Validating & tuning

supervised
learning
factor
investing

WARNING

(disclaimer)

- This deck is a pot pourri of several concepts that are useful in Machine Learning:
  - the variance-bias trade-off
  - the tuning of hyperparameters
  - the risk of overfitting
  - the way all of this should be handled in backtesting
- The structure of the presentation is rather arbitrary (the links between these topics are not really ordered).

Agenda

1. Performance metrics in ML
2. Bias-variance tradeoff
3. Tools
4. Practical considerations
5. Wrap-up

Regression & co.

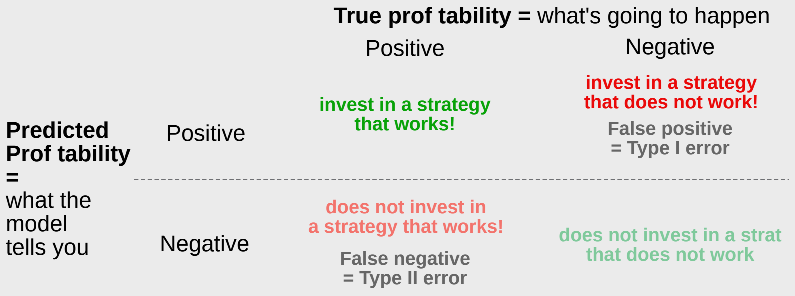
Not many choices...

There is only one real number to predict/explain.

- **mean squared error** (MSE): MSE $= I^{-1}\sum_{i=1}^{I}(y_i-\tilde{y}_i)^2$, sometimes translated into Root-MSE (RMSE) for scale purposes: RMSE $=\sqrt{I^{-1}\sum_{i=1}^{I}(y_i-\tilde{y}_i)^2}$
- **mean absolute error**: MAE $= I^{-1}\sum_{i=1}^{I}|y_i-\tilde{y}_i|$, easier to interpret, but analytically less tractable

When y_i and $\tilde{y}_i$ are positive, prior scaling is possible, as well as logarithmic transform.
It is also possible to introduce weights w_i to account for heterogeneity in instances (some may matter more than others, e.g., the left tail of returns).

Binary classification (1/2)



		True profitability = what's going to happen	
		Positive	Negative
Predicted Profitability = what the model tells you	Positive	**invest in a strategy that works!**	**invest in a strategy that does not work!** **False positive = Type I error**
	Negative	**does not invest in a strategy that works!** **False negative = Type II error**	**does not invest in a strat that does not work**

As is often the case, there is a strong **asymmetry** between the two error types because the two rows have different impacts (assuming no short selling). If we simplify:

- the first row decides what's in the portfolio: Type I is **bad** because it hurts actual performance
- the second decides what's **not** in the portfolio: Type II is **sad** because it's missed opportunities

→ the matrix is bigger (!) when dealing with **many classes**.

Binary classification (2/2)

		True	
		Positive	Negative
Predicted	Positive	TP=true positive	FP=false positive
	Negative	FN=false negative	TN=true negative

Other metrics include:

- **accuracy**: $\frac{TP+TN}{TP+TN+FP+FN}$
- **recall**: $\frac{TP}{TP+FN}$: probability to detect a winning strategy/asset
- **precision**: $\frac{TP}{TP+FP}$: 'probability to make a good investment'
- $\mathbf{F}_1 = 2\frac{\text{recall}\times\text{precision}}{\text{recall}+\text{precision}}$ (harmonic average of the two)

All of these metrics lie between zero (bad score) and one (perfect score).

see also:

- https://en.wikipedia.org/wiki/Confusion_matrix
- https://en.wikipedia.org/wiki/Receiver_operating_characteristic

Multi-class loss

How can we assess **goodness-of-fit** for **categorical** data? A classical metric is the **cross-entropy** (or log-loss). In the formula below, both the true value of the label $y_{i,c}$ and the model output $\tilde{y}_{i,c}$ are C dimensional. $y_{i,c}$ takes values $\{0,1\}$ (FALSE vs TRUE) and $\tilde{y}_{i,c} \in [0,1]$ with $\sum_{c=1}^C \tilde{y}_{i,c} = 1$.

$$\text{CE}_i = -\sum_{c=1}^C y_{i,c} \log(\tilde{y}_{i,c}),$$

where $c = 1, \dots, C$ are the class indices. The two components:

- $y_{i,c}$: does (true) the label belong to class c?
- $\log(\tilde{y}_{i,c})$: log of probability of predicted output belonging to c

So, if $y_{i,c} = 1$, then the loss is minus the log prob of belonging to c. If the probability is high, the loss is small.

