fdf
@0xfdf

QR2: Peel, pith and seeds

I describe and motivate the risk methodology used by institutional quants. I also implement a complete (albeit basic) multi-factor risk model from scratch, in the same style as Barra and Axioma. I walk through its development end to end, and I explain why various feature engineering and modeling techniques are used. I also compare my results to good sources of truth; namely Barra's style factors.

The entire library, called Toraniko, can be found in this GitHub repo. It's implemented in modern Python, permissively licensed (MIT) and fully documented. The only third party dependencies are numpy and polars.

This is the second article in my series on quantitative research*. The series is inspired by a training manual I put together at a previous firm, which was used to turn numerate software engineers - especially backend and systems developers - into competent and commercial quant researchers. None of my current or any previous firm's intellectual property, code, equipment or data was used in any fashion for any part of this series.*

On pomegranates and AAPLs

The seeds of the pomegranate are used in a dazzling variety of culinary cultures. But the fruit itself, consisting of its tough outer skin (peel) and bitter interior flesh (pith), is bitter and inedible. It's relatively difficult and time-consuming to extract the seeds, which are only a tiny part of the fruit by total weight. You must slice the pomegranate, loosen the pith in water and then carefully remove the seeds without popping them. Yet they are so delicious, they have been consumed throughout antiquity. So it is with risk.

Now suppose you want to buy AAPL, because you have a fundamental thesis or systematic signal indicating the company is undervalued. All return implies some measure of risk; in this case, the risk you want to underwrite is that you're wrong about AAPL being undervalued.

Investing in AAPL actually underwrites several distinct *factor risks*:

1. A bet on AAPL's *idiosyncratic risk*, i.e. its performance in complete isolation from the rest of the market.
2. A bet on AAPL's *sector factor*, in this case technology.
3. A bet on AAPL's *market factor*, in this case the S&P 500.
4. Bets on a number of different *style factors*, such as *value* and *momentum*, which are related to the zeitgeist of investor behavior around AAPL.

In other words, equities have many *betas*, not just one market beta. In fact, for most companies the idiosyncratic part of its return, or *alpha*, you'd like to bet on is a minority of its total risk. They're almost completely driven by the ancillary factors you don't have a view on and (essentially) can't predict.

Why do those factors exist, and why do they drive equity returns?

Much like a pomegranate, the tantalizing alpha you seek is just small part of the whole. So let's learn how to separate the pith from its seeds.

The risk model

There are several kinds of fundamental risk models. The classical model used in the academic literature is the *timeseries model*, in which the assets' returns are regressed against some measure of interest. Instead I'll implement a *characteristic model*, which is the same kind implemented by major commercial risk model vendors such as Barra and Axioma. In this setting we estimate cross-sectional betas of each asset to a fundamental characteristic we believe is economically and statistically significant, and then in a second pass we use the betas to estimate the factor returns. The betas of the asset to the factor are also called its *loadings* on that factor.

In the case of a single factor, the timeseries model looks like a familiar regression:

$$r_i = \alpha_i + \beta_{f,i} \times f + \epsilon_i$$

The timeseries returns of an asset i are (in expectation) a linear function of the asset's alpha and its beta to the factor returns timeseries. The idiosyncratic risk is given by the error term, epsilon. For practical purposes we fold it into the alpha term, so that the alpha we're really after is

$$\alpha_i \approx r_i - \beta_i \times f$$

Thus we obtain a simple but deep truth: you cannot know alpha before you know beta. With poetic brevity at the expense of scientific clarity: alpha is *defined by beta*. The thing you seek is shrouded by that which you don't want (and may even hurt you). Alpha is unrequited care from an abusive lover.

Now instead assume we have sensible betas first but don't have a good guess for the factor returns. We'll reverse the regression to estimate factor returns from the betas, which are statistically and economically significant in the *cross-section* of returns. Thus for *all* asset returns at time t:

$$r_t = \alpha_t + B_t \times f_t + \epsilon_t$$

where r is an n-vector of all assets' returns at time t, alpha is an n-vector of the expected returns of each asset, B is an n x m matrix of factor betas, f is an m-vector comprising the returns of each factor at time t, and epsilon is an n-vector of the error, or idiosyncratic returns.

