Lecture 7: Core Machine Learning, part 2

LSE ME314: Introduction to Data Science and Machine Learning (https://github.com/me314-lse)

2025-07-24

Ryan Hübert

Slides last updated: 24 July 2025

Administrative update

Date/time: Friday, 1 August 2025 from 12:00 to 14:00.

Location: CKK rooms 1.04 to 1.07.

- ➜ CKK is the Cheng Kin Ku Building located at corner of Lincoln's Inn Field.

Please arrive early to find out which room you need to be in.

Performance on new data

Underfitting is a problem. . .

So far, we've been very concerned about improving performance, e.g.,

- ➜ Lower MSE on estimated regression models.
- ➜ High accuracy, precision and/or recall on classification models.

We're worried about our estimated model **underfitting** the data.

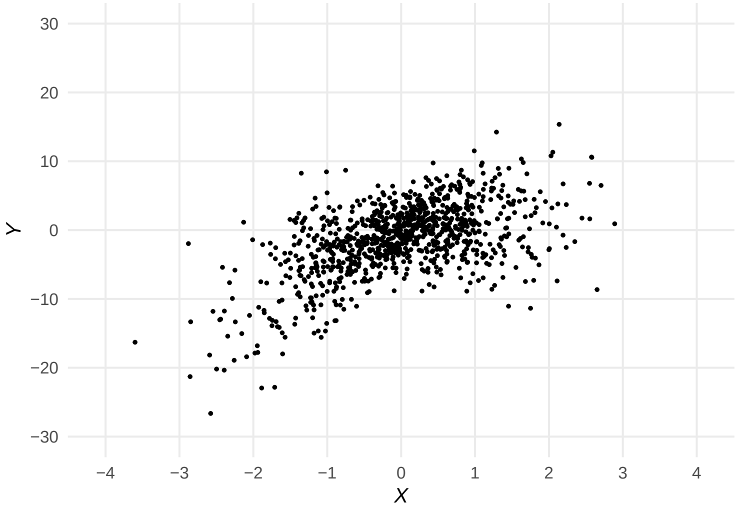
- ➜ This is just another term for generating predictions that are far from the actual data (e.g., high MSE).

Why are we concerned?

- ➜ Because we'd ideally like to use models trained by the machine on labeled data to predict labels in unlabeled data.
- ➜ In other words: we want to use the model on other data sets.

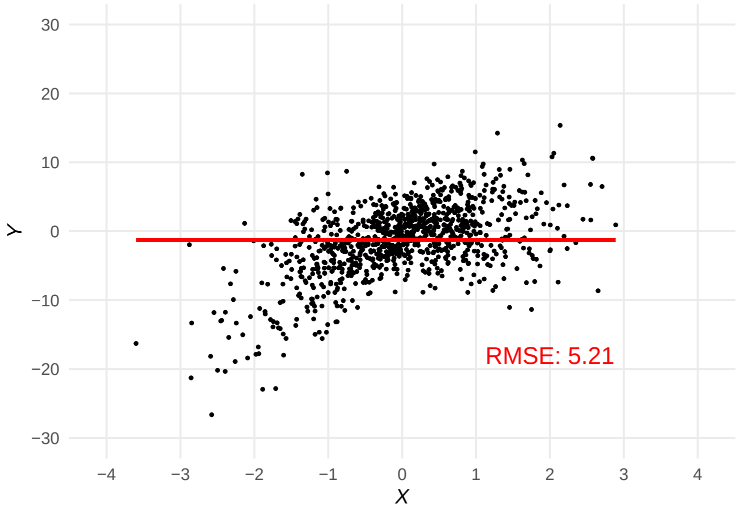
Underfitting is a problem. . .

