MS&E 228 (Winter 2026) — Lecture 5 Notes (Student Handout)

Prediction with Modern ML for Downstream Causal Estimation

Vasilis Syrgkanis
Stanford University

January 24, 2026

Readings. *Applied Causal Inference Powered by ML and AI*, §1, §3, §8 (and references therein).

1 Why prediction matters for doubly robust causal estimation

In the previous lecture, we introduced the doubly robust (DR) estimator for the average treatment effect (ATE). We saw how it combines a plug-in g-formula estimator with an IPW-style correction term to achieve desirable statistical properties. A key message was that the DR estimator's bias depends on the *product* of the estimation errors of the outcome regression and the propensity:

$$g_0(d,x) = \mathbb{E}[Y \mid D=d, X=x] \qquad \text{(outcome regression)}$$
$$p_0(x) = \mathbb{E}[D \mid X=x] = \Pr(D=1 \mid X=x) \qquad \text{(propensity)}$$

as measured by the corresponding root-mean-squared-errors:

$$\mathrm{RMSE}(\hat{g}) = \sqrt{\mathbb{E}[(\hat{g}(D,X) - g_0(D,X))^2]} \qquad \mathrm{RMSE}(\hat{p}) = \sqrt{\mathbb{E}[(\hat{p}(X) - p_0(X))^2]}$$

This product structure means that we don't need both estimators to be perfect; we just need the product of their average squared errors to be small enough. This lecture is all about how to make those errors small using modern machine learning.

Discussion

A Guiding Rule of Thumb (from DR Theory).
A recurring sufficient condition for the DR estimator to be asymptotically normal and root-n consistent is that the product of the root-mean-squared errors of the outcome regression and the propensity shrinks faster than $n^{-1/2}$:[a]

$$\sqrt{n} \cdot \mathrm{RMSE}(\hat{g}) \cdot \mathrm{RMSE}(\hat{p}) \approx 0.$$

This is often satisfied if each model has an RMSE that goes down to zero at a rate that is strictly better than $O(n^{-1/4})$. This $n^{-1/4}$ rate is a very useful mental benchmark. It tells us that we don't need parametric-speed estimators for our outcome regression and propensity models; we can use flexible, non-parametric ML methods, as long as they converge reasonably fast. Today, we'll explore which methods can achieve this and why.

[a]We remind that in the course $\approx$ is used for convergence in probability and $\overset{a}{\sim}$ for convergence in distribution.

2 Conditional expectations as best prediction rules

2.1 The conditional expectation function (CEF)

Both of our target models, the outcome regression $g_0(d, x)$ and the propensity score $p_0(x)$, are examples of a more general object: the **conditional expectation function (CEF)**. Given a random variable Y and a set of covariates $Z \in \mathcal{Z}$, the CEF is defined as:

$$f_0(z) := \mathbb{E}[Y \mid Z = z].$$

For the outcome regression, Y is the outcome and $Z = (D, X)$. For the propensity score, Y is the treatment indicator D and $Z = X$. The CEF is the fundamental object we need to learn from data.

2.2 Population risk minimization: CEF is the best predictor

It turns out that the CEF is also the *best possible predictor* of Y given Z if our goal is to minimize the mean squared prediction error (MSPE). Let's formalize this. For any measurable predictor $f : \mathcal{Z} \to \mathbb{R}$, define the mean squared prediction error (MSPE) as:

$$\text{MSPE}(f) := \mathbb{E}\big[(Y - f(Z))^2\big].$$

Theorem 1 (Best prediction rule under squared loss)**.** *The conditional expectation $f_0(Z) = \mathbb{E}[Y \mid Z]$ is a minimizer of* $\text{MSPE}(f)$ *over all measurable functions f. Any minimizer must agree with f_0 on all but measure zero sets.*

Proof. Let $f_0(Z) = \mathbb{E}[Y \mid Z]$. The proof relies on a simple but powerful decomposition. Let's add and subtract $f_0(Z)$ inside the square and then expand the square:

$$\begin{aligned}\mathbb{E}[(Y - f(Z))^2] &= \mathbb{E}[(Y - f_0(Z) + f_0(Z) - f(Z))^2] \\ &= \mathbb{E}[(Y - f_0(Z))^2] + \mathbb{E}[(f_0(Z) - f(Z))^2] + 2\mathbb{E}\big[(Y - f_0(Z))(f_0(Z) - f(Z))\big].\end{aligned}$$

The first term, $\mathbb{E}[(Y - f_0(Z))^2]$, is the irreducible error, or the variance of the outcome that cannot be explained by Z. It does not depend on our choice of predictor f. The second term, $\mathbb{E}[(f_0(Z) - f(Z))^2]$, is the squared error of our predictor f relative to the true CEF f_0. This is the term we want to minimize.

What about the cross-product term? Let's use the law of iterated expectations (tower property). First, we condition on Z:

$$\mathbb{E}\big[(Y - f_0(Z))(f_0(Z) - f(Z)) \mid Z\big] = (f_0(Z) - f(Z))\mathbb{E}[Y - f_0(Z) \mid Z].$$

By the definition of the CEF, $\mathbb{E}[Y \mid Z] = f_0(Z)$, so $\mathbb{E}[Y - f_0(Z) \mid Z] = 0$. Therefore, the entire cross-product term is zero. This leaves us with:

$$\mathbb{E}[(Y - f(Z))^2] = \mathbb{E}[(Y - f_0(Z))^2] + \mathbb{E}[(f_0(Z) - f(Z))^2].$$

To minimize this expression, we only need to minimize the second term, which is non-negative. The minimum is achieved when $f(Z) = f_0(Z)$, which makes the second term zero. If f differs with f_0 on some non-measure zero set, then the second term is strictly positive. Thus any such f cannot be a minimizer. This completes the proof. □

Discussion

The Bridge Between Causal Inference and Machine Learning. This theorem is the fundamental bridge to the world of supervised machine learning. It tells us that the problem of estimating the functions g_0 and p_0 is equivalent to finding the best predictor for the outcome Y and the treatment D under a squared loss. This is precisely the problem that modern ML algorithms are designed to solve.

2.3 Finite samples: empirical risk minimization (ERM)

In the real world, we don't have access to the true data-generating process, so we can't minimize the population MSPE directly. Instead, we use a finite sample of data $(Y_i, Z_i)_{i=1}^n$ and minimize the **empirical risk**, which is just the sample average of the loss:

$$\hat{f} \in \arg\min_{f \in \mathcal{F}} \mathbb{E}_n[(Y - f(Z))^2] := \frac{1}{n}\sum_{i=1}^{n}(Y_i - f(Z_i))^2,$$

Here, $\mathcal{F}$ is the class of functions we are searching over (e.g., linear models, decision trees, neural networks). This strategy is called **Empirical Risk Minimization (ERM)**. In modern ML, we almost always add *regularization* to this objective to prevent overfitting and improve generalization to unseen data.

Remark

Population vs. Training Error. It is crucial to remember that for the purposes of doubly robust causal estimation, we care about the *population* error of our models, i.e., $\mathbb{E}[(\hat{g}(D, X) - g_0(D, X))^2]$ and $\mathbb{E}[(\hat{p}(X) - p_0(X))^2]$. A model that has very low training error (in-sample error) might have a very high population error (out-of-sample error) if it has overfit the training data. This is why techniques like cross-validation are so important for choosing and evaluating our models.