For our model we will consider 15 factors:

1. The market factor,
2. The sector factors (11 total)
3. The momentum, value and size style factors

This gives us enough material to build a structurally complete factor model that can be extended with arbitrarily many subsectors, industries, or additional style factors. The main caveat is the implemented model only supports a single country (i.e. there is no support for *country factors*). But with minor extension that would be resolved.

Why do factors exist?

A large amount of literature has empirically shown that these factors are real and do in fact explain the cross-section of asset returns (though arguably they've gotten a bit carried away in defining a *zoo* of factors in recent years...). I'll point you to the list of references for more detail here.

On the other hand there is no single compelling answer as to *why* they exist. Many believe they just emerge from persistent but irrational investor neuroses and overreactions. There are credible arguments the factors represent *risk premia,* i.e. compensation for holding unwanted risk to supply an economic demand in the market). And others say they are the result of crowded alpha that has become old hat, priced in by the market.

Personally I think there's an element of truth to all these. The market is simply human behavior writ large; it is a macrocosm of us and all our attendant biases. Maybe that's all we have to say about it.

> **fdf** @0xfdf · Jun 26
> Replying to @macrocephalopod and @nope_its_lily
> Don't understand the causes? Have you never watched people dance and felt the urge to join them? Have you never known love, then felt it slowly fade away until nothing remained?
>
> Then you understand momentum and mean reversion.
>
> 4 30 791

A definitional digression

Like a middle schooler emboldened upon their first encounter with a list of logical fallacies, one might be tempted by the foregoing model to argue against widespread claims of "alpha". On first glance it looks like it has a precise definition: simply identify all betas to other factors, residualize your asset returns against them, and you have alpha! And anything not that, is not alpha!

Not so fast. "Alpha" as colloquially used is underspecified. What most reasonable people can agree on is that it's both rare and fleeting. I will offer three common perspectives:

1. "Today's risk factor is yesterday's alpha factor". In this setting, alpha is a risk premium. It has a legitimate return which is outsized because it's esoteric and relatively unknown. Eventually that risk premium will become a part of the regular old market mechanics and cease to be as profitable, even if it persists as a factor. In this setting, mid-frequency CTAs that specialize in trend and carry may have alpha if they are able to consistently provide a positive market neutral return in excess of the momentum factor.
2. Alpha refers only to true market inefficiencies which must be kept secret in order to work, and which are largely capacity constrained. It decays rapidly throughout the crowding lifecycle of its strategies. These proponents would argue Jane Street's Indian options are emblematic of "true" alpha, and in fact you don't really find it outside the intraday horizon.
3. Alpha is superlative operational efficiency, wherein you gather together enough people who are just skilled enough and just diverse enough to get positive market neutral returns in (apparently) any regime. In this regard, Citadel's and Millennium's fundamental equities groups that operate on quarter+ time horizons have alpha.

In my view this debate is largely unimportant, in the sense of Wittgenstein. If you demonstrate legitimate skill at mixing together risk premia, keep at that. If you specialize in identifying inefficient parts of market microstructure you can harvest, do that. Find your flame and give it more kindling. Above all else, *understand and attribute* the risks you take, so you can be honest with yourself and your investors.

Suppose you and your friend are sitting on a beach and bet with one another about who could reach an island first. A group of gamblers comes around and decides to place wagers. You agree to terms, and at the last moment find a raft with a mediocre but serviceable sail. Your friend on the other hand swims. With a favorable tailwind at your back, you handily win.

To what do you attribute your success? Certainly not your swimming prowess. You won with your sail but you cannot control the winds, and one day that tailwind will become a headwind.

You can cleave the debate about what alpha "is", because this is what really matters. If you understand your risk you become the master of your fate: you know if winds favored your sail, or if you were actually a good swimmer. Now, back to our scheduled programming.

Implementing the model

We have our asset returns r and our asset betas B. We don't have our expected returns. In practice that's okay: if we construct approximately normal factors, standardizing them will theoretically fold the expected returns vector, alpha, into the idiosyncratic returns vector, epsilon.