Remember our labeled data from last lecture



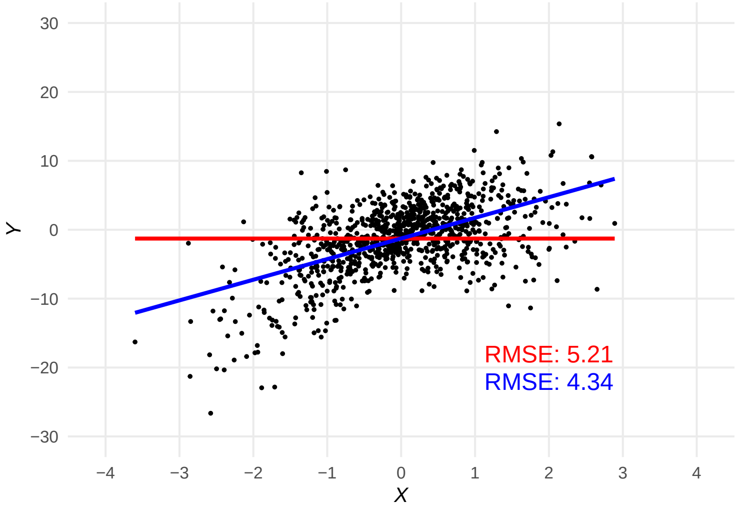
Underfitting is a problem…

An egregious example of underfitting: predict with mean of Y.



Underfitting is a problem...

An egregious example of underfitting: predict with mean of Y.



. . . and so is overfitting

Once we acknowledge we want to use our estimated model to do predictions on other datasets, there's another subtler problem.

Machines are very good at finding patterns in data, if allowed to.

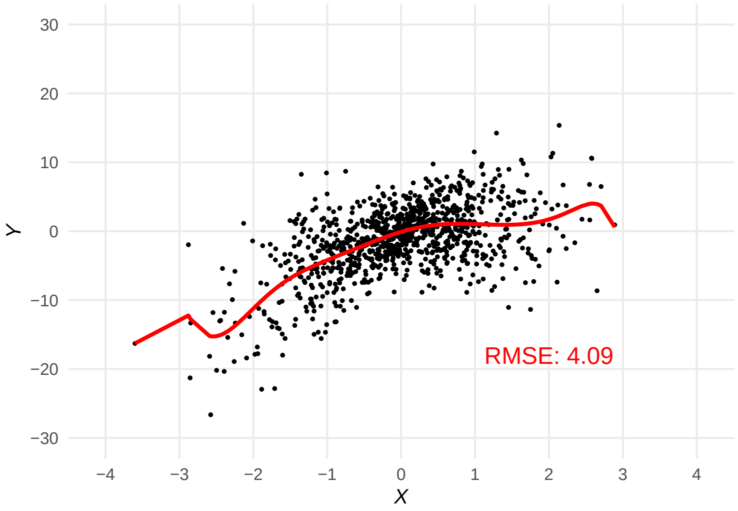
As a result, they tend to **overfit** the data they're given.

What do I mean by this?

- ➜ The machine builds a model that is too closely tailored for the sample it used to learn (the "training set").
- ➜ Overfitting means an estimated model won't perform well on new data.

... and so is overfitting

Overfitting example: using a 10th degree polynomial.



Overfitting

It's hard to see why this is a problem: our RMSE is lower!

But, it should be intuitive that this set of predictions won't do well on a different dataset.

The RSME we calculated is measuring **in-sample performance**.

➜ Specifically: in the *training* sample the machine used to learn.

Since we care about using the model to predict in *other* datasets, maybe in-sample RMSE is not all we care about?

In-sample RMSE does not tell us how the model will do on other data.

Test sets

How can we see whether an estimated model performs well on other datasets?

We test it on other datasets!