(see also: multiclass hinge loss)

But ultimately

It’s all about financial performance

Sure, the metrics above will give indications on the quality of the trading signal.
But, financial performance is the overruling criterion.

- The **translation** of the signal into the portfolio composition is important.
- Most of the time, the ‘global’ (financial) performance metric is different from the loss function.
- For instance, the loss function can be an L^2 norm on returns, but one metric we care more about is the **hit ratio**[1] or the average return of the portfolio.

*The two levels of performance analysis (**ML engine** and **portfolio returns**) are obviously connected. Nonetheless, the crucial step of ‘signal translation’ can ruin a good signal or bolster a mediocre one.*

[1] The hit ratio can be viewed as the accuracy at predicting the sign of a return.

Agenda

1. Performance metrics in ML
2. Bias-variance tradeoff
3. Tools
4. Practical considerations
5. Wrap-up

In math language

Using simple notations, the true generation of data is

$$y = f(x) + \epsilon, \qquad \mathbb{E}[\epsilon] = 0, \quad \mathbb{V}[\epsilon] = \sigma^2$$

but our estimation can only be

$$y = \hat{f}(x) + \hat{\epsilon}, \quad \text{with again } \mathbb{E}[\hat{\epsilon}] = 0.$$

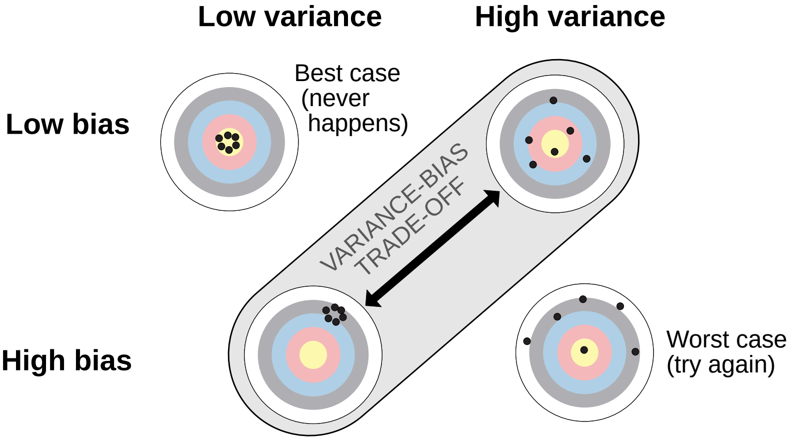
So the **error we make** is $\hat{\epsilon}$ and for some unknown sample x,

$$\begin{aligned}
\mathbb{E}[\hat{\epsilon}^2] &= \mathbb{E}[(y - \hat{f}(x))^2] = \mathbb{E}[(f(x) + \epsilon - \hat{f}(x))^2] \\
&= \underbrace{\mathbb{E}[(f(x) - \hat{f}(x))^2]}_{\text{total quadratic error}} + \underbrace{\mathbb{E}[\epsilon^2]}_{\text{irreducible error}} \\
&= \underbrace{\mathbb{V}[\hat{f}(x)]}_{\text{variance of model}} + \underbrace{\mathbb{E}[(f(x) - \hat{f}(x))]^2}_{\text{squared bias}} + \sigma^2
\end{aligned}$$

Note: in the above derivation, $f(x)$ is not random, but $\hat{f}(x)$ is.
In the second line, we assumed $\mathbb{E}[\epsilon(f(x) - \hat{f}(x))] = 0$, which sometimes may not hold if x were a r.v.