We don't have the idiosyncratic variance of the returns; neither do we have the idiosyncratic covariance matrix. We'd like to use the inverse of the idio covariance to estimate the model via

$$\min(r_t - \mathrm{B}_t \times f_t)^T \Omega^{-1}(r_t - \mathrm{B}_t \times f_t)$$

Instead we'll use the (square root of) assets' market caps as a proxy. Then our factor returns are given by

$$f_t = (\mathrm{B}_t^T \times \mathrm{W} \times \mathrm{B}_t)^{-1}\mathrm{B}_t \times \mathrm{W} \times r_t$$

Which is estimable via weighted least squares. So, let's implement that function:

python

```python
def _factor_returns(
    returns: np.ndarray,
    mkt_caps: np.ndarray,
    sector_scores: np.ndarray,
    style_scores: np.ndarray,
    residualize_styles: bool,
) -> tuple[np.ndarray, np.ndarray]:
    """Estimate market, sector, style and residual asset returns for one time

    Parameters
    ----------
    returns: np.ndarray returns of the assets (shape n_assets x 1)
    mkt_caps: np.ndarray of asset market capitalizations (shape n_assets x 1)
    sector_scores: np.ndarray of asset scores used to estimate the sector retu
    style_scores: np.ndarray of asset scores used to estimate style factor ret
    residualize_styles: bool indicating if styles should be orthogonalized to

    Returns
    -------
    tuple of arrays: (market/sector/style factor returns, residual returns)
    """
    n_assets = returns.shape[0]
    m_sectors, m_styles = sector_scores.shape[1], style_scores.shape[1]

    # Proxy for the inverse of asset idiosyncratic variances
    W = np.diag(np.sqrt(mkt_caps.ravel()))

    # Estimate sector factor returns with a constraint that the sector factors
    # Economically, we assert that the market return is completely spanned by
    beta_sector = np.hstack([np.ones(n_assets).reshape(-1, 1), sector_scores])
    a = np.concatenate([np.array([0]), (-1 * np.ones(m_sectors - 1))])
    Imat = np.identity(m_sectors)
    R_sector = np.vstack([Imat, a])
    # Change of variables to add the constraint
    B_sector = beta_sector @ R_sector

    V_sector, _, _, _ = np.linalg.lstsq(
        B_sector.T @ W @ B_sector, B_sector.T @ W, rcond=None
    )
    # Change of variables to recover all sectors
    g = V_sector @ returns
    fac_ret_sector = R_sector @ g

    sector_resid_returns = returns - (B_sector @ g)

    # Estimate style factor returns without constraints
    V_style, _, _, _ = np.linalg.lstsq(
        style_scores.T @ W @ style_scores, style_scores.T @ W, rcond=None
    )
    if residualize_styles:
        fac_ret_style = V_style @ sector_resid_returns
    else:
        fac_ret_style = V_style @ returns

    # Combine factor returns
    fac_ret = np.concatenate([fac_ret_sector, fac_ret_style])

    # Calculate final residuals
    epsilon = sector_resid_returns - (style_scores @ fac_ret_style)

    return fac_ret, epsilon
```

Note the change in variables: economically, the market return is spanned by the sector returns. That's a sound assertion for our model, but it means that the model doesn't have unique solution. Algebraically it causes the matrix of betas to be less than full rank, underdetermining the solution space due to the column of ones.

To resolve this we need to constrain our regression to a unique solution. That's easily doable by constraining the sector betas to sum to 0. This leaves them with a convenient economic interpretation as deviations from the market factor. The style factors do not have such an issue, so we leave them be.

This function returns the factor returns and asset residual returns for a single period. Let's fill it out so we can do this on a rolling basis for asset returns across many dates. We'll also implement a *winsorization* function - easy enough to do, and saves us from having to import scipy.