So when can and how can we argue that some variant of empirical risk minimization guarantees a small population RMSE?

3 Linear models and what breaks in high dimension

3.1 Linear regression as restricted best prediction

Let's start with the simplest case: linear models. Suppose we make the assumption that $\mathbb{E}[Y \mid Z]$ is linear in some set of known engineered features $\phi(Z) \in \mathbb{R}^p$, i.e.

$$f_0(Z) = a_0^\top \phi(Z).$$

Then when minimizing the MSPE it suffices to restrict our function class $\mathcal{F}$ to be linear $\phi(Z)$:

$$f(Z) = a^\top \phi(Z).$$

The population best linear predictor is the one that minimizes the MSPE over all possible linear coefficients a:

$$a_0 \in \arg\min_{a \in \mathbb{R}^p} \mathbb{E}[(Y - a^\top \phi(Z))^2],$$

and the empirical analogue is the familiar Ordinary Least Squares (OLS) estimator:

$$\hat{a} \in \arg\min_{a \in \mathbb{R}^p} \; \mathbb{E}_n[(Y - a^\top \phi(Z))^2], \qquad \hat{f}(Z) := \hat{a}^\top \phi(Z).$$

3.2 Low-dimensional accuracy and the role of p

In classical statistical regimes, where the number of features p is much smaller than the number of samples n ($p \ll n$), OLS works very well. Under standard assumptions, the error of the OLS estimator scales like:

$$\mathbb{E}\big[(\hat{a}^\top \phi(Z) - a_0^\top \phi(Z))^2\big] \asymp \frac{p}{n}, \qquad \text{so RMSE}(\hat{f}) \asymp \sqrt{\frac{p}{n}}.$$

This tells us that the error depends on the ratio of features to samples. If p is fixed, we get the parametric $n^{-1/2}$ rate. But if p grows with n, the rate slows down. This makes it explicit that the effective number of parameters matters.

3.3 High-dimensional regime: interpolation and overfitting

What happens when the number of features p is greater than or equal to the number of samples n? In this high-dimensional regime, OLS breaks down. If we let Φ be the $n \times p$ matrix of features, we can often find coefficients a that perfectly *interpolate* the training data:

$$\Phi a = Y, \quad \text{where} \quad \Phi = \begin{bmatrix} \phi(Z_1)^\top \\ \vdots \\ \phi(Z_n)^\top \end{bmatrix} \in \mathbb{R}^{n \times p}.$$

This means the training error is zero! However, this is usually a disaster for prediction. The model has fit the noise in the training data, and it will likely generalize very poorly to new data. Moreover, when $p > n$, there are typically infinitely many such interpolating solutions, and the one returned by a standard OLS solver (the minimum ℓ_2-norm solution) has no reason to be close to the true underlying function. Hence, in such regimes, we should not expect plain OLS to guarantee RMSE convergence rates, as required for instance by the doubly robust theorem.

Quick Check

When $p > n$ and we run plain OLS on engineered features, what is the typical failure mode?

1. Underfitting (the model is too simple)
2. Overfitting (the model has great training error, but poor test error)
3. No solution exists

4 Regularization: inducing structure for generalization

To make progress in the high-dimensional setting, we need to introduce some form of **regularization**. Regularization is a way of adding an *inductive bias* to the learning process, which is a preference for certain types of solutions over others.

4.1 Penalized ERM

Instead of just minimizing the empirical risk, we add a penalty term to the objective function that discourages complexity:

$$\hat{a} \in \arg\min_{a} \ \mathbb{E}_n[(Y - a^\top \phi(Z))^2] + \lambda R(a),$$

where $R(a)$ is a norm or seminorm that encodes our inductive bias (i.e., we believe that the true parameter a_0 has small $R(a_0)$), and λ is a tuning parameter that controls the strength of the penalty. Common choices for $R(a)$ include:

- **Lasso**: $R(a) = \|a\|_1$. This encourages sparsity, meaning that many of the coefficients in $\hat{a}$ will be exactly zero. This penalty encodes our inductive bias that we believe that the true parameter a_0 has only $s \ll p$ non-zero entries, in which case, we expect $\|a\|_1 = O(s) \ll p$.
- **Ridge**: $R(a) = \frac{1}{2}\|a\|_2^2$. This encourages small, dense coefficients in which case our inductive bias is that we believe that the true parameter a_0 has bounded ℓ_2-norm, $\|a_0\| \leq B \ll p$.
- **Elastic Net**: $R(a) = \gamma\|a\|_1 + (1-\gamma)\frac{1}{2}\|a\|_2^2$. A combination of the ℓ_1 and ℓ_2 penalties, which works well if either the true parameter is sparse or if it is dense.

Remark

The Bias-Variance Trade-off. Regularization is a classic example of the bias-variance trade-off. By adding the penalty, we are introducing some bias into our estimator (it's no longer the unconstrained minimizer of the empirical risk). However, in return, we dramatically reduce the variance of the estimator, leading to much better generalization performance.

4.2 Headline rates (why they are dimension-friendly)

The great advantage of regularized estimators is that their convergence rates are much less sensitive to the ambient dimension p. For example:

Lasso. If the true coefficient vector a_0 is s-sparse (meaning it has only $s \ll p$ non-zero entries), then under appropriate conditions, the Lasso estimator can achieve an RMSE of:

$$\mathrm{RMSE}(\hat{f}) \lesssim \sqrt{\frac{s \log p}{n}}.$$

The key is that the dependence on the total number of features p is only logarithmic, while the rate depends primarily on the sparsity s. This allows us to handle very high-dimensional data as long as the underlying structure is sparse.

Ridge. If we assume the true coefficient vector has a bounded ℓ_2 norm, $\|a_0\|_2 \leq B$, then ridge regression can deliver rates on the order of:

$$\mathrm{RMSE}(\hat{f}) \lesssim \left(\frac{B}{n}\right)^{1/4}.$$

This rate is slower than the parametric rate, but it does not explicitly depend on the dimension p at all. This is exactly at the cusp of the kind of $n^{-1/4}$ rate that we need for our DR estimator to work well.[1]

[1]Note that for the validity of the confidence intervals of the doubly robust estimator we need the product to decay strictly faster than $n^{-1/2}$. So if, for instance, the outcome regression achieves a rate of exactly $n^{-1/4}$, as is provided by the Ridge bound above, then the propensity function should achieve strictly faster than $n^{-1/4}$ rates.