©Guillaume Coqueret

Graphically (1/3)



Low variance

High variance

Low bias

Best case
(never
happens)

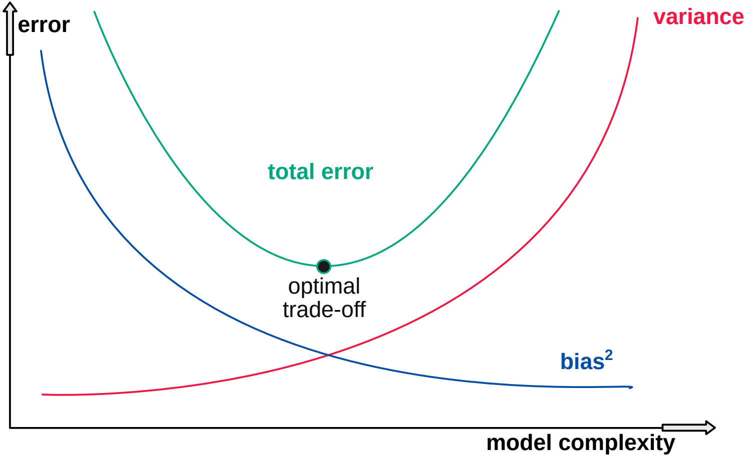
VARIANCE-BIAS
TRADE-OFF

High bias

Worst case
(try again)

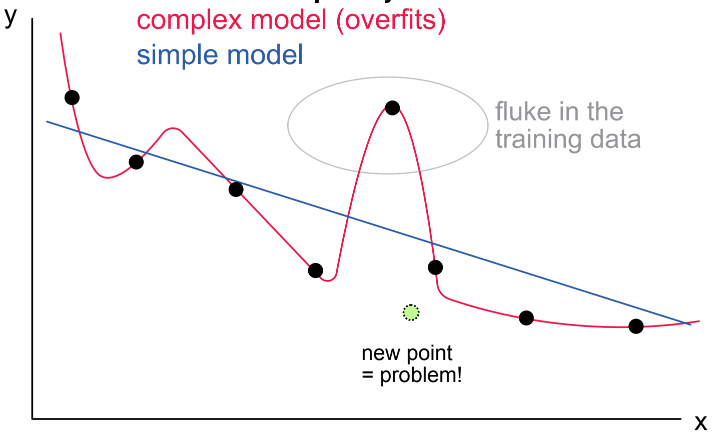
Graphically (2/3)

Finding the **overall minimum** is the goal! (the function may not be convex...)



Graphically (3/3) - overfitting!

Empirically, this is linked to **model complexity**.



fluke in the training data

new point
= problem!

→ A complex model will catch the **patterns** in the sample, but may fail to **generalise** out-of-sample. The simple model has a possible bias, but small variance; the complex model has zero bias, but a larger variance.

Illustration

The ridge regression

We work with $\boldsymbol{y} = \boldsymbol{Xb} + \epsilon$ with $\mathbb{E}[\epsilon] = 0$ and $\mathbb{V}[\epsilon] = \mathbb{E}[\epsilon\epsilon'] = \sigma^2\boldsymbol{I}$. Under a simple OLS regression: $\hat{\boldsymbol{b}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$, with

$$\mathbb{E}[\hat{\boldsymbol{b}}] = \boldsymbol{b} \text{ and } \mathbb{V}[\hat{\boldsymbol{b}}] = \sigma(\boldsymbol{X}'\boldsymbol{X})^{-1}$$

and if the regression is penalised, seeking

$$\min_{\boldsymbol{\beta}} \, (\boldsymbol{Y} - \boldsymbol{Xb})'(\boldsymbol{Y} - \boldsymbol{Xb}) + \lambda||\boldsymbol{b}\,||_2^2,$$

then

$$\mathbb{E}[\hat{\boldsymbol{b}}_\lambda] = \boldsymbol{b} - \lambda(\boldsymbol{X}'\boldsymbol{X} + \lambda\boldsymbol{I}_N)^{-1}\boldsymbol{b}$$

$$\mathbb{V}[\hat{\boldsymbol{b}}_\lambda] = \sigma^2(\boldsymbol{X}'\boldsymbol{X} + \lambda\boldsymbol{I}_N)^{-1}\boldsymbol{X}'\boldsymbol{X}(\boldsymbol{X}'\boldsymbol{X} + \lambda\boldsymbol{I}_N)^{-1}$$

In most (well behaved) cases, the bias of $\hat{\boldsymbol{b}}_\lambda$ increases with λ. In contrast, its variance decreases with λ. In the limiting case $\lambda \to \infty$, the variance is null and the model is **constant**.