python

```python
def winsorize(data: np.ndarray, percentile: float = 0.05, axis: int = 0) -> np
    """Windorize each vector of a 2D numpy array to symmetric percentiles give

    This returns a Polars expression, not a DataFrame, so it be chained (inclu
    a `select` or `with_columns` invocation without needing to set a new inter

    Parameters
    ----------
    data: numpy array containing original data to be winsorized
    percentile: float indicating the percentiles to apply winsorization at
    axis: int indicating which axis to apply winsorization over (i.e. orientat

    Returns
    -------
    numpy array
    """
    if not 0 <= percentile <= 1:
        raise ValueError("`percentile` must be between 0 and 1")

    fin_data = np.where(np.isfinite(data), data, np.nan)

    # compute lower and upper percentiles for each column
    lower_bounds = np.nanpercentile(
        fin_data, percentile * 100, axis=axis, keepdims=True
    )
    upper_bounds = np.nanpercentile(
        fin_data, (1 - percentile) * 100, axis=axis, keepdims=True
    )

    # clip data to within the bounds
    return np.clip(data, lower_bounds, upper_bounds)

def estimate_factor_returns(
    returns_df: pl.DataFrame,
    mkt_cap_df: pl.DataFrame,
    sector_df: pl.DataFrame,
    style_df: pl.DataFrame,
    winsor_factor: float | None = 0.05,
    residualize_styles: bool = True,
) -> tuple[pl.DataFrame, pl.DataFrame] | pl.DataFrame:
    """Estimate factor and residual returns across all time periods using inpu

    Parameters
    ----------
    returns_df: Polars DataFrame containing | date | symbol | asset_returns |
    mkt_cap_df: Polars DataFrame containing | date | symbol | market_cap |
    sector_df: Polars DataFrame containing | date | symbol | followed by one c
    style_df: Polars DataFrame containing | date | symbol | followed by one co
    winsor_factor: winsorization proportion
    residualize_styles: bool indicating if style returns should be orthogonali

    Returns
    -------
    tuple of Polars DataFrames melted by date: (factor returns, residual retur
    """
    returns, residuals = [], []
    try:
        sectors = sorted(sector_df.select(pl.exclude("date", "symbol")).column
    except AttributeError as e:
        raise TypeError(
            "`sector_df` must be a Polars DataFrame, but it's missing required
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            "`sector_df` must have columns for 'date' and 'symbol' in addition
        ) from e
    try:
        styles = sorted(style_df.select(pl.exclude("date", "symbol")).columns)
    except AttributeError as e:
        raise TypeError(
            "`style_df` must be a Polars DataFrame, but it's missing required
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            "`style_df` must have columns for 'date' and 'symbol' in addition
        ) from e
    try:
        returns_df = (
            returns_df.join(mkt_cap_df, on=["date", "symbol"])
            .join(sector_df, on=["date", "symbol"])
            .join(style_df, on=["date", "symbol"])
        )
        dates = returns_df["date"].unique().to_list()
        # iterate through, one day at a time
        # this could probably be made more efficient with Polars' `.map_groups
        for dt in dates:
            ddf = returns_df.filter(pl.col("date") == dt).sort("symbol")
            r = ddf["asset_returns"].to_numpy()
            if winsor_factor is not None:
                r = winsorize(r, winsor_factor)
            f, e = _factor_returns(
                r,
                ddf["market_cap"].to_numpy(),
                ddf.select(sectors).to_numpy(),
                ddf.select(styles).to_numpy(),
                residualize_styles,
            )
            returns.append(f)
            residuals.append(dict(zip(ddf["symbol"].to_list(), e)))
    except AttributeError as e:
        raise TypeError(
            "`returns_df` and `mkt_cap_df` must be Polars DataFrames, but ther
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            "`returns_df` must have columns 'date', 'symbol' and 'asset_return
            "`mkt_cap_df` must have 'date', 'symbol' and 'market_cap' columns"
        ) from e
    ret_df = pl.DataFrame(np.array(returns))
    ret_df.columns = ["market"] + sectors + styles
    ret_df = ret_df.with_columns(pl.Series(dates).alias("date"))
    eps_df = pl.DataFrame(residuals).with_columns(pl.Series(dates).alias("date
    return ret_df, eps_df
```

Why winsorize returns? Consider the pitfalls of least squares loss function: squared errors are disproportionately penalized more harshly the further they are from 0s. So outlier returns can significantly skew our model. You might be tempted to scheme up some moon math nonlinear model to deal with this, but actually winsorization works very well in practice.

Parsimony is a moral virtue, do not abandon it at the first sign of trouble.