4.3 Misspecification: approximation + estimation error

What if the true CEF f_0 is not actually linear in our features $\phi(Z)$? In this case, our linear model is *misspecified*. We can decompose the total error into two parts: approximation error and estimation error. Let $f_\star(Z) = a_\star^\top \phi(Z)$ be the best possible linear approximation to f_0 in the population, i.e.,

$$a_\star = \arg\min_a \mathbb{E}[(f_0(Z) - a^\top \phi(Z))^2], \qquad\qquad f_\star(Z) := a_\star^\top \phi(Z)$$

Then:

$$\mathbb{E}[(f_0(Z) - \hat{f}(Z))^2] \leq \underbrace{\mathbb{E}[(f_0(Z) - f_\star(Z))^2]}_{\text{approximation (APX)}} + \underbrace{\mathbb{E}[(f_\star(Z) - \hat{f}(Z))^2]}_{\text{estimation (EST)}}.$$

This is a very useful template for thinking about model selection. We want to choose a function class that is rich enough to make the approximation error small, but simple enough (or regularized enough) that we can keep the estimation error small. Moreover, if $a_\star$ satisfies the inductive biases we described in the previous section, then the EST term is upper bounded by the rates provided in the previous section. So for instance, we can consider a high-dimensional and growing (in the samples) set of engineered features $\phi(Z)$. If we believe that the population best linear predictor $a_\star$ is s-sparse, then our overall RMSE will be of the order of:

$$\sqrt{\frac{s\log(p)}{n}} + \sqrt{\text{APX}} \tag{1}$$

If we further believe that this growing set of engineered features well approximates the true CEF and the approximation error APX goes to zero faster than $n^{-1/4}$, then we get the desired rate.

5 Nonlinear prediction and the curse of dimensionality

Linear models are a good starting point, but what if the true CEF is highly nonlinear? We could try to use a very flexible, non-parametric estimator. However, this runs into a fundamental problem known as the **curse of dimensionality**.

5.1 Worst-case smoothness is too weak in moderate dimension

If we only assume that the true CEF f_0 is, say, β-smooth (meaning it has bounded continuous derivatives up to order β) in p dimensions, then minimax theory tells us that for *any* estimator (not necessarily ERM based), the best possible RMSE we can hope for is:

$$\text{RMSE}(\hat{f}) \gtrsim n^{-\beta/(2\beta+p)}.$$

This rate is disastrously slow. For example, if we have $p = 10$ features and assume the function is once-differentiable ($\beta = 1$), the rate is $n^{-1/12}$. To get an RMSE of just 0.1, we would need a sample size on the order of $10^{12} = 1$ trillion! This tells us that smoothness alone is not a strong enough assumption to enable efficient learning in even moderately high dimensions.

Focal Point: Key Message

To get fast rates of convergence in practice, we need to assume that the true CEF has some **additional structure** beyond just smoothness. This is the key insight that allows modern ML methods to "bypass" the curse of dimensionality.

5.2 How modern ML "bypasses" the curse: adaptivity to effective complexity

Modern ML methods are best understood as being **adaptive** to some notion of *effective complexity* that is much smaller than the ambient dimension p. Different methods are designed to exploit different types of structure:

- **Lasso** adapts to sparse linear structure.
- **Decision trees, random forests, and boosting** are good at capturing partition structure and low-order interactions.
- **Neural networks** are good at learning representations and exploiting compositional structure.
- **Nearest neighbor methods** work well when the data has a low intrinsic dimension (e.g., lies on a manifold).

6 A practical menu of modern ML methods

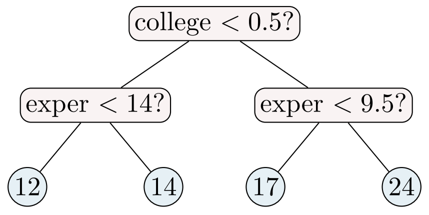
Let's briefly review some of the most popular and effective ML methods for solving best prediction rule problems (equiv. CEF problems). To familiarize yourselves with all these methods, this Colab Notebook implements all the methods discussed in this lecture and showcases how it performs on a simple data generating process across various hyperparameter configurations.

6.1 Decision trees

A regression tree works by recursively partitioning the feature space into a set of hyper-rectangles (leaves) and then predicting the average outcome within each leaf:

$$\hat{f}(z) = \sum_{\ell=1}^{L} \hat{\mu}_\ell \, \mathbb{I}\{z \in \text{leaf } \ell\}.$$

These hyper-rectangles can be represented as a tree where on each node, we decide whether to split the covariate space based on whether some features takes value smaller than some threshold.



Trees are great at capturing nonlinearities and interactions automatically. The partitioning of the space is data-driven and chosen so as to minimize prediction error (typically in a greedy manner). However, they are also known to be unstable and have high variance; small changes in the training data can lead to very different trees. The typical hyperparameters the control performance of decision trees is the max depth and the minimum sample size of a leaf. Larger depth leads to better approximation power but also higher variance, as leafs have fewer samples. A small `max_depth` leads to high bias (underfitting), as the model is too simple to capture the underlying function. A large (or unlimited) `max_depth` leads to high variance (overfitting), as the model starts fitting the noise in the training data.

