MS&E 228: Applied Causal Inference Powered by ML and AI

Lecture 5: Prediction with Modern ML for Downstream Causal Estimation

Vasilis Syrgkanis

Stanford University

Winter 2026

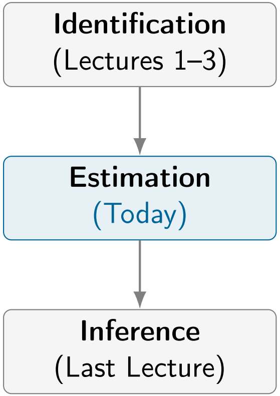
Readings: *Applied Causal Inference Powered by ML and AI*, §1, §3, §8.

Why a Lecture on Prediction?

- Last lecture: **doubly robust (DR)** ATE estimation + **cross-fitting**.
- DR requires estimating two **conditional expectation functions** with small **population RMSE**:

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

- Today: **how do we actually learn these well with modern ML** without strong parametric assumptions?



Four Takeaways for Today

1. **CEF = best predictor:** conditional expectations solve population risk minimization.
2. **Generalization needs structure:** without inductive bias, in-sample fit is meaningless (overfitting).
3. **Modern ML is adaptive:** trees/forests/boosting/NNs exploit *effective complexity* rather than raw dimension.
4. **Causal plug-in requires hygiene:** CV for prediction, cross-fitting for DR, and **semi-cross-fitting** as a practical improvement.

Part I: Conditional Expectations as Prediction

From causal nuisances to ML objectives

The Conditional Expectation Function (CEF)

We want to estimate a function of the form:

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

Key examples in causal inference:

$$\begin{aligned} g_0(d, x) &= \mathbb{E}[Y \mid D = d, X = x] && \text{(outcome regression)} \\ p_0(x) &= \mathbb{E}[D \mid X = x] && \text{(propensity / treatment model)} \end{aligned}$$

Unifying View

Both regression and classification are **conditional expectation problems**. Different targets Y vs. D mainly change the **loss** (square loss vs. log-loss), not the object.

Best Prediction Rule: Population Risk

For any candidate predictor f, define the mean squared prediction error:

$$\mathrm{MPE}(f) := \mathbb{E}\left[(Y - f(Z))^2\right].$$

Theorem (Best Prediction Rule)

$$f_0(Z) = \mathbb{E}[Y \mid Z] \in \arg\min_f \mathbb{E}[(Y - f(Z))^2].$$

Intuition: $f_0(Z)$ is the *best mean-square predictor* of Y given Z.

Proof Sketch: Error Decomposition

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

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

- First term does *not* depend on f.
- Cross-term is 0 because $\mathbb{E}[Y - f_0(Z) \mid Z] = 0$ (tower property).

Consequence

Minimizing MPE is equivalent to minimizing MSE $:= \mathbb{E}[(f_0(Z) - f(Z))^2]$.

Finite Samples: Empirical Risk Minimization (ERM)

With samples $(Y_i, Z_i)_{i=1}^n$ we typically solve:

$$\hat{f} \in \arg\min_{f \in \mathcal{F}} \mathbb{E}_n\left[(Y - f(Z))^2\right] \quad \text{(plus regularization / early stopping / bells and whistles).}$$

Key Question for Causal Inference

How do we ensure small **population** root-MSE (RMSE) $:= \sqrt{\text{MSE}}$?
DR ATE cares about *population* RMSE, not training error.

Big picture: generalization guarantees depend on the **complexity** of $\mathcal{F}$ and the **inductive bias** of the algorithm induced by the “bells and whistles” we add.