Where do we get our betas?

Good question. It may not be clear from the model, but we actually don't need to supply a market factor. That will be estimated for us from the constrained, weighted regression on the sector betas. The sector betas themselves are straightforward: use dummy variables, so that each return has a sparse vector in which the only sector it corresponds to is represented by 1, and every other sector is 0. Then only the returns in that sector will count to estimate the sector factor's return. Like so:

Pictured above are the sectors as designated by the Global Industry Classification Standard (GICS). That's a pretty good source. You can pull the GICS sector classifications for any asset from Bloomberg, FactSet or (if you're on a budget) freely from Yahoo Finance.

The style factors are much trickier. Factor construction is as much an art as a science. There are sound statistical methods, but you can't apply them in a recipe-like fashion. As with geometry, there is no royal road. You have to deeply understand your data and work at it. So let's start with the momentum factor since it's pretty easy. Here is how Barra defines it:

That's pretty easy to implement. We'll need a few additional utility functions first so we can exponentially weight the returns and standardize the result (recall what I said earlier about the intercept when we standardize our factors).

python

```python
def exp_weights(window: int, half_life: int) -> np.ndarray:
    """Generate exponentially decaying weights over `window` trailing values,

    Parameters
    ----------
    window: integer number of points in the trailing lookback period
    half_life: integer decay rate

    Returns
    -------
    numpy array
    """
    decay = np.log(2) / half_life
    return np.exp(-decay * np.arange(window))[::-1]

def center_xsection(
    target_col: str, over_col: str, standardize: bool = False
) -> pl.Expr:
    """Cross-sectionally center (and optionally standardize) a Polars DataFram

    This returns a Polars expression, so it be chained in a `select` or `with_
    without needing to set a new intermediate DataFrame or materialize lazy ev

    Parameters
    ----------
    target_col: the column to be standardized
    over_col: the column over which standardization should be applied, cross-s
    standardize: boolean indicating if we should also standardize the target c

    Returns
    -------
    Polars Expr
    """
    expr = pl.col(target_col) - pl.col(target_col).drop_nulls().drop_nans().me
        over_col
    )
    if standardize:
        return expr / pl.col(target_col).drop_nulls().drop_nans().std().over(o
    return expr

def winsorize_xsection(
    df: pl.DataFrame | pl.LazyFrame,
    data_cols: tuple[str, ...],
    group_col: str,
    percentile: float = 0.05,
) -> pl.DataFrame | pl.LazyFrame:
    """Cross-sectionally winsorize the `data_cols` of `df`, grouped on `group_
    given by `percentile`.

    Parameters
    ----------
    df: Polars DataFrame or LazyFrame containing feature data to winsorize
    data_cols: collection of strings indicating the columns of `df` to be wins
    group_col: str column of `df` to use as the cross-sectional group
    percentile: float value indicating the symmetric winsorization threshold

    Returns
    -------
```

```
        Polars DataFrame or LazyFrame
        """

    def winsorize_group(group: pl.DataFrame) -> pl.DataFrame:
        for col in data_cols:
            winsorized_data = winsorize(group[col].to_numpy(), percentile=perc
            group = group.with_columns(pl.Series(col, winsorized_data).alias(c
        return group

    grouped = df.groupby(group_col).apply(winsorize_group)
    return grouped

def factor_mom(
    returns_df: pl.DataFrame | pl.LazyFrame,
    trailing_days: int = 504,
    half_life: int = 126,
    lag: int = 20,
    winsor_factor: float = 0.01,
) -> pl.LazyFrame:
    """Estimate rolling symbol by symbol momentum factor scores using asset re

    Parameters
    ----------
    returns_df: Polars DataFrame containing columns: | date | symbol | asset_r
    trailing_days: int look back period over which to measure momentum
    half_life: int decay rate for exponential weighting, in days
    lag: int number of days to lag the current day's return observation (20 tr

    Returns
    -------
    Polars DataFrame containing columns: | date | symbol | mom_score |
    """
    weights = exp_weights(trailing_days, half_life)

    def weighted_cumprod(values: np.ndarray) -> float:
        return (np.cumprod(1 + (values * weights[-len(values) :])) - 1)[-1]  #

    try:
        df = (
            returns_df.lazy()
            .sort(by="date")
            .with_columns(
                pl.col("asset_returns").shift(lag).over("symbol").alias("asset
            )
            .with_columns(
                pl.col("asset_returns")
                .rolling_map(weighted_cumprod, window_size=trailing_days)
                .over(pl.col("symbol"))
                .alias("mom_score")
            )
        ).collect()
        df = winsorize_xsection(df, ("mom_score",), "date", percentile=winsor_
        return df.lazy().select(
            pl.col("date"),
            pl.col("symbol"),
            center_xsection("mom_score", "date", True).alias("mom_score"),
        )
    except AttributeError as e:
        raise TypeError(
            "`returns_df` must be a Polars DataFrame | LazyFrame, but it's mis
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            "`returns_df` must have 'date', 'symbol' and 'asset_returns' colum
        ) from e
```