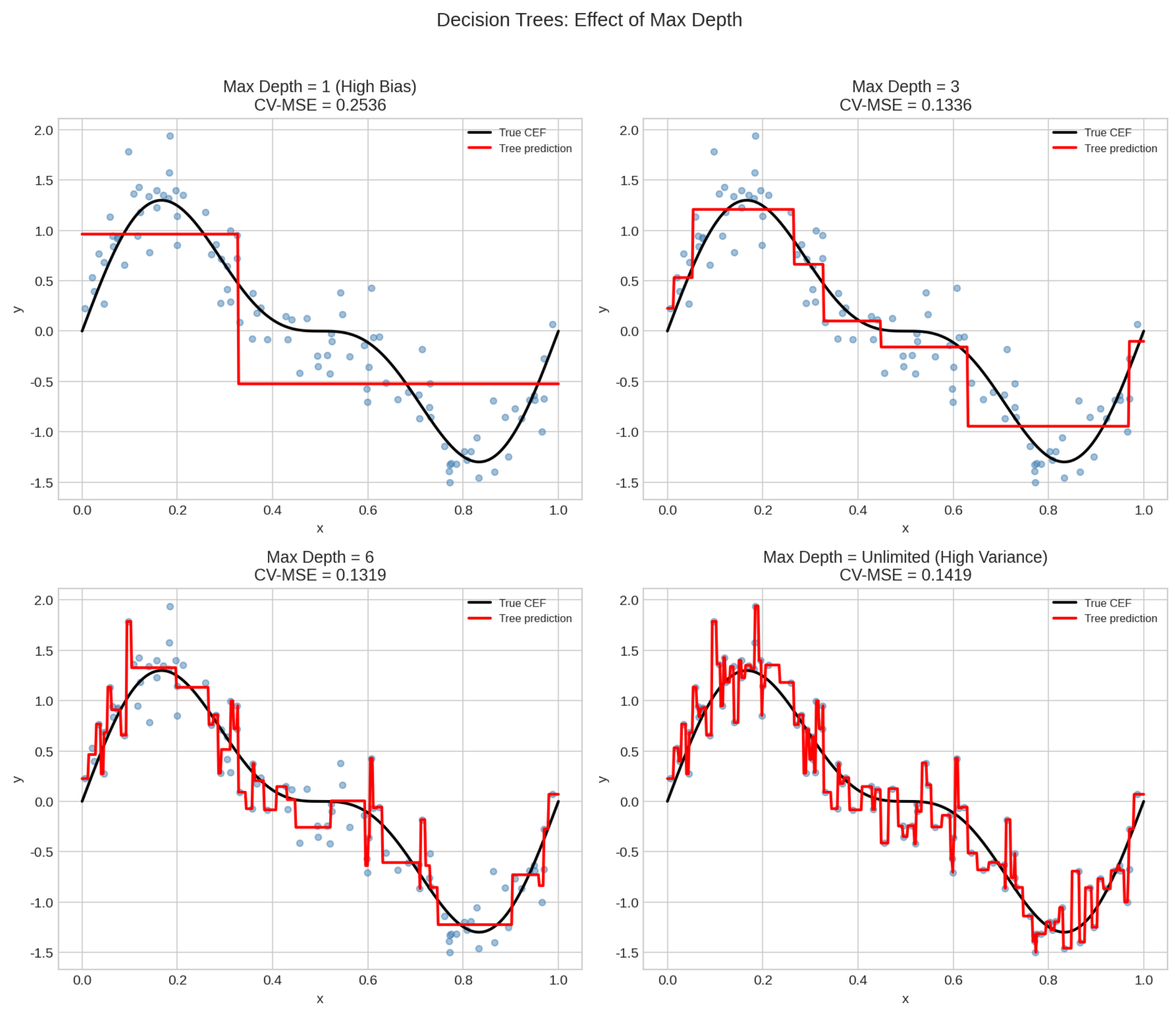


Figure 1: Effect of `max_depth` on decision tree predictions. A depth of 1 is too simple, while an unlimited depth clearly overfits. A depth of 6 provides a reasonable fit.

6.2 Random forests

Random forests are a way to address the high variance of individual decision trees. They do this by building many trees on bootstrap samples of the data (a technique called **bagging**) and then averaging their predictions:

$$\hat{f}_{\mathrm{RF}}(z) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(z),$$

where each $\hat{f}_b$ is a randomized tree. The randomization comes from both the bootstrap sampling and from considering only a random subset of features at each split. This averaging process dramatically reduces the variance and makes the predictions much more stable. Adding more trees reduces the variance of the prediction without increasing the bias. The prediction becomes smoother and more stable. Random forests are an excellent default choice for tabular data.

Bagging vs. Subsampling. The process of creating these data subsets is done in one of two ways: **bagging**, where we sample n data points with replacement (meaning some points may appear multiple times), or **subsampling**, where we sample $s < n$ points without replacement. While most theoretical results for random forests are based on subsampling, most practical implementations (including scikit-learn) use bagging.

Pseudo-code for fitting Random Forests

```
from sklearn.ensemble import RandomForestRegressor,

# n_estimators, max_depth and min_samples_leaf are key parameters!
rf = RandomForestRegressor(n_estimators=500, max_depth=5, min_samples_leaf=5)
rf.fit(Z_train, Y_train)
```

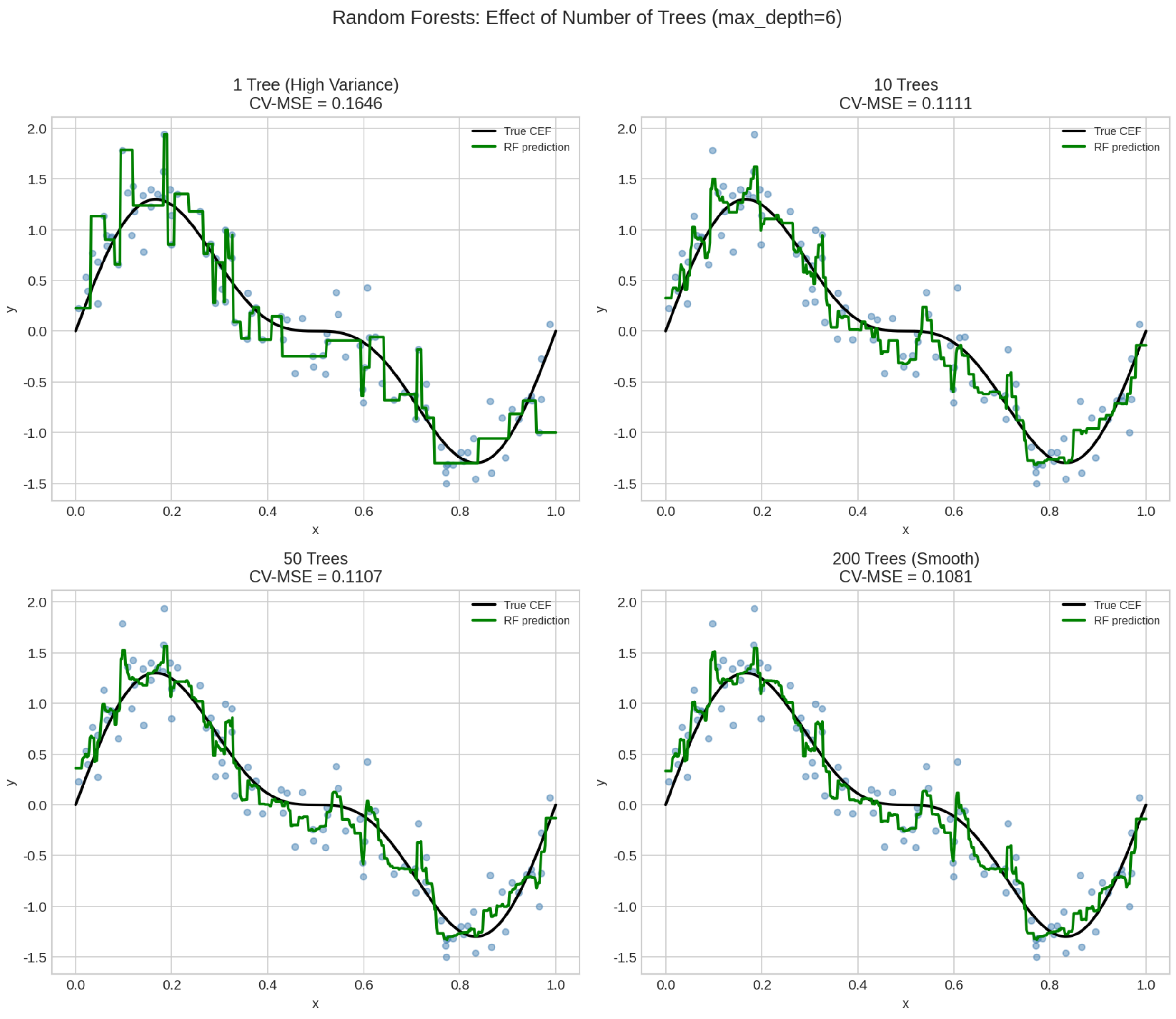


Figure 2: Effect of `n_estimators` (number of trees) on random forest predictions. With just one tree, the prediction is noisy and identical to the single decision tree. As more trees are added, the prediction becomes progressively smoother.

Remark

Effective Dimension for Trees and Forests. Similar to the lasso guarantees, one can argue provable rates for decision trees and random forests that adapt to sparsity, without the linearity assumption. Suppose the true CEF depends only on $s \ll p$ relevant variables. This s is an effective dimension. It has been shown (Syrgkanis & Zampetakis, 2020) that regression trees can achieve an RMSE of roughly $\sqrt{\frac{2^s \log p \log n}{n}}$. This rate is exponential in the effective dimension s, but only logarithmic in the ambient dimension p.