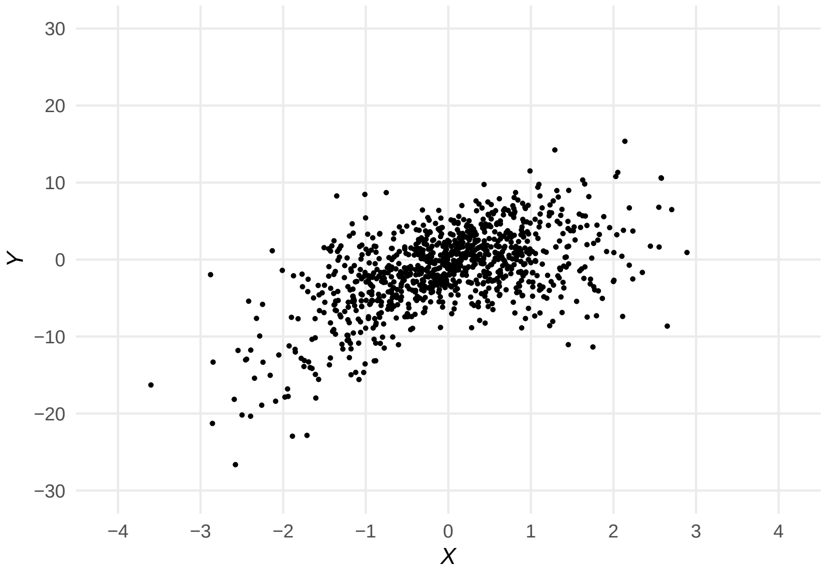
A **test set** is a subset of our labeled data that we "hold out."

- ➜ The machine is not allowed to look at it, or use it in any way.

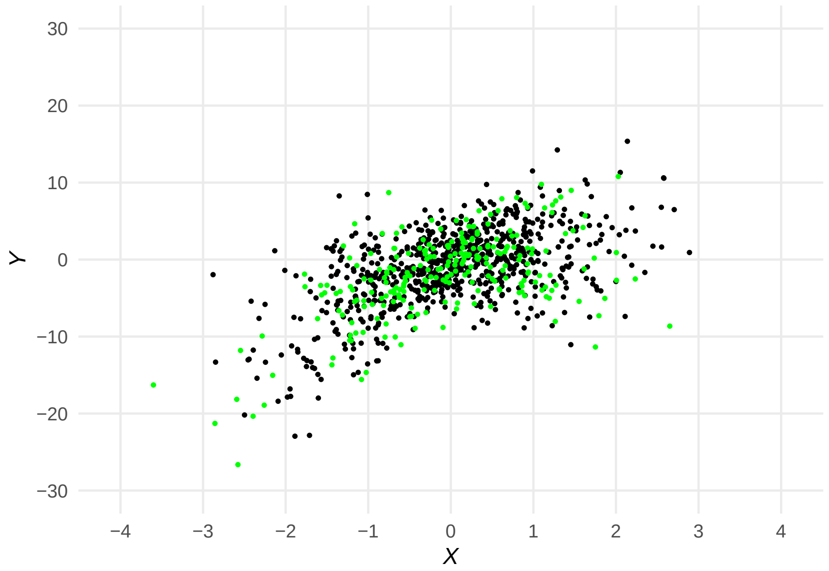
The machine builds its model using training data only.

- ➜ It probably will overfit to the training data (at least a little).
- ➜ But we can see how much by looking at how well model does on test data.

