset you build catches a report date versus release date mistake immediately, rather than three chapters later when a backtest result looks unexpectedly strong and you have to work backward to find out why.

Red flag. A reader submits a cleaned dataset with no accompanying assertion or test verifying that no future information leaked into any row during the cleaning process itself. The cleaning code can be correct and the resulting bug can still exist, introduced by a change in the raw vendor file months later, and a dataset with no automated check for its own core invariant will not catch that change until the backtest results built on top of it start behaving strangely.

What you have at the end of this process is a single parquet file where every row's `close_adjusted`, `close_unadjusted`, `eps`, and `revenue` values are genuinely knowable as of that row's date, where gaps are either filled within a deliberate two day limit or left as an explicit NaN rather than silently smoothed over, and where the join logic itself has been tested rather than assumed correct. This is the dataset every feature in Chapter 3 will be built from, and the discipline is not a one time setup cost. It is the standard every future data source you add to this pipeline needs to meet before it touches a single feature calculation.