6.3 Gradient boosting (GB)

Gradient boosting is a powerful ensemble method that builds an additive model in a sequential fashion. Unlike random forests, which build trees in parallel, boosting is a sequential learner that gradually improves its predictions at each step.

The Additive Model. The core idea is to build a model of the form:

$$\hat{f}_M(z) = \sum_{m=0}^{M} \nu\, h_m(z),$$

where the initial model $h_0(z)$ is typically a simple constant (e.g., the mean of the outcome), and each subsequent model $h_m(z)$ is a simple learner (usually a shallow decision tree) trained to correct the errors of the previous ones. The parameter ν is a **learning rate** that shrinks the contribution of each new tree.

Fitting the Residuals. In the simplest case of squared error loss, each new tree $h_m(z)$ is trained to predict the residuals from the current ensemble:

$$r_{im} = Y_i - \sum_{j=0}^{m-1} \nu h_j(Z_i).$$

This is a very intuitive process: at each stage, we fit a new model to capture the part of the outcome that the current ensemble gets wrong.

Regularization. Boosting models can be very prone to overfitting if not properly regularized. The most important regularization techniques are:

- **Learning Rate ν:** A small learning rate (e.g., 0.01 to 0.1) reduces the contribution of each tree and requires more trees to be added, but generally leads to better generalization.
- **Tree Depth:** Limiting the maximum depth of the individual trees (e.g., to 4-8 levels) prevents them from capturing spurious interactions in the data.
- **Early Stopping:** We can monitor the model's performance on a validation set and stop training when the performance stops improving. This is one of the most crucial forms of regularization for boosting.

Pseudo-code for fitting Gradient Boosting

```python
from sklearn.ensemble import GradientBoostingRegressor

# n_estimators, learning_rate and max_depth are key tuning parameters!
gbdt = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3)
gbdt.fit(Z_train, Y_train)
```

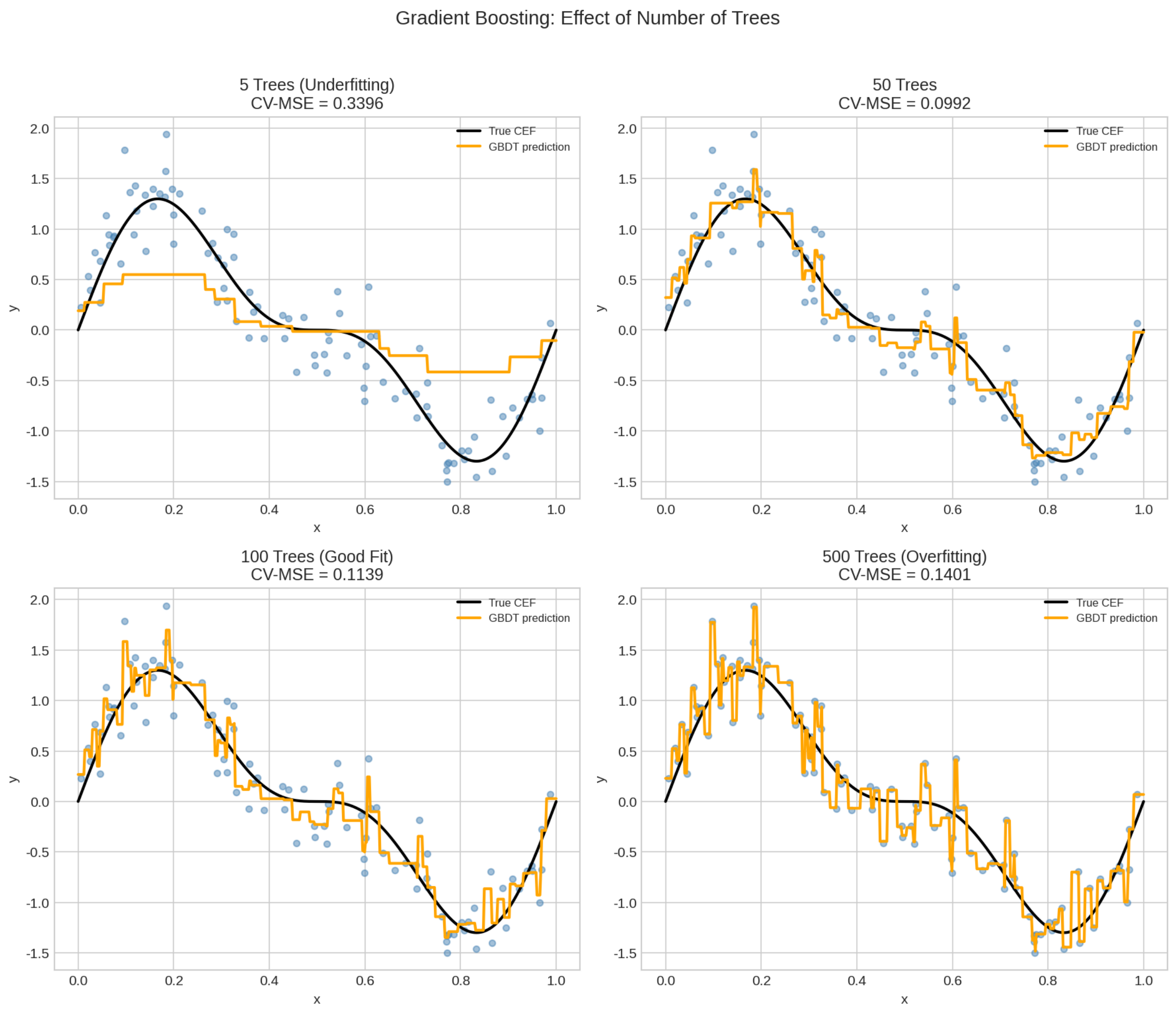


Figure 3: Effect of `n_estimators` on gradient boosting predictions. With 5 trees, the model underfits. With 100 trees, it provides a good fit. With 500 trees, it starts to overfit the training data, capturing noise.

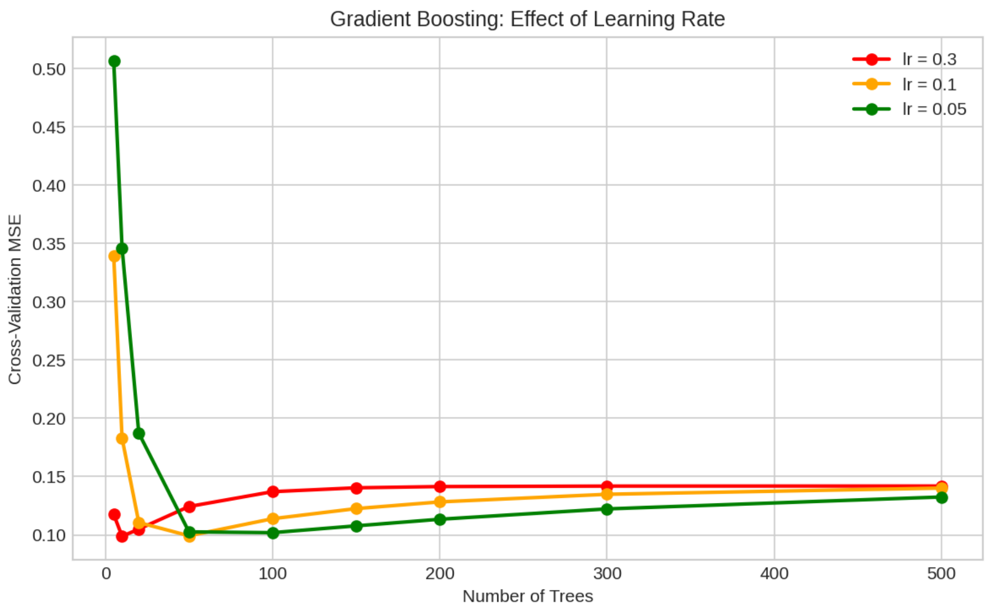


Figure 4: Effect of learning rate on gradient boosting performance. Smaller learning rates require more trees but can lead to better final models.