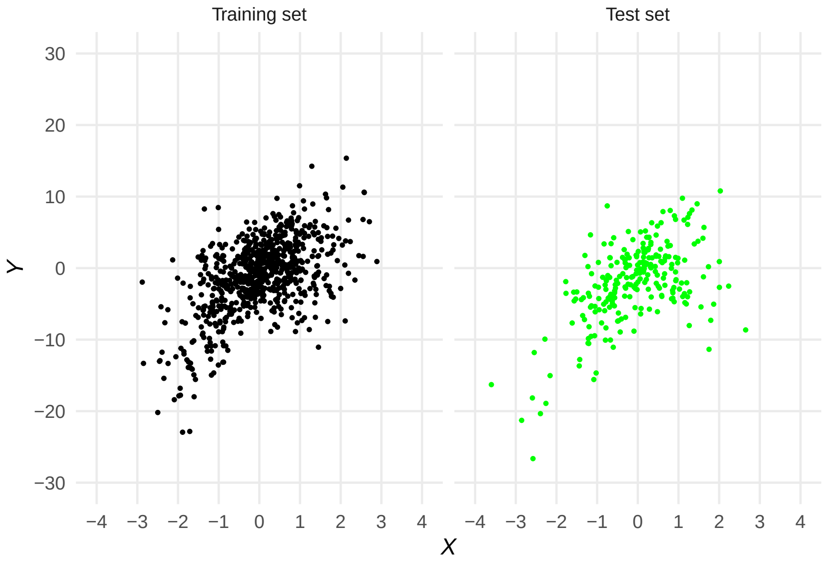
Test sets



Test sets



Test sets



Test sets

Let's calculate RMSE on the training and test sets.

```
# A tibble: 4 x 3
  Model     TrainingRMSE TestRMSE
  <chr>            <dbl>    <dbl>
1 mean              5.09     5.57
2 linear            4.23     4.64
3 quadratic         4.10     4.26
4 poly10            4.03    23.7
```

Performance is always *worse* in the test set than in the training set.

Even more interestingly, notice:

- In training set: always RMSE goes down as model becomes more complex.
- In test set: RMSE goes down then back up.

Key evidence that 10th degree polynomial is overfitting.

Bias-variance tradeoff

What we've seen so far is an example of the **bias-variance tradeoff**.

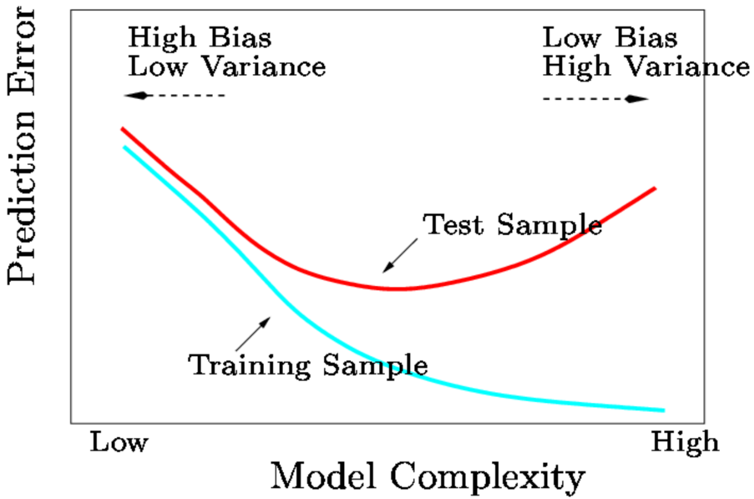
Some ML agorithms' predictions will have:

- *High bias*: Across datasets, the algorithm's predictions are consistently wrong in the same direction.
- *High variance*: Across datasets, the algorithm's predictions fluctuate wildly.

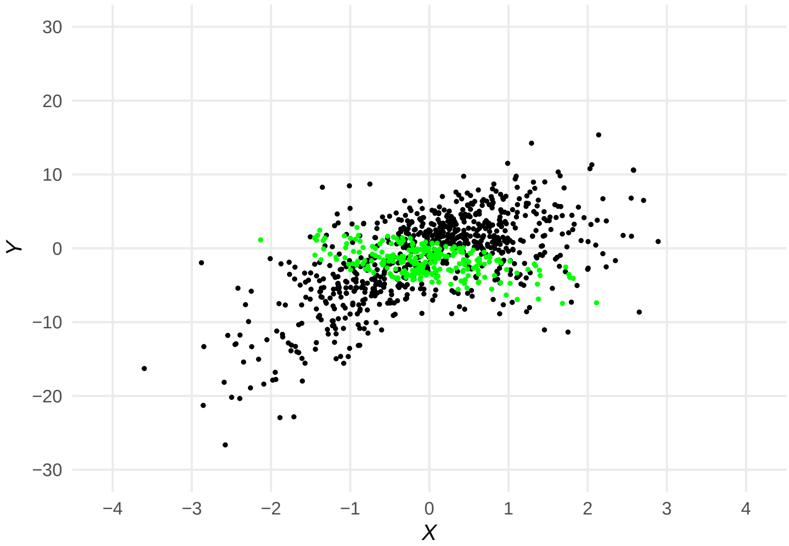
It turns out that algorithms with high bias have low variance and vice versa.