For simplicity I'm using simple returns, not log returns. That's not strictly speaking correct, but it's close enough. I also assume you remove the risk-free rate yourself: that operation is not implemented in each of the functions. As a general rule, these functions want the data to have been preprocessed per the single-responsibility principle.

Now here are the returns I start with:

Passing these through the function with a 99th percentile winsorization and 504 trailing days, we obtain:

All pricing data is spotty, even from excellent vendors like Bloomberg. So let's clean our results a bit with a function to make sure everything is numeric, and forward will gaps. We should be cautious when we do timeseries operations on returns like this, but in this case we're dragging forward old observations whenever present observations are null. That should smoothen out our distribution.

python

```python
def fill_features(
    df: pl.DataFrame | pl.LazyFrame, features: tuple[str], sort_col: str, over
) -> pl.LazyFrame:
    """Cast feature columns to numeric (float), convert NaN and inf values to
    for each column of `features`, sorted on `sort_col` and partitioned by `ov

    Parameters
    ----------
    df: Polars DataFrame or LazyFrame containing columns `sort_col`, `over_col
    features: collection of strings indicating which columns of `df` are the f
    sort_col: str column of `df` indicating how to sort
    over_col: str column of `df` indicating how to partition

    Returns
    -------
    Polars LazyFrame containing the original columns with cleaned feature data
    """
    try:
        return (
            df.lazy()
            .with_columns([pl.col(f).cast(float).alias(f) for f in features])
            .with_columns(
                [
                    pl.when(
                        (pl.col(f).abs() == np.inf)
                        | (pl.col(f) == np.nan)
                        | (pl.col(f).is_null())
                        | (pl.col(f).cast(str) == "NaN")
                    )
                    .then(None)
                    .otherwise(pl.col(f))
                    .alias(f)
                    for f in features
                ]
            )
            .sort(by=sort_col)
            .with_columns(
                [pl.col(f).forward_fill().over(over_col).alias(f) for f in fea
            )
        )
    except AttributeError as e:
        raise TypeError(
            "`df` must be a Polars DataFrame | LazyFrame, but it's missing req
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            f"`df` must have all of {[over_col, sort_col] + list(features)} as
        ) from e
```

Then voila, look at this beautiful distribution:

A bit Laplacian, but honestly this is close to as good as you can expect these things to get on real world data. In the real world our loved ones have ugly tails, but we still accept them.

Now let's try the value factor. This is more complicated. Per Barra:

This is hairy data to get if you're retail. You need to get quarterly and annual fundamental data for a very large universe; namely book value, revenue, cash flow and outstanding share counts. Then each of those need to be normalized by price to obtain "to price" ratios. In theory, each of these is a common fundamental heuristic for how "cheap" a company is relative to its outstanding assets, incoming/outgoing cash and earnings power.

Before we implement the value factor, let's get comfortable with our data. Take a look at AAPL, a traditionally "low value" name:

And then compare to Phizer, a "high value" name:

So far the posterior matches our priors about value companies. That's good. But let's play Devil's Advocate and really litigate the data. Can we find issues?

So there are absurd book-prices that break any semblance of a reasonable distribution. Once you step outside the top 1000 equities by market cap, these issues will become more and more apparent. But you want a universe of several thousand if you're going to build your own factor model, you have to work around this somehow.