Discussion

Random Forests vs. Gradient Boosting: A Tale of Two Ensembles.
At first glance, random forests and gradient boosting seem similar: they both combine many decision trees to make a prediction. However, the way they do so is fundamentally different, leading to important practical consequences.
In a **random forest**, the trees are built independently in parallel. One tree has no knowledge of what the others have learned. This is a powerful variance reduction technique, but it has a weakness: if there is a very strong predictive pattern in the data, many of the trees might discover and model the exact same pattern, failing to explore other, more subtle relationships.
Gradient boosting, on the other hand, builds trees sequentially. Each new tree is explicitly trained to predict the errors (residuals) of the existing ensemble. This means that each tree focuses on the part of the signal that the previous trees missed. This sequential, error-correcting process is what makes boosting so powerful.
This leads to a critical difference in how we should think about the number of trees:

- For **random forests**, more trees are almost always better. Adding more trees continues to average out the noise and smooth the prediction function, reducing variance without increasing bias.

- For **gradient boosting**, more trees can be worse. After a certain point, the model will have captured the main signal, and subsequent trees will start to fit the random noise in the residuals. This leads to overfitting. This is why **early stopping** is the most important form of regularization for boosting.

This also explains why shallow trees are acceptable in boosting but not in random forests. In a random forest, a shallow tree is a high-bias model. In boosting, the bias of a shallow tree is systematically corrected by the subsequent trees that are added to the ensemble.

6.4 Neural networks

Neural networks are a class of models inspired by the structure of the human brain. They are composed of interconnected nodes, or "neurons," organized in layers. A typical feedforward neural network consists of an input layer, one or more hidden layers, and an output layer.

Representation Learning. The key idea behind deep learning is **representation learning**. Each layer of the network transforms its input into a slightly more abstract representation. The first layer might learn to recognize simple patterns (like edges in an image), the next layer might combine those patterns into more complex objects (like eyes or noses), and so on. We can view the hidden layers as learning a set of features $\phi(z;\alpha)$, where the parameters α of this feature extractor are learned from the data. The final layer is then a simple linear model that makes a prediction based on these learned features:

$$f(z) = \beta^\top \phi(z;\alpha).$$

This process of learning the features and the final model simultaneously is what makes neural networks so powerful.

Training. The parameters of the network (α and β) are learned by minimizing a loss function (like MSE or log-loss) using an algorithm called **backpropagation** combined with **gradient descent**. Backpropagation is an efficient way to compute the gradient of the loss function with respect to every parameter in the network, and gradient descent uses these gradients to iteratively update the parameters to minimize the loss.

Regularization. Like boosting, neural networks have many parameters and can easily overfit. Common regularization techniques include:

- **Weight Decay (L2 Regularization):** This is equivalent to adding a ridge penalty on the weights of the network, encouraging them to be small.
- **Dropout:** During training, randomly set a fraction of the neuron activations to zero at each update. This prevents the network from becoming too reliant on any single neuron and forces it to learn more robust representations.
- **Early Stopping:** Monitor the loss on a validation set and stop training when it begins to increase.

Neural networks are especially powerful for unstructured data like images and text, or when the dataset is very large. For medium-sized tabular datasets, Gradient Boosted Forests and Random Forests are often stronger out-of-the-box because their inductive bias (based on hierarchical partitioning) is often a better fit for this type of data. A common and very effective practical strategy is to use pre-trained neural networks (e.g., from the Hugging Face library) to extract feature representations from unstructured data like text or images, and then use those features as inputs to a Gradient Boosted or Random Forest or Lasso model.

