Machine Learning for Return Prediction

BUSI 722: Data-Driven Finance II

Kerry Back

Standardizing Features

Why Standardize?

- Raw features have very different scales: marketcap is in thousands, roe is a fraction, momentum can be $\pm 100\%$.
- Many models (linear regression, neural networks) are sensitive to scale.
- Standardizing puts all features on a common scale and tames outliers.

Cross-Sectional Standardization

Standardize each feature **within each month**, not over the full panel.

For feature x in month t:

$$z_{i,t} = \frac{x_{i,t} - \bar{x}_t}{\sigma_{x,t}}$$

- Each month's cross section has mean 0 and standard deviation 1.
- Removes time-series level shifts (e.g., all P/E ratios rising over time).
- Ensures the model learns **which stocks are cheap relative to their peers this month**, not which months had low valuations overall.

Ranks as an Alternative

Instead of z-scores, rank stocks 1 through n each month, then scale to $[0, 1]$:

$$\text{rank}_{i,t} = \frac{\text{rank of } x_{i,t} \text{ among all stocks in month } t}{n_t}$$

- Completely eliminates outliers — every stock gets a value in $[0, 1]$.
- Preserves only the **ordering** of stocks, not the magnitudes.
- Robust to skewed distributions; can also compute z-scores of ranks for a hybrid approach.

Which Features Are “Good”?

When using ranks, we want a **high rank to always mean good**.

- **Higher is better:** momentum, roe, grossmargin $\Rightarrow$ rank low to high.
- **Lower is better:** asset growth (conservatism effect) $\Rightarrow$ rank high to low.

Alternatively, just rank all features the same way and let the model learn the sign. This is simpler and works well with flexible models (trees, neural nets).

Standardizing the Target

Predicting Relative Returns

We care more about **which stocks will outperform** than about predicting exact return magnitudes.

If every stock's return rises by 2% because the market rallies, that tells us nothing about which stocks to buy. What matters is the cross-sectional ranking of returns.

This suggests we should think carefully about what target variable we give the model.

Ranks as the Target

Replace the raw return $r_{i,t}$ with its **cross-sectional rank** each month:

$$y_{i,t} = \frac{\text{rank of } r_{i,t} \text{ among all stocks in month } t}{n_t}$$

- The model learns to predict **relative performance**, not absolute returns.
- Removes the influence of market-wide shocks and reduces the impact of return outliers.
- Scaled to $[0, 1]$: top stock ≈ 1, bottom stock ≈ 0.

Comparison of Targets

	Raw Returns	**Ranked Returns**
Scale	varies by month	always $[0, 1]$
Outliers	can be extreme	bounded
Market effect	included	removed
Interpretation	absolute return	relative ranking
Loss function	MSE on returns	MSE on ranks

Using ranked returns is common in quantitative equity research. It focuses the model on what actually drives portfolio construction: **selecting the right stocks relative to their peers**.

Models: A Review

The Prediction Problem

We have a panel of stock-months. For each stock i in month t:

- Features: $\mathbf{x}_{i,t-1}$ (known at the start of month t)
- Target: $y_{i,t}$ (return or return rank during month t)

We want to learn a function f such that

$$y_{i,t} \approx f(\mathbf{x}_{i,t-1})$$

- Train on past data, predict on future data (no look-ahead bias).
- The function f can be linear, a tree ensemble, a neural network, etc.

Linear Models

$$f(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$$

- Simple, interpretable, fast to estimate.
- Each coefficient β_j measures the marginal effect of feature j.
- Assumes the effect of each feature is **additive and linear** — no interactions, no diminishing effects.

Regularization

Add a penalty to prevent overfitting: $\min \sum (y_i - f(\mathbf{x}_i))^2 + \text{penalty}$

- **Ridge** (L_2): penalty $= \lambda \sum \beta_j^2$. Shrinks coefficients toward zero, keeps all features.
- **Lasso** (L_1): penalty $= \lambda \sum |\beta_j|$. Drives some coefficients to exactly zero (feature selection).
- **Elastic net**: combines L_1 and L_2 penalties. Balances shrinkage and sparsity.

The hyperparameter λ controls the strength of regularization. Larger λ = simpler model.