Handling the risk of overfitting

Resisting (too) complex models

- training is always performed on **known/past data**. Even if the size of the dataset is large, it is always possible to imagine a model that can capture the idiosyncrasies of each occurrence (e.g., consider a neural net with millions of parameters). Now, this is not a good idea because what we want is the model to learn the patterns that will continue to occur in the future and not the noise pertaining to the training sample.
- How? Impose simplification constraints:
  - norm penalisation or weight constraints
  - dropout
  - impose a (small) learning rate
  - use small tree depth

Overfitting is the shortest way towards **false positives**: strategies that seem to work in sample but fail in live trading.

A taxonomy of overfitting

From: Benign, Tempered, or Catastrophic: A Taxonomy of Overfitting

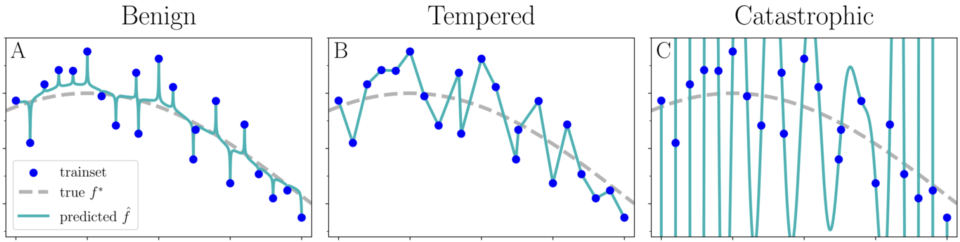
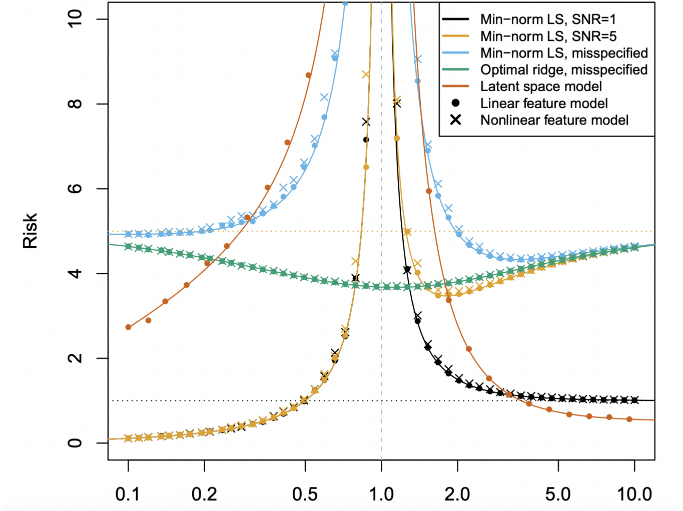


Figure 1: **As $n \to \infty$, interpolating methods can exhibit three types of overfitting.** **(A)** In *benign overfitting*, the predictor asymptotically approaches the ground-truth, Bayes-optimal function. Nadaraya-Watson kernel smoothing with a singular kernel, shown here, is asymptotically benign. **(B)** In *tempered overfitting*, the regime studied in this work, the predictor approaches a constant test risk greater than the Bayes-optimal risk. Piecewise-linear interpolation is asymptotically tempered. **(C)** In *catastrophic overfitting*, the predictor generalizes arbitrarily poorly. Rank-n polynomial interpolation is asymptotically catastrophic.

Double descent

From: Surprises in High-Dimensional Ridgeless Least Squares Interpolation



Agenda

1. Performance metrics in ML
2. Bias-variance tradeoff
3. Tools
4. Practical considerations
5. Wrap-up

supervised
learning
factor
investing

The curse of dimensionality

Many degrees of freedom when it comes to backtesting

Assuming that early (**HUGE**) work has been done (investment universe, dataset, features/labels + engineering), there remains to set:

1. the ML family (regression, tree, NN, others (SVM), etc.)
2. the hyperparameters (handling the validation)
3. the translation of signal into portfolio weights

The (true) best combination of choices is sadly out of reach.

We deal with the second point below (and in this session).

What's a **hyperparameter** (HP)?
It's a parameter that **does not change during the training**. Weights in neural nets and splits in trees are determined during the learning. They are simple parameters. Hyperparameters define the **structure** and some properties of the prediction tool.