- Less complicated algorithms (the mean): high bias, low variance
- More complicated algorithms (complex polynomials): low bias, high variance

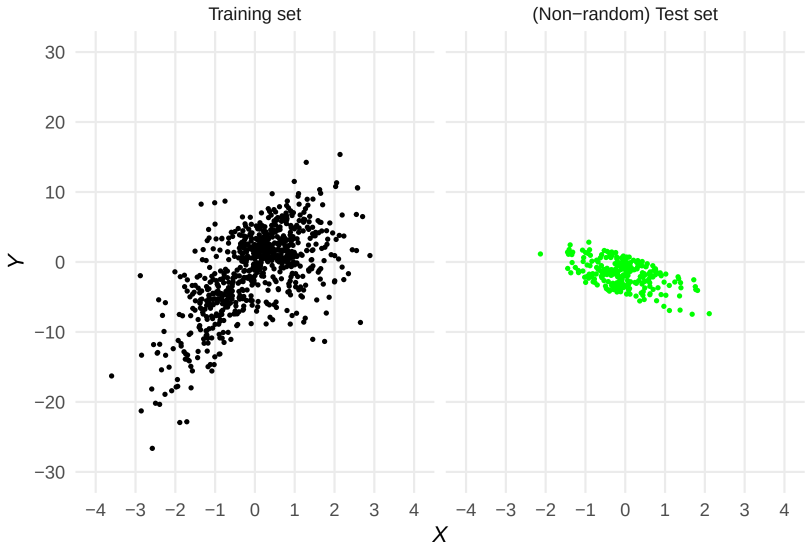
The bias-variance tradeoff



Importance of random sampling



Importance of random sampling



Classification with trees

Decision trees

A very intuitive way to classify: decision trees.

Assume you have a dataset with Y, X_1, ..., X_J.

Basic idea:

- Start with a predictor X_1.
- Split the dataset into two parts: high X_1 versus low X_1.
- Look to see how good that split separated the values of Y.
- Keep doing this until you get the "best" split based on X_1.
- Do this for all the other variables.
- Pick the variable with the best-best split and split data using that best-best split.
- On your new subsets of data, repeat the process.

This is (probably) how most humans think about classification!

Decision trees

Let's consider a simple example with a Y and X in a data set with six observations.

```
# A tibble: 6 x 2
      Y     X
  <dbl> <dbl>
1     1   3.1
2     1   2.7
3     0   1.3
4     0   1  
5     1   1.9
6     0   0.3
```

Decision trees

If you arrange by X, you can see a great split:

```
# A tibble: 6 x 2
      Y     X
  <dbl> <dbl>
1     0   0.3
2     0   1
3     0   1.3
4     1   1.9
5     1   2.7
6     1   3.1
```

A decision tree is perfectly predictive. Here's the estimated model:

$$\widehat{Y}_i = \begin{cases} 0 & \text{if } X < 1.6 \\ 1 & \text{if } X \geq 1.6 \end{cases}$$

Decision trees

Most datasets have much messier patterns in them.