Limitations of Linearity

- A linear model says: “if momentum goes up by 1, expected return goes up by β.”
- But the real relationship may be **nonlinear**:
  - Momentum may help only above a threshold.
  - Value may matter more for small stocks than large stocks (interaction).
  - The effect of leverage may reverse at high levels.
- We need models that can capture these patterns automatically.

Decision Trees

A decision tree partitions the feature space into rectangular regions using a sequence of binary splits.

- At each node, pick the feature and threshold that best reduces prediction error.
- Naturally captures **nonlinearities** and **interactions**.
- A single tree is interpretable but tends to overfit.

Solution: combine many trees into an **ensemble**.

Random Forest

- Grow B trees, each on a **bootstrap sample** of the training data.
- At each split, consider only a **random subset** of features.
- Prediction $=$ average of all B trees.

Key properties:

- Reduces variance through averaging (bagging).
- Random feature selection decorrelates trees.
- Robust to outliers, requires little tuning, and hard to overfit by adding more trees.

Gradient Boosting (GBM)

Instead of averaging independent trees, build trees **sequentially**, each correcting the errors of the previous ones.

1. Start with a constant prediction (e.g., the mean).
2. Fit a shallow tree to the **residuals** (prediction errors).
3. Add the new tree's prediction (scaled by learning rate η) to the running total; repeat for B rounds.

Key hyperparameters: number of trees B, learning rate η, tree depth, minimum samples per leaf.

LightGBM

- Microsoft's implementation of gradient boosting, optimized for speed and scale.
- Uses **histogram-based** splitting and **leaf-wise** growth for faster, more accurate tree building.
- Widely used in quantitative finance and Kaggle competitions.

In practice, LightGBM often outperforms random forests on tabular financial data.

Random Fourier Features

A way to add nonlinearity to a linear model without the complexity of trees or neural networks.

Idea: transform the original k features into a much larger set of D random nonlinear features, then fit a linear model on the transformed features.

$$\phi(\mathbf{x}) = \sqrt{\frac{2}{D}} \begin{pmatrix} \cos(\mathbf{w}_1'\mathbf{x} + b_1) \\ \vdots \\ \cos(\mathbf{w}_D'\mathbf{x} + b_D) \end{pmatrix}$$

where $\mathbf{w}_j \sim N(0, \gamma^2 I)$ and $b_j \sim \text{Uniform}(0, 2\pi)$ are drawn **randomly and fixed**.

Why Random Fourier Features Work

- The cosine transformation creates **nonlinear combinations** of the original features, capturing interactions and curved relationships.
- The random weights $\mathbf{w}_j$ are not learned — only the final linear coefficients are estimated.
- Approximates a **kernel** method (RBF kernel) that scales to large datasets; bandwidth γ controls the "wiggliness" of the fit.

Advantages: fast to fit (just linear regression on transformed features), no neural network training, easy to regularize with ridge or lasso.

Neural Networks

A neural network is a sequence of linear transformations interleaved with nonlinear **activation functions**.

For a network with one hidden layer of h units:

$$f(\mathbf{x}) = \mathbf{w}_2' \, \sigma\big(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1\big) + b_2$$

- $\mathbf{W}_1$ is an $h \times k$ weight matrix, $\sigma(\cdot)$ is a nonlinear activation (ReLU, tanh, etc.).
- The hidden layer learns **nonlinear features** from the raw inputs.
- Adding more layers = deeper network = more complex representations.

Training Neural Networks

- Minimize a loss function (e.g., MSE) using **stochastic gradient descent** (SGD) or variants like Adam.
- Process data in **mini-batches**; one pass through the full training set = one **epoch**.
- Training continues for many epochs; monitor validation loss to detect overfitting.

Regularization:

- **Dropout:** randomly zero out a fraction of hidden units during training.
- **Early stopping:** stop training when validation loss stops improving.
- **Weight decay** (L_2 penalty on weights).

Neural Networks vs. Other Models

	Linear	**Trees**	**RFF**	**Neural Net**
Nonlinearity	no	yes	yes	yes
Interactions	no	yes	yes	yes
Interpretable	yes	moderate	no	no
Tuning effort	low	moderate	low	high
Training speed	fast	fast	fast	slow

No single model dominates. In practice, **ensembles** that combine predictions from multiple model types often perform best.