If we look closely, we see similarly odd magnitude for sales-price and cash-flow-price. Sometimes the data is null or NaN in one column and not others, sometimes one or more features is completely outlandish, and in any case the distributions are completely different. So we need to do some serious data cleaning to wrangle this into a useful factor. Here is what we'll do:

python

```python
def factor_val(
    value_df: pl.DataFrame | pl.LazyFrame, winsorize_features: float | None =
) -> pl.LazyFrame:
    """Estimate rolling symbol by symbol value factor scores using price ratio

    Parameters
    ----------
    value_df: Polars DataFrame containing columns: | date | symbol | book_pric
    winsorize_features: optional float indicating if the features should be wi

    Returns
    -------
    Polars DataFrame containing: | date | symbol | val_score |
    """
    try:
        if winsorize_features is not None:
            value_df = winsorize_xsection(
                value_df, ("book_price", "sales_price", "cf_price"), "date"
            )
        return (
            value_df.lazy()
            .with_columns(
                pl.col("book_price").log().alias("book_price"),
                pl.col("sales_price").log().alias("sales_price"),
            )
            .with_columns(
                center_xsection("book_price", "date", True).alias("book_price"
                center_xsection("sales_price", "date", True).alias("sales_pric
                center_xsection("cf_price", "date", True).alias("cf_price"),
            )
            .with_columns(
                # NB: it's imperative you've properly handled NaNs prior to th
                pl.mean_horizontal(
                    pl.col("book_price"),
                    pl.col("sales_price"),
                    pl.col("cf_price"),
                ).alias("val_score")
            )
            .select(
                pl.col("date"),
                pl.col("symbol"),
                center_xsection("val_score", "date", True).alias("val_score"),
            )
        )
    except AttributeError as e:
        raise TypeError(
            "`value_df` must be a Polars DataFrame or LazyFrame, but it's miss
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            "`value_df` must have 'date', 'symbol', 'book_price', 'sales_price
        ) from e
```

You'll notice several things are happening here:

1. I winsorize each of the three value features cross-sectionally (so we have no look-ahead bias). The reason for winsorization is because some of those values are truly wacky, either because of ridiculous outliers or because of spotty data. You will never have perfect data (but you should strive for it!), so we have to make allowances. Recall what was said about the least squares loss and outlier fitting.
2. I demean each feature, so it's actually in a relative space, neutral to its entire peer set on that day. This projects the features into a space where they can be compared like for like. The winsorization reduced the variance, so this step brings things into a similar window of magnitude for comparison.
3. I take the simple average of the features, cross-sectionally by day, as a poor man's method of projecting a three-dimensional space into a single dimension. There are many more sophisticated ways of doing this: always start with this one. It will work far more often than you give it credit for, and it has no look-ahead bias.

After calling `fill_features` on our features, each one goes from looking something like this:

To this:

And with each feature centered, standardized and averaged together, we obtain another beautiful distribution:

It's a little lumpy to be sure, but it's serviceable as we'll see.

All together now

Now we're going to implement two final utility functions:

python

```python
def smooth_features(
    df: pl.DataFrame | pl.LazyFrame,
    features: tuple[str],
    sort_col: str,
    over_col: str,
    window_size: int,
) -> pl.LazyFrame:
    """Smooth the `features` columns of `df` by taking the rolling mean of eac
    partitioned by `over_col`, using `window_size` trailing periods for the mo

    Parameters
    ----------
    df: Polars DataFrame | LazyFrame containing columns `sort_col`, `over_col`
    features: collection of strings indicating which columns of `df` are the f
    sort_col: str column of `df` indicating how to sort
    over_col: str column of `df` indicating how to partition
    window_size: int number of time periods for the moving average

    Returns
    -------
    Polars LazyFrame containing the original columns, with each of `features`
    """
    try:
        return (
            df.lazy()
            .sort(by=sort_col)
            .with_columns(
                [
                    pl.col(f)
                    .rolling_mean(window_size=window_size)
                    .over(over_col)
                    .alias(f)
                    for f in features
                ]
            )
        )
    except AttributeError as e:
        raise TypeError(
            "`df` must be a Polars DataFrame | LazyFrame, but it's missing req
        ) from e
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            f"`df` must have all of {[over_col, sort_col] + list(features)} as
        ) from e

def top_n_by_group(
    df: pl.DataFrame | pl.LazyFrame,
    n: int,
    rank_var: str,
    group_var: tuple[str, ...],
    filter: bool = True,
) -> pl.LazyFrame:
    """Mark the top `n` rows in each of `group_var` according to `rank_var` de

    If `filter` is True, the returned DataFrame contains only the filtered dat
    the returned DataFrame has all data, with an additional 'rank_mask' column
    is in the filter.

    Parameters
    ----------
    df: Polars DataFrame | LazyFrame
    n: integer indicating the top rows to take in each group
    rank_var: str column name to rank on
    group_var: tuple of str column names to group and sort on
    filter: boolean indicating how much data to return

    Returns
    -------
    Polars LazyFrame containing original columns and optional filter column
    """
    try:
        rdf = (
            df.lazy()
            .sort(by=list(group_var) + [rank_var])
            .with_columns(
                pl.col(rank_var)
                .rank(descending=True)
                .over(group_var)
                .cast(int)
                .alias("rank")
            )
        )
        match filter:
            case True:
                return (
                    rdf.filter(pl.col("rank") <= n)
                    .drop("rank")
                    .sort(by=list(group_var) + [rank_var])
                )
            case False:
                return (
                    rdf.with_columns(
                        pl.when(pl.col("rank") <= n)
                        .then(pl.lit(1))
                        .otherwise(pl.lit(0))
                        .alias("rank_mask")
                    )
                    .drop("rank")
                    .sort(by=list(group_var) + [rank_var])
                )
    except pl.ColumnNotFoundError as e:
        raise ValueError(
            f"`df` is missing one or more required columns: '{rank_var}' and '
        ) from e
    except AttributeError as e:
        raise TypeError(
            "`df` must be a Polars DataFrame or LazyFrame but is missing a req
        ) from e
```

We use the first one to obtain rolling averages for our market caps, so we can smooth them. We use the second one to restrict our universe to only the top 3000 equities by market cap each day. This is a substantially representative universe on which to estimate a production-capable factor model. Then we have the following data:

Then we estimate our model. In 70 seconds, we have 10 years of daily data for 15 risk factors.

Let's take a look at the market factor first. The correlation of daily returns timeseries between our estimated market factor on ~3,000 equities and the S&P 500 is about 97%. That's very good and is clearly shown by a graph of returns below; so that worked!

However, it also shows that small errors in tracking error can compound to meaningful divergence in performance on large time horizons. Let's take a look at our value factor next. Here we'll use GSP1VALU as our source of truth, which is Barra's high minus low value factor as delivered by Goldman Sachs. This is what it looks like over the same period as our factor estimation:

And our freshly estimated value factor, for comparison:

That's also very good - the visual paths and cumulative returns are both extremely similar. It also matches an intuitive understanding about what the value factor did over the past decade. Let's also take a look at momentum. Goldman Sachs provides a similar factor index from Barra, GSP1MOMO:

Likewise, the visual paths between our fresh factor and a good source of truth are very similar, and end up very close together on cumulative returns over a ten year period.

Voila. And now you can slot arbitrary factors into this model and come

- Via research, ascertain a strong understanding of some credible dataset

Home
Explore
Notifications
Messages
Grok
Lists
Bookmarks
Communities
Premium
Verified Orgs
Profile
More

Post

Risk-taking (esp. in the sense of entrepreneurialism) is an example of a non-ergodic "good" for society. The rewards of success are so high that it's often (very) positive EV. But the modal outcome is actually very poor (even ruinous). And individuals don't get infinite shots.

Steve Hou @stevehouf · Jul 1

The answer to "if you are so smart, how come you are not rich?" is "risk-taking". Just because someone is smart or thinks clearly and correctly about many things, it doesn't mean they are good at risk-taking. To be rich, you need to take risk. Inordinate if not ex post seemingly x.com/stevehouf/stat...
Show more

4 12 109 24K

Entertainment ·
Disney
204K posts

Show more

Terms of Service
Accessibility A

•
•
•
•
•