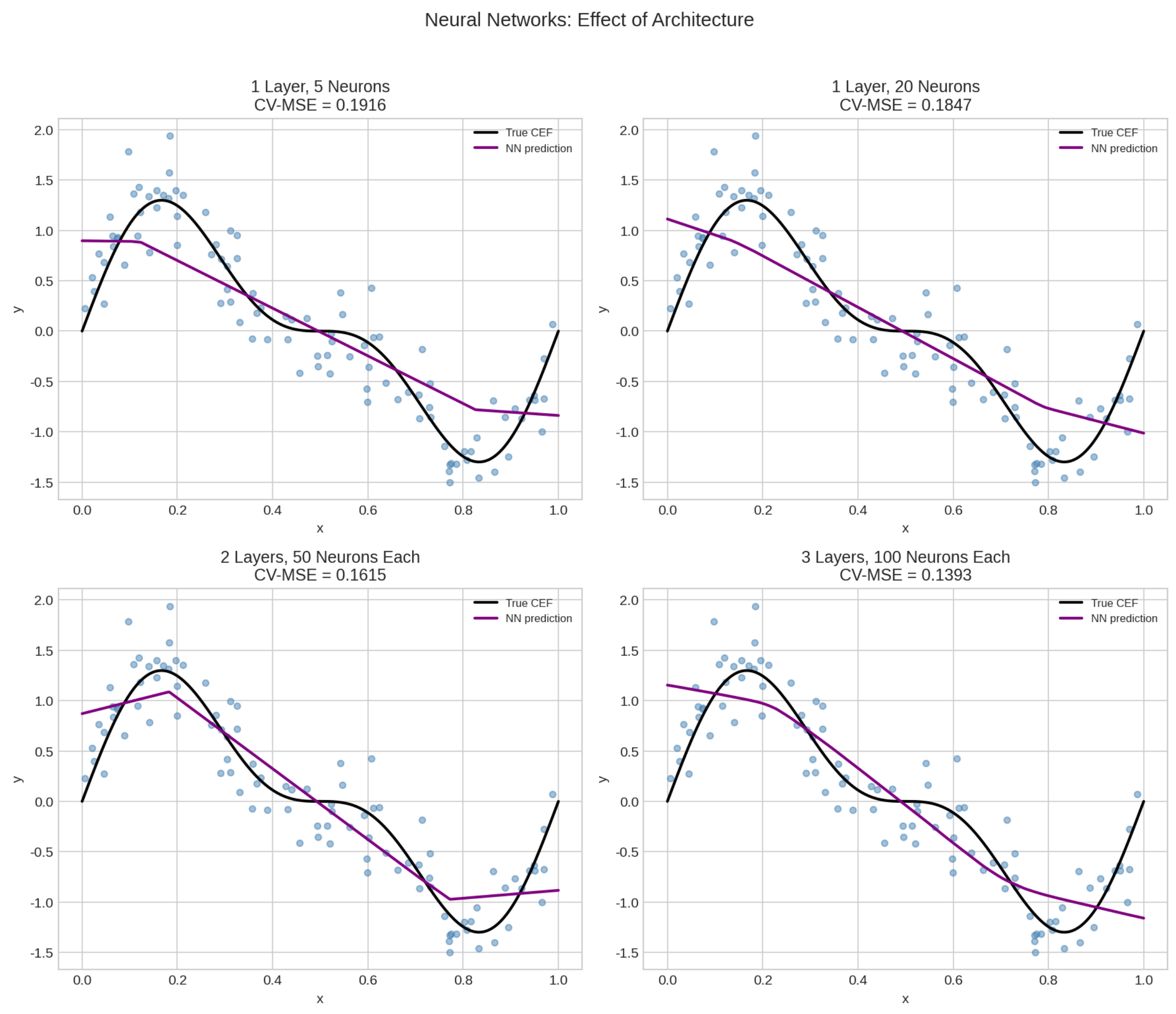


Figure 5: Effect of architecture on neural network predictions. A simple network (5 neurons) underfits, while larger networks (e.g., 2 layers of 50 neurons) can better capture the true function. The predictions are noticeably smoother than tree-based methods.

Pseudo-code for fitting a Neural Network

```
from sklearn.neural_network import MLPRegressor

# architecture (depth and number of neurons in each layer),
# alpha and max_iter are key parameters!
nn = MLPRegressor(hidden_layer_sizes=(50, 50),
                  activation='relu',
                  alpha=0.01, # L2 regularization
                  max_iter=2000,
                  random_state=42)
nn.fit(Z_train, Y_train)
```

7 Model selection: cross-validation, metrics, and AutoML

With so many models and hyperparameters to choose from, how do we select the best one for our problem?

7.1 Cross-validation (CV)

The most important tool for model selection is **cross-validation**. As we've discussed, in-sample error can be a very misleading guide to a model's true performance. K-fold cross-validation provides a much more reliable estimate of the population mean squared prediction error (MSPE). Moreover, better MSPE implies better MSE with respect to the true CEF, which is what we want. The procedure is as follows:

1. Split the data into K folds.
2. For each fold k, train the model on the other $K-1$ folds and make predictions on fold k.
3. Pool the out-of-fold predictions $\hat{Y}_i$ for each sample i and compute an empirical loss.

We then choose the model and hyperparameters that give the best out-of-fold empirical performance.

7.2 Metrics

What loss should we use to evaluate our models?

Outcome regression (continuous Y). For a continuous outcome, the natural choice is the empirical Mean Squared Prediction Error (MSPE) or Root Mean Squared Prediction Error (RMSPE).

$$\text{Empirical MSPE} \coloneqq \mathbb{E}_n[(Y-\hat{Y})^2]$$

A useful normalized version is the R^2 score:

$$R^2 = 1 - \frac{\mathbb{E}_n[(Y-\hat{Y})^2]}{\widehat{\text{Var}}(Y)}.$$

An R^2 of 1 means a perfect fit, while an R^2 of 0 means the model is no better than just predicting the mean of Y. An $R^2 < 0$ means the model is worse than predicting the mean. An R^2 of $x \in (0,1)$ can be interpreted as "the model explained $100x\%$ of the variance of the outcome".

Propensity (binary D). For the propensity score, which is a probability, the best metric is the **log-loss** (also known as cross-entropy):

$$-\mathbb{E}_n[D\log\hat{p}(X) + (1-D)\log(1-\hat{p}(X))]\,.$$

Log-loss evaluates the quality of the predicted probabilities directly. It heavily penalizes predictions that are confident and wrong (e.g., predicting $\hat{p}(X) = 0.01$ when $D = 1$). Metrics like accuracy or AUC can be misleading and even dangerous, because they don't directly assess probability calibration, which is crucial for inverse propensity weighting. It is also good practice to explicitly post-process and calibrate the probabilistic predictions of your final propensity model (e.g. using sklearn's CalibratedClassifierCV).

Quick Check

When tuning a propensity model for a DR estimator, what is the safest default metric to use?

1. Accuracy
2. AUC
3. Log-loss (cross-entropy)
4. R^2

7.3 AutoML

Manually trying out many different models and hyperparameters can be tedious. **Automated Machine Learning (AutoML)** frameworks automate this process. You provide the data and a time budget, and the AutoML library will automatically search for the best model and hyperparameters for you. In this course, we use the `FLAML` library in the homework.

Discussion

A Critical Warning for AutoML in Causal Inference.
Many AutoML libraries were not designed with downstream causal inference tasks in mind. A common pitfall is using the wrong metric for propensity score estimation. For example, if you were to use 'metric='r2'' for a classification task in FLAML, the library would evaluate models based on their binary 0/1 predictions, not their predicted probabilities. This would produce a terribly calibrated model that is useless for inverse propensity weighting. It is absolutely critical to use a proper probabilistic metric like 'metric='log_loss'' when tuning propensity models.

FLAML (regression for g).

```
from flaml import AutoML
automl_g = AutoML(
    task="regression",
    metric="r2",
    eval_method="cv",
    n_splits=5,
    time_budget=60,
    early_stop=True,
    seed=123
)
automl_g.fit(X_train, y_train)
```

FLAML (classification for p).

```
automl_p = AutoML(
    task="classification",
    metric="log_loss",
    eval_method="cv",
    n_splits=5,
    time_budget=60,
    early_stop=True,
    seed=123
)
automl_p.fit(X_train, d_train)
phat = automl_p.predict_proba(X_test)[:, 1]
```

There is a plethora of well-established AutoML libraries that you can choose from:

- **FLAML** (*Fast Lightweight AutoML*): https://microsoft.github.io/FLAML/
- **Auto-sklearn**: https://automl.github.io/auto-sklearn/master/
- **H2O AutoML**: https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html
- **Hyperopt-sklearn**: https://hyperopt.github.io/hyperopt-sklearn/
- **AutoGluon**: https://auto.gluon.ai/stable/index.html
- **Azure AutoML**: https://learn.microsoft.com/azure/machine-learning/concept-automated-ml

8 Back to DR: cross-fitting and semi-cross-fitting

Now let's put everything together and see how to use these prediction tools within our doubly robust estimation framework.

8.1 Why nested tuning can be expensive

Recall that to avoid data reuse bias, we must use **cross-fitting** when constructing our DR estimator. A naive approach would be to run our full AutoML tuning procedure inside each fold of the cross-fitting. This is called **nested cross-validation**:

$$\text{outer folds for cross-fitting} \times \text{inner folds for tuning.}$$

While statistically valid, this can be extremely computationally expensive. It can also lead to instability, as we might select different models or hyperparameters in each outer fold.

8.2 Semi-cross-fitting (practical improvement)

A more practical and often better-performing approach is **semi-cross-fitting**. The idea is to separate the model selection step from the estimation step:

1. Use the full dataset once to choose a *recipe* (i.e., a model class and a set of hyperparameters) for g and p, for example using AutoML.
2. Freeze the recipe. Then, perform ordinary cross-fitting, where in each fold, you simply refit the chosen recipe on the training folds and make predictions on the held-out fold.