Part II: Linear Models

Low-dimensional vs high-dimensional regimes

Linear Models as Restricted Best Prediction Rules

Assume f is linear in engineered features $\phi(Z) \in \mathbb{R}^p$:

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

Then it suffices to minimize the MPE over linear predictors (aka **best linear predictor**):

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

Empirical analogue (OLS):

$$\hat{a} \in \arg\min_{a} \; \mathbb{E}_n[(Y - a^\top \phi(Z))^2].$$

Lens

OLS is *ERM over a linear function class.*

How Accurate is OLS? (Low Dimension)

Under benign regularity assumptions:

$$\mathrm{MSE}(\hat{a}) = \mathbb{E}\left[(\hat{a}^\top \phi(Z) - a_0^\top \phi(Z))^2\right] \approx \frac{p}{n} \quad \Rightarrow \quad \mathrm{RMSE} \approx \sqrt{\frac{p}{n}}.$$

- Parametric rate $n^{-1/2}$ when p is fixed and small.
- But if p grows with n, rates slow down.

Takeaway

Dimensionality matters. You cannot ignore p when reasoning about nuisance RMSE.

High-Dimensional Regime: Why Plain OLS Breaks

If $p \geq n$, we can often fit the training data *perfectly*:

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

- If feature vectors are not pathological, solution exists (for $p > n$ infinitely many).
- We get zero empirical loss $\mathbb{E}_n[(Y - \hat{a}^\top \phi(Z)]$ (not representative of population loss)
- This can **overfit noise**: training error ≈ 0 but population RMSE is large.
- Standard solvers return the **minimum ℓ_2-norm solution** ($\|a\|_2 = \sqrt{\sum_i a_i^2}$).
- True solution a_0 can be very far from minimum norm solution.

Causal Consequence

You should not use OLS to estimate $\hat{g}$ and $\hat{p}$ in the DR estimator, when $p > n$.

Poll

Question

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

1. Underfitting (too simple)
2. Overfitting (great training error, poor test error)
3. No solution exists

(Poll Everywhere)

Regularization: Add an Inductive Bias

We encode structure via a penalty $R(a)$.
Inductive bias: the true parameter a_0 has small $R(a)$.
Regularization Solution: Minimize empirical loss, while penalizing "large" solutions

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

- **Lasso:** $R(a) = \|a\|_1$ (sparsity; only $s << p$ entries of a_0 are non-zero).
- **Ridge:** $R(a) = \frac{1}{2}\|a\|_2^2$ (dense but small; all entries of a_0 are non-zero but small).
- **Elastic Net:** $\gamma\|a\|_1 + (1-\gamma)\frac{1}{2}\|a\|_2^2$: mix ℓ_1 and ℓ_2.

Interpretation

Regularization trades a bit of bias for a lot less variance (better generalization).

Rates in High-Dimensional Linear Models (Headlines)

Lasso (sparsity): if a_0 has $s \ll p$ nonzeros,

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

Ridge (bounded ℓ_2 norm): if $\|a_0\|_2 \leq B$,

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

Why This Matters for DR

Rates like $n^{-1/4}$ are exactly the "threshold" that often shows up in DR product rate theorem.

Misspecification: Approximation Error + Estimation Error

Even if f_0 is not linear in $\phi(Z)$, Regularized ERM $\hat{f}(Z) = \hat{a}^\top \phi(Z)$ converges to the **best-in-class** predictor with the same rates:

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

Decomposition:

$$\underbrace{\mathbb{E}[(f_0(Z) - \hat{f}(Z))^2]}_{\text{total MSE}} \leq \underbrace{\mathbb{E}[(f_0(Z) - f_\star(Z))^2]}_{\substack{\text{APX} \\ \text{Approximation error of} \\ \text{population best linear predictor}}} + \underbrace{\mathbb{E}[(f_\star(Z) - \hat{f}(Z))^2]}_{\substack{\text{EST} \\ \text{Convergence rate of} \\ \text{empirical best linear predictor} \\ \text{to population best linear predictor}}} .$$

Takeaway

To get small RMSE you need (i) a rich enough class (small APX) and (ii) regularization/structure (small EST).