Examples of hyperparameters

Each family has at least a few of them

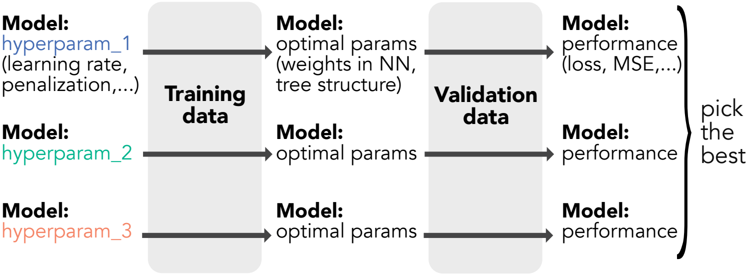
- **trees**: complexity and max depth (nb levels)
- **random forests**: number of trees, tree complexity, number of features per tree, size of training sample
- **boosted trees**: number of trees, learning rate, regularisers, max depth
- **neural networks**:
  - NN structure (feed fwd vs recurrent), nb layers, nb units, activations
  - loss function
  - training pars: epochs, batch size
  - initialisation choices
  - regularisers + (hard) norm constraints
  - dropout

The list is not exhaustive.

The validation set

The **validation set** is used to adjust the choice of hyperparameters.

- The model is trained on the the usual training set but its performance is assessed on unseen data (the validation set).
- The search for proper hyperparameters is not done on the training data because this would only increase the risk of non-benign overfitting.



Model: hyperparam_1 (learning rate, penalization,...) → **Training data** → **Model:** optimal params (weights in NN, tree structure) → **Validation data** → **Model:** performance (loss, MSE,...)

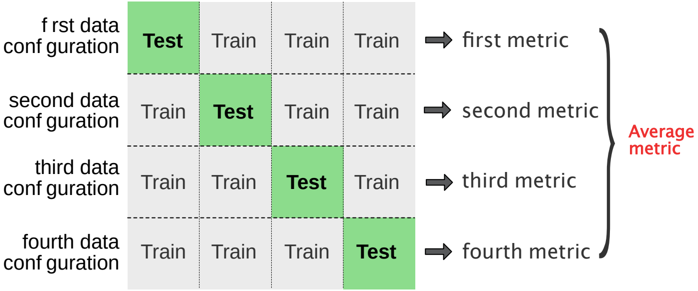
Model: hyperparam_2 → **Model:** optimal params → **Model:** performance

Model: hyperparam_3 → **Model:** optimal params → **Model:** performance

pick the best

The ‘full’ model can then be tested (out-of-sample) on the so-called **testing set**.

Cross-validation



first data configuration	**Test**	Train	Train	Train	➡ first metric
second data configuration	Train	**Test**	Train	Train	➡ second metric
third data configuration	Train	Train	**Test**	Train	➡ third metric
fourth data configuration	Train	Train	Train	**Test**	➡ fourth metric

Average metric

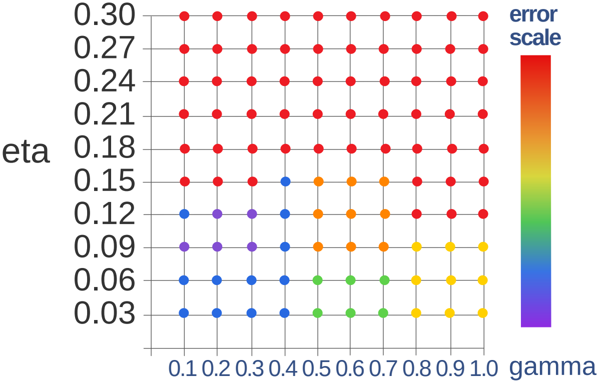
Split the sample into *k* groups. Sequentially, each group will serve as testing/validating set while the rest of data is used for training. Hence, a validation metric is computed and can be averaged over each testing group. → problem in Finance: **chronology**! You can't test on data prior to training set. This would be **forward looking**: not available in practice.

→ dates are obviously ordered, which is why bootstrapping is probably not a great idea.

→ CV is nonetheless possible if you consider **splits** of the training data!