This is much faster and often more stable, since the model selection is done once on the full dataset. This approach is theoretically valid as long as the number of configurations (i.e., model class + hyperparameter value) you are choosing from, M, is not too large relative to the sample size. Specifically, the condition is that $\log(M)$ should not grow faster than the mean squared error rate of the final chosen estimator and should not be growing faster than $\sqrt{n}$, i.e. $\log(M)\left(\sqrt{n} + \text{RMSE}(\hat{g}) + \text{RMSE}(\hat{p})\right) \approx 0$. This is a very permissive condition in practice and allows for trying exponentially many configurations (in the number of samples).

Pseudo-code sketch.

```
from sklearn.base import clone

# 1) Tune once on full data
best_g = automl_regression_recipe(X, Y) # e.g., FLAML
best_p = automl_classification_recipe(X, D) # e.g., FLAML w/ log\_loss

# 2) Cross-fit with fixed recipes
for train_idx, test_idx in KFold(n_splits=5).split(X):
    # clone creates a new un-fitted instance of the estimator
    # with the same hyperparameter settings as the original instance
    g_model = clone(best_g)
    g_model.fit(np.c_[D[train_idx], X[train_idx]], Y[train_idx])
    p_model = clone(best_p)
    p_model.fit(X[train_idx], D[train_idx])

    g1hat[test_idx] = g_model.predict(np.c_[np.ones(len(test_idx)), X[test_idx]])
    g0hat[test_idx] = g_model.predict(np.c_[np.zeros(len(test_idx)), X[test_idx]])
    phat[test_idx] = p_model.predict_proba(X[test_idx])[:, 1]

theta_hat = dr_score(g1hat, g0hat, phat, Y, D)
```

8.3 Stacking: Choosing an Ensemble of Models

Instead of selecting a single best model, we can often get better performance by **stacking**, which means creating an ensemble of models. The simplest form of stacking is to learn a linear combination of the out-of-fold predictions from multiple candidate learners. This can improve performance when different learners capture different aspects of the data, but it must be done carefully to avoid overfitting the meta-learner.

In the context of the doubly robust estimator, stacking can also be performed in a semi-cross-fitting manner. In particular, we can train all the "recipes" (i.e. configurations of model classes and hyperparameters), in a cross-fitting manner. Then we can use the out-of-fold predictions all these models to learn the best weights to combine them, so as to best predict the target outcome. Thus, we are learning the weights of the ensemble of the M candidate models, using all the data.

Pseudo-code: Semi-Cross-Fitted Stacking in the DR Pipeline

```
for train, test in cv.split(X, y):
    for it, modely in enumerate(modely_list):
        mdl = clone(modely)
        mdl.fit(np.c_[D[train], X[train]], y[train])
        ghats[test, it] = mdl.predict(np.c_[D[test], X[test]])
        ghats1[test, it] = mdl.predict(np.c_[np.ones(len(test)), X[test]])
        ghats0[test, it] = mdl.predict(np.c_[np.zeros(len(test)), X[test]])
    for it, modelp in enumerate(modelp_list):
        mdl = clone(modelp)
        mdl.fit(X[train], D[train])
        phats[test, it] = mdl.predict_proba(X[test])[:, 1]
# calculate stacking weights for the outcome model
# and combine the outcome model predictions. We fit the weights using all the data.
stacker = LinearRegression().fit(ghats, y) # Regularized linear models can be used
ghats = stacker.predict(ghats)
ghats0 = stacker.predict(ghats0)
```

```
ghats1 = stacker.predict(ghats1)
# Similarly for propensity model; combine phats linearly
phats = LinearRegression().fit(phats, D).predict(phats)
```

However, the theory for semi-cross-fitting with stacking is more restrictive than for semi-cross-fitting; it requires that the number of models being combined, M (and not $\log(M)$), does not grow with the sample size (i.e., $M\,(\sqrt{n} + \text{RMSE}(\hat{g}) + \text{RMSE}(\hat{p})) \approx 0$). So you should only be using semi-crossfitting with stacking only when you choose an ensemble among a small number of "recipes". A practical tip is to first use cross-validation to find the best hyperparameters for each model class using all the data (e.g., find the best Random Forest, the best Gradient Boosted Forest, the best Lasso hyperparameters), and then use stacking to combine just these few best-in-class models. In this, case, if C is the number of model classes and M is maximum number of configurations that were examined for each model class, then we only need $C\log(M)\,(\sqrt{n} + \text{RMSE}(\hat{g}) + \text{RMSE}(\hat{p})) \approx 0$.

Pseudo-code: Semi-Cross-Fitted Stacking over Model Classes and Tuning within Class

```
# e.g., call FLAML for each type of estimator separately
# each call will just tune the hyperparams of that estimator type
# and return the best configuration for each model type
modely_list = [automl_regression_recipe(X, Y, estimator_list=[mdt])
               for mdt in ('rf', 'gb', 'lasso', 'enet', ...)]
# e.g., FLAML w/ log\_loss for each type of estimator separately
modelp_list = [automl_classification_recipe(X, D, estimator_list=[mdt])
              for mdt in ('rf', 'gb', 'lrl1', 'lrl2', ...)]

# then we do semi-cross-fitting with stacking
# with these best hyperparams for each estimator type
for train, test in cv.split(X, y):
    for it, modely in enumerate(modely_list):
        mdl = clone(modely)
        mdl.fit(np.c_[D[train], X[train]], y[train])
        ghats[test, it] = mdl.predict(np.c_[D[test], X[test]])
        ghats1[test, it] = mdl.predict(np.c_[np.ones(len(test)), X[test]])
        ghats0[test, it] = mdl.predict(np.c_[np.zeros(len(test)), X[test]])
    for it, modelp in enumerate(modelp_list):
        mdl = clone(modelp)
        mdl.fit(X[train], D[train])
        phats[test, it] = mdl.predict_proba(X[test])[:, 1]
# calculate stacking weights for the outcome model
# and combine the outcome model predictions. We fit the weights using all the data.
stacker = LinearRegression().fit(ghats, y) # Regularized linear models can be used
ghats = stacker.predict(ghats)
ghats0 = stacker.predict(ghats0)
ghats1 = stacker.predict(ghats1)
# Similarly for propensity model; combine phats linearly
phats = LinearRegression().fit(phats, D).predict(phats)
```

9 Summary: four takeaways

1. **Conditional expectations are best predictors.** The problem of estimating the outcome regression and propensity (more broadly called nuisance functions) for causal inference is equivalent to a standard supervised learning problem of finding the best predictor under squared loss.

2. **Generalization requires inductive bias.** To learn effectively in high dimensions, we need to go beyond simple ERM and introduce regularization or structural assumptions that guide the learning process.

3. **Modern ML is adaptive to effective complexity.** Methods like random forests, boosting, and neural networks are powerful because they can adapt to the underlying structure of the data, rather than being cursed by the ambient dimension.

4. **For DR: use CV for prediction, cross-fitting for nuisance reuse.** The standard, robust pipeline for causal estimation with ML involves using cross-validation to select good predictive models, and then using cross-fitting to plug those models into the DR estimator to avoid data reuse bias. Semi-cross-fitting is a practical way to make this process more efficient.