Part III: Beyond Linear Models

Curse of dimensionality and adaptive ML

Curse of Dimensionality (Worst-Case Smooth Functions)

If we assume only that f_0 is β-smooth in p dimensions (has bounded and continuous β-higher order derivatives), the minimax rate is

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

- If $p = 10$ and $\beta = 1$:

$$n^{-1/12} \approx 0.1 \Rightarrow n \approx 10^{12} = 1 \text{ trillion}.$$

- If $\beta = 2$:

$$n^{-2/14} \approx 0.1 \Rightarrow n \approx 10^{7} = 10 \text{ million}.$$

Message

Pure smoothness is not enough in moderate dimensions.
We need *additional structure*.

How Modern ML “Bypasses” the Curse

Modern ML methods are **adaptive** to *effective complexity*, e.g.:

- sparsity / approximate sparsity (lasso)
- low-order interactions / partition structure (trees)
- compositional structure (deep nets)
- low intrinsic dimension (manifolds / representations / nearest neighbor)

Key Idea

A method is good when the true f_0 is “simple” in the method’s notion of complexity.

Decision Trees: Piecewise-Constant CEFs

A regression tree partitions feature space into leaves and predicts leaf averages:

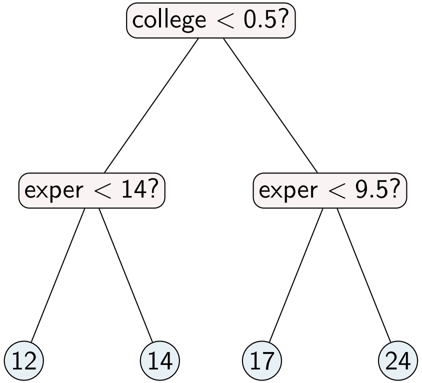
$$\hat{f}(z) = \sum_{\ell=1}^{L} \hat{\mu}_\ell \cdot \mathbb{I}\{z \in \text{leaf } \ell\}.$$

Strengths:

- captures interactions automatically
- handles mixed data types naturally

Weaknesses:

- high variance / instability
- discontinuous estimated functions



Some Theory (Effective Dimension for Trees)

- Suppose all variables are binary/categorical.
- Suppose the CEF depends only on $s \ll p$ relevant variables
- s is an effective dimension.

Theorem (Syrgkanis–Zampetakis '20 (informal))

Under regularity conditions, greedily grown regression trees with max depth $\gtrsim s$ and $\lesssim c\, s$ achieve

$$RMSE \approx \sqrt{\frac{2^s \log p \log n}{n}}.$$

Random Forests: Variance Reduction by Averaging

Random forests average many randomized trees:

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

Each tree is constructed on a dataset created by sampling n samples from the original dataset with replacement (bagging) or without replacement (subsampling).

- Bagging/subsampling reduces variance and improves stability.
- Great default for **tabular** data in many applied settings.

Regularization/Overfitting controls: tree depth, minimum samples in leaf.

Practical Tip

For propensities: evaluate with **log-loss** and check **calibration**.

Random Forests

Code (Python)

```python
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
rf = RandomForestRegressor(n_estimators=500, min_samples_leaf=5,
                           random_state=0)
rf.fit(Z_train, Y_train)
Yhat = rf.predict(Z_test)

rf = RandomForestClassifier(n_estimators=500, min_samples_leaf=5,
                            random_state=0)
rf.fit(Z_train, D_train)
Dhat = rf.predict_proba(Z_test)[:, 1]
```

Gradient Boosting: Sequentially Fit Residuals

Boosting builds an additive model:

$$\hat{f}_M(z) = \sum_{m=1}^{M} \nu \, h_m(z),$$

where each h_m predicts residuals from the previous stage, i.e. $R_m = Y - \sum_{\tau=1}^{m-1} \nu h_\tau(Z)$.

Why it shines:

- strong accuracy on tabular data (often beats RF)
- can capture complex interactions with controlled complexity
- unlike RF each tree we build is "cognizant" of what prior trees have already learned

Regularization/Overfitting controls: learning rate ν, tree depth, early stopping.