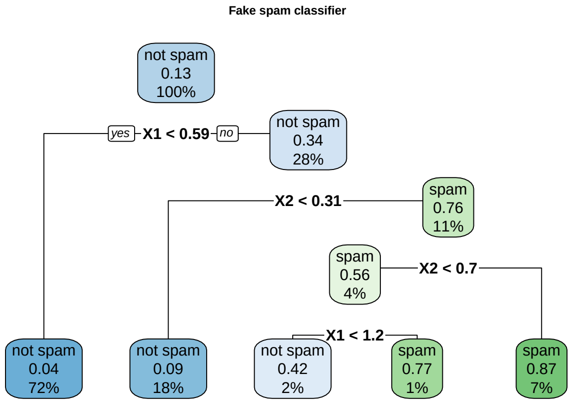
➜ This will give much messier models.

A preview of the data for a messier example:

```
         Y         X1          X2
1     spam  0.9086071  1.10651071
2 not spam  2.3791234 -0.44262748
3     spam  1.2611259  0.06157537
4     spam  1.0928224  0.48917626
5 not spam -0.5572497 -0.10269217
6 not spam -0.3847158  2.03572050
```

Decision trees

```r
library(rpart)
library(rpart.plot)
mod <- rpart(Y ~ X1 + X2, data = dft, method = "class")
rpart.plot(mod, main = "Fake spam classifier", cex = 1.5)
```



Anatomy of a decision trees

Some specialized terminology for trees:

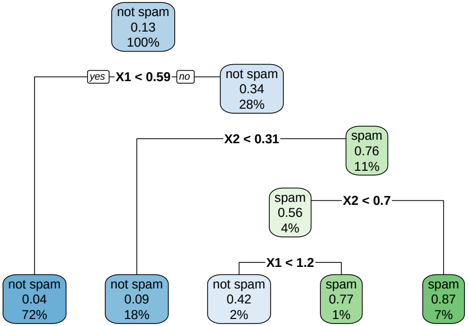
- **Nodes** are divided into two categories:
    - **Internal nodes**: places where splits/decisions occur.
    - **Terminal nodes** or **leaves**: the ends of a tree that tell you the predicted classifications.

A tree is a diagram of the estimated model:

- If you have some predictor values, you can find out the predicted class by navigating down the tree.

You can also write out the estimated model as a function.

Deriving a model from a decision tree



Note: as many rows as there are leaves...

$$\widehat{Y}_i = \begin{cases} \text{not spam} & \text{if } X_{1i} < 0.59 \\ \text{not spam} & \text{if } X_{1i} > 0.59 \text{ and } X_{2i} < 0.31 \\ \text{not spam} & \text{if } 0.59 < X_{1i} < 1.2 \text{ and } 0.31 < X_{2i} < 0.70 \\ \text{spam} & \text{if } X_{1i} > 1.2 \text{ and } 0.31 < X_{2i} < 0.70 \\ \text{spam} & \text{if } X_{1i} > 0.59 \text{ and } X_{2i} > 0.70 \end{cases}$$

The splitting process

How do the splits happen in practice?

A **greedy**, **top down** approach with **binary splits**:

➜ For all predictors, find an *optimal binary split*.

➜ Whichever predictor's optimal binary split is best, that's the first split in the tree.

Then, with each subset created from this split, repeat the process.

Two issues:

1. how to measure "optimal" splits?
2. when to stop splitting ("growing the tree")?

Measuring impurity

Split based on a measure of **impurity**:

- ➜ After a split, how mixed are response values within a new node?
- ➜ Goal: find splits that reduce impurity as much as possible.

For continuous response (regression trees)—use SSR.

For categorical response (classification trees)—use log-loss or Gini.

Regardless of measure of impurity, algorithm selects splits resulting in largest decrease in impurity.

Measuring impurity

Recall our first example:

```
# A tibble: 6 x 2
      Y     X
  <dbl> <dbl>
1     0   0.3
2     0   1
3     0   1.3
4     1   1.9
5     1   2.7
6     1   3.1
```

If you split at $X = 1.6$: no impurity in each resulting subset!