Grid search

This technique is used when looking for the best values for a set of parameters. Unfortunately, it can be very costly in computation time if the mesh is too fine or if the number of dimensions is too large. The idea is to exhaustively span the **parameter space**.



Other methods

Several options

- **Random search**: same as grid, but random
- **Bayesian optimisation**: assume random loss and update posterior distribution for each new point; each new point is carefully chosen to optimise a so-called acquisition function (the aim is to find a new point that will optimise gain compared to the current minimum).
  → not obvious to set the prior and to manage the acquisition function
- **Gradient descent** if the gradient w.r.t. the HP is easy to obtain

BUT don't forget: don't spend too much time optimising the HP, because this can ultimately lead to non-benign **overfitting**.
AND: focus on the HP that matter. 4-5+ dimensional searches are hard and often unnecessary.

Sample size trade-off

When computation time is an issue

- a larger (chronologically deeper) training set is preferable because it embeds more information on market states
- but, obviously, the computational cost is larger

Usually, factor-based portfolios are rebalanced every month or quarter, so computation time is not too much of an issue, unless reactivity is required. (this is more an issue for HF trading: microstructure models)

- It can be argued that training models on the **most recent data** me not be a bad idea because the effect of features on returns is time varying and old patterns may not be useful anymore.
- alternatively, practitioners may like to go back in time to aggregate market crashes. This can improve the **generalisation** potential.

(it's largely a matter of beliefs)

Agenda

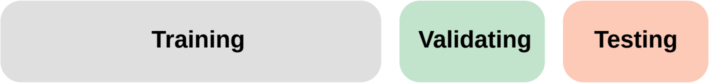
1. Performance metrics in ML
2. Bias-variance tradeoff
3. Tools
4. Practical considerations
5. Wrap-up

Dynamic tuning?

A costly alternative in backtesting:

- cost in terms of samples (is it deep enough?) and computation time (multiplied by the number of testing dates)
- tuned HP may vary over time: is it true? Is it desirable?

Classical ML framework

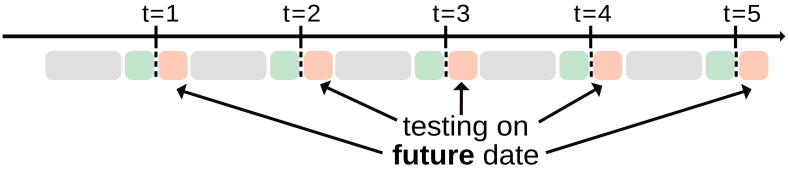


Training

Validating

Testing

Dynamic portfolio backtesting



t=1

t=2

t=3

t=4

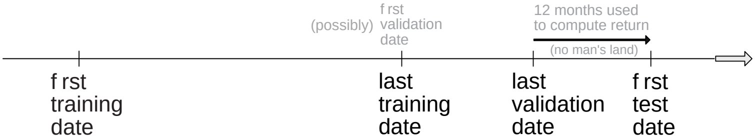
t=5

testing on **future** date

Dynamic tuning II

A few discussion points:

- it's hard to imagine strongly varying hyper-parameters + don't overfit your model by optimising too much!
- **be careful**: it's easy to train data on future points with forward-looking bias. For instance, if the dependent variable is a 12M future performance proxy, then the features of the validation set have to stop 12M before the test sample. Since we are often given a continuum of data values, it's tempting to use OOS points during validation or training.
- to a lesser extent, this is also true between training and validation if the computation of the dependent variable creates an overlap between the two



Agenda

1. Performance metrics in ML
2. Bias-variance tradeoff
3. Tools
4. Practical considerations
5. Wrap-up

Key takeaways

WHAT IS YOUR MODEL LEARNING? True patterns or noise?
Hint: that's hard to know upfront... validating... & testing brings answers!

The art of tuning

1. **validation** is a key step before the test stage: hyperparameter values matter!
2. the important metric is probably different than the loss defined in the training step.
3. it is tempting to **overfit during validation**: resist the temptation!
4. finally, it is sometimes tricky how to carry it out-of-sample in a non-forward-looking fashion; the backtester should be very careful!

Reminder: your project

- **DEADLINE**: June 18th
- **Specs**: see document on brightspace
- What matters:
  1. code, originality
  2. **comments** (not only about code), interpretation, guidance for the reader

Thank you for your attention

Any questions?