Gradient Boosted Forests: Theory Pointers

- High-dimensional theory is less developed than lasso/forests, but:
  - consistency/adaptivity results exist
  - early stopping is crucial
- Some pointers (selected):
  - Beygelzimer et al. (2015) *Online Gradient Boosting*
  - Zhang & Yu (2005) *Boosting with Early Stopping: Convergence and Consistency*
  - Wei, Yang & Wainwright (2017) *Early stopping for kernel boosting*

Neural Networks: Learned Representations

Think of a neural net as learning features:

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

- Strong when inputs are **unstructured** (text, images) or when data are very large.
- For medium-sized tabular data, GBDT / RF are often stronger "out of the box".

Regularization/Overfitting controls: weight decay, dropout, early stopping.

Connection to Causal ML

Representation learning can create low-dimensional "effective" features that help both g_0 and p_0.

Summary

- Trees: adaptive partitions (effective dimension via relevant variables).
- Forests: average randomized trees (smoother, lower variance).
- Boosting: sequentially explain residuals (powerful; early stopping is key).
- NNets: learn features $\phi(z; \alpha)$ from data (best with lots of data or unstructured data; representation learning).

Part IV: Choosing Models

Cross-validation, metrics, and AutoML

Which Method Should I Use? Cross-Validation (CV)

In-sample error can be optimistic due to overfitting. CV estimates **out-of-sample** prediction error.

***K*-fold CV:**

1. Split data into K folds.
2. For each fold k, fit on data excluding k and predict on fold k.
3. Pool out-of-fold predictions $\hat{Y}_i$ and compute empirical loss.

Use configuration, i.e. (method, hyperparameter), with smallest average out-of-fold empirical loss.

Theoretical Properties

Guarantees RMSE of best configuration + extra factor of $\approx \sqrt{\log(M)/n}$, where M is number of configurations evaluated.

Metrics: Regression vs Classification

Regression (Outcome Model $\hat{g}$)

$$\text{MSE} = \mathbb{E}_n[(Y - \hat{Y})^2],$$

For normalization, we compare to performance of a "constant model":

$$R^2 = 1 - \frac{\text{MSE}}{\widehat{\text{Var}}(Y)}.$$

If $R^2 < 0$, then you are not even beating always predicting the mean of Y. If $R^2 \approx 1$, outcome very predictable. If $R^2 = x \in (0, 1)$, your model explains $100x\%$ of the variance of Y.

Classification (Propensity $\hat{p}$)

Log-loss:

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

Targets accuracy of the probabilities. Upper bounds the MSE.
Don't use AUC (can ignore miscalibrated extreme probabilities) or any other metric that uses the binary predictions of the classifier.

Practical Tip

R^2 can also be used for classification. Make sure you calculate R^2 of the probabilistic and not the binary predictions.

Poll 3: Tuning Metric for Propensity Models

Question

When tuning a propensity model for DR (to be used in weights), what is the safest default metric?

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

(Poll Everywhere)

AutoML with FLAML (Homework Preview)

You can automate model selection + hyperparameter tuning under a time budget.

Listing 1: FLAML: regression (outcome model)

```python
from flaml import AutoML
from sklearn.base import clone

automl_g = AutoML(
    task="regression",
    metric="r2",
    eval_method="cv",
    n_splits=5,
    time_budget=60,
    early_stop=True,
    estimator_list=['lassolars', 'enet', 'rf', 'lgbm', 'xgboost'],
    seed=123)
automl_g.fit(X_train, y_train)
best_unfitted_model = clone(automl_g.model)
```

AutoML with FLAML: Classification for Propensity

Use `log_loss` for propensity tuning; report overlap diagnostics alongside DR estimates.
If you set `r2` unfortunately FLAML uses the binary predictions of the classifier.

Listing 2: FLAML: classification (propensity model)

```
automl_p = AutoML(
    task="classification",
    metric="log_loss",
    eval_method="cv",
    n_splits=5,
    time_budget=60,
    early_stop=True,
    estimator_list=['lrl1', 'lrl2', 'rf', 'lgbm', 'xgboost'],
    seed=123)
automl_p.fit(X_train, d_train)
p_hat = automl_p.predict_proba(X_test)[:, 1]
```

Many AutoML libraries to 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

Part V: Back to Doubly Robust Estimation

Cross-fitting and semi-cross-fitting

Where Prediction Enters DR

DR combines outcome regression and propensity in an “insensitive” score.