Growing and pruning decision trees

In principle, trees can keep growing (splitting) and get very deep.

Trees typically must stop growing if:

- ➜ All splits exhausted (every leaf is a single observation)
- ➜ All responses are identical in each leaf (no impurity left)

Trees can grow too big.

- ➜ Extreme example: when all splits are exhausted on large dataset.
- ➜ This model will be too complex, hard to use.
- ➜ It will also not perform well on new datasets (overfitting!!!)

Dealing with large trees

You could just stunt the growth of your tree, e.g.:

- ➜ Tell it to stop growing after certain # of splits.
- ➜ Tell it to stop growing when impurity is below a threshold.

Not ideal: what if important splits never get to happen?!

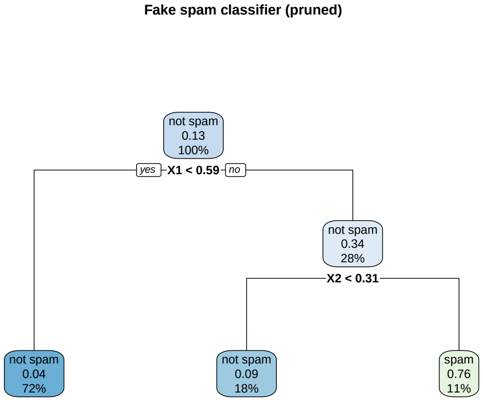
- ➜ Some important splits can happen deep in a tree.
- ➜ It is *not* the case that first split is most important.
- ➜ Why? Complex interactions in dataset.

Better: grow the largest tree possible then **prune** it.

- ➜ Snip off least important splits.
- ➜ Least important: high impurity $+ \alpha \times (\#$ terminal nodes$)$
- ➜ α is a hyperparameter – this is chosen by the analyst
- ➜ $\alpha = 0$ is no pruning

Dealing with large trees

```r
pruned_tree <- prune(mod, cp = 0.1) # larger numbers prune more
rpart.plot(pruned_tree, main = "Fake spam classifier (pruned)")
```



Pros and cons of trees

Decision trees are a very simple, intuitive way to classify.

➜ Don't require any advanced statistics or DGP assumptions.

They also tend to capture complex, non-linear relationships and interations between variables.

➜ Common in social science!

Major downside of decision trees: models are *too complex*

1. Small changes in data can produce completely different tree.
2. Trees often fit the training data too closely.
3. Not robust to outliers or slight shifts in distribution.

In brief: they tend to overfit!

Building a robust model

Choosing your model

Suppose you have a task in mind that's well suited for ML.

- Specifically: you want to train a model to label some unlabeled data.

Which approach should you take? You have to choose some things:

- Which algorithm? Such as regression, naive Bayes, decision tree, etc.
- What assumed DGP (e.g., for regression)?
- What **hyperparameters** do you want to set?

This process is called **model selection**.

It's tempting to try a lot of options and pick the best performer, but this is tricky. . .

Choosing your model

First, you already know that in-sample performance (MSE, precision, recall, etc), will be misleading.

It won't necessarily tell you how well a model will do on unseen data.

Suppose you:

- Create a test set for the purposes of evaluating performance of several approaches.
- Train 10 different algorithms on the training set.
- Evaluate performance of each on the test set.
- Choose the algorithm that has highest performance in test set.

This probably will create some (potentially serious) overfitting. Why?

Choosing your model

You already know: the machine tends to overfit to the training data.

- ➜ It does so by making choices based on how well its model performs in the training data.
- ➜ This is unavoidable.

But by evaluating several models on the test set, the human is now doing the same thing:

- ➜ She is making model selection decisions based on performance in the test set.
- ➜ And thus, tailoring the model to the test set.

Oops!

Choosing your model

Good grief: this seems impossible.

- On the one hand, we want the best performing model.
- On the other hand, we can't figure out which one is the best performing without potentially causing overfitting.