Reminder

DR behaves well if at least one nuisance is accurate *and* we avoid overfitting bias via cross-fitting.

Cross-fitting idea: split into folds; fit $\hat{g}^{(-k)}, \hat{p}^{(-k)}$ on training folds and evaluate DR score on held-out fold.

Nested CV Pain: Why DR Can Be Expensive

If we run AutoML *inside each cross-fitting fold*, we get nested validation:

$$\text{(outer) cross-fitting folds } \times \text{(inner) CV for tuning}$$

- computationally expensive (especially with many candidate models)
- can add randomness / instability fold-to-fold

Goal

Keep statistical validity but reduce compute and (often) improve power.

Semi-Cross-Fitting (Practical Improvement)

Semi-cross-fitting:

1. Use all data once to **select** a model class + hyperparameters (e.g. via FLAML).
2. Freeze the recipe; then do standard K-fold cross-fitting, but **refit only the chosen recipe** in each fold.

Benefits

- much faster (no nested CV per fold)
- tuning uses more data $\Rightarrow$ more stable
- often better finite-sample power

Caveat

Do not search an astronomically large model space. Roughly, keep $\log M$ modest relative to n and estimation rates ($\log(M) \cdot MSE(\hat{f}) \approx 0$ and $\log(M) \ll \sqrt{n}$)

Pseudo-code: Semi-Cross-Fitted DR Pipeline

```
# 1) Model selection (once, on full data)
best_g = flaml_fit(X, Y, time_budget=60, metric='r2')
best_g = clone(best_g.model)
best_p = flaml_fit_classification(X, D, time_budget=60, metric='log_loss')
best_p = clone(best_p.model)

# 2) Cross-fitting with fixed recipes
for train, test in cv.split(X):
    model_g = clone(best_g)
    model_g.fit(np.c_[D[train], X[train]], Y[train])
    model_p = clone(best_p)
    model_p.fit(X[train], D[train])

    g1hat[test] = model_g.predict(np.c_[np.ones(len(test)), X[test]])
    g0hat[test] = model_g.predict(np.c_[np.zeros(len(test)), X[test]])
    phat[test] = model_p.predict_proba(X[test])[:, 1]

# 3) Plug cross-fitted predictions into DR score formulas
theta_hat = dr(g1hat, g0hat, phat, Y, D)
```

Stacking: A More Fancy Model Selection (Ensembling Models)

Instead of picking one model, combine many via a meta-learner (stacking):

$$\hat{w} \in \arg\min_{w} \mathbb{E}_n \left[\left(Y - \sum_{m=1}^{M} w_m \hat{f}_m^{\text{oof}}(Z) \right)^2 \right].$$

f_m^{oof} is the model fitted out-of-fold (i.e. on the folds that don't contain the i-th sample). Then predict with $\sum_m \hat{w}_m \hat{f}_m(Z)$.

Takeaway

Stacking can outperform the best single model if different learners capture different structure.

Semi-Cross-Fitting with Stacking

Can be used in a semi-cross-fitting manner in the DR estimator.

1. Learn the weights $\hat{w}_m$ for each "recipe" using all the data.
2. During cross-fitting in the DR estimator, fit all the recipes out-of-fold
3. Combine them on the target fold linearly. Use all the data to learn the best linear combination $\hat{w}_m$.

Caveat

You have to be much more frugal on how many configurations M you search over. Roughly, keep M (not $\log(M)$) modest relative to n and estimation rates ($M \cdot MSE(\hat{f}) \approx 0$ and $M \ll \sqrt{n}$)

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

```python
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)
```

Wrap-up: Four Takeaways

1. Conditional expectations are **best predictors** (risk minimizers).
2. Generalization requires **inductive bias** (regularization / structure).
3. Modern ML methods are **adaptive** (effective complexity).
4. For DR: use **CV** $+$ **cross-fitting**; consider **semi-cross-fitting** to reduce compute.