At some level, this is just a hard problem.

But there are some things we can do:

1. Validation
2. Choose approaches we *know* will do better without testing

Validation set

In addition to training and test sets, also create a validation set.

➜ The validation set is used to evaluate your model performance and choose a model.

A major advantage: it allows you to keep your test set untouched.

Seems like a small thing, but it is key:

The test set is only used one time to evaluate a model at the very end once the model has been chosen and estimated.

You cannot try to train another model after looking at test set performance.

At least now you will be able to see if you overfit!

Validation set

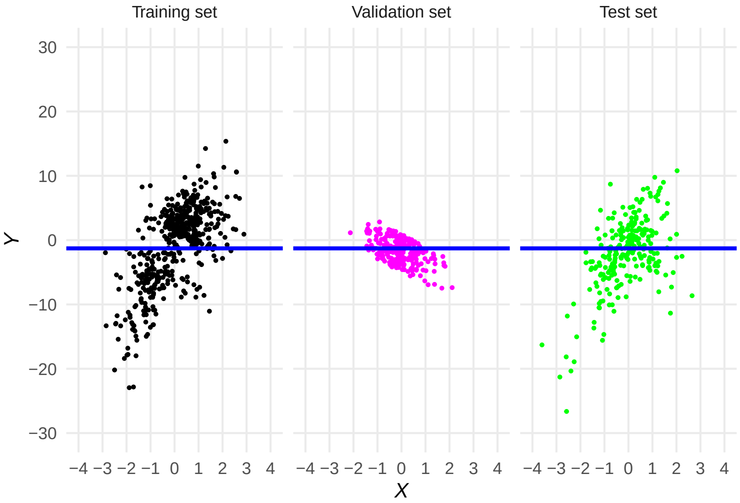
Doesn't solve all our problems, potentially creates new one:

1. Same principles apply: can still overfit to the *validation set*.
2. What if you (randomly) get an unusual validation set?
   - ➜ This will make overfitting even worse.
3. You're eating into your training set, reserving more of it for validation.
   - ➜ Machines aren't as good training with less data.
   - ➜ Limits the kinds of models you can train.
   - ➜ Specifically: can't fit useful complex models.

Pitfall: weird validation set

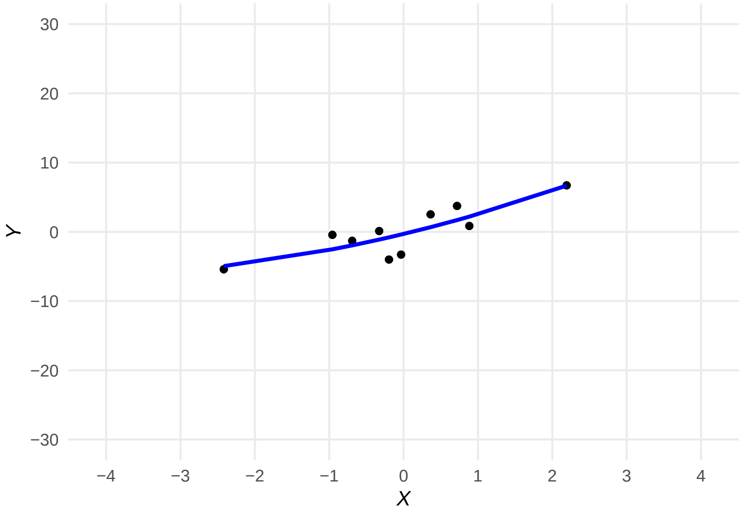
Suppose you create a “weird” validation set

➜ Remember: can happen by chance!



Pitfall: smaller training set

Extreme example: only 10 observations left in training set. . .
cannot fit more complex regressions, not enough data!



Goal: train and validate on all (non-test) data

A core issue with this validation approach: wasting data.

- ➜ You didn't use all your data to train, nor did you validate your model on all your data.

If you validate on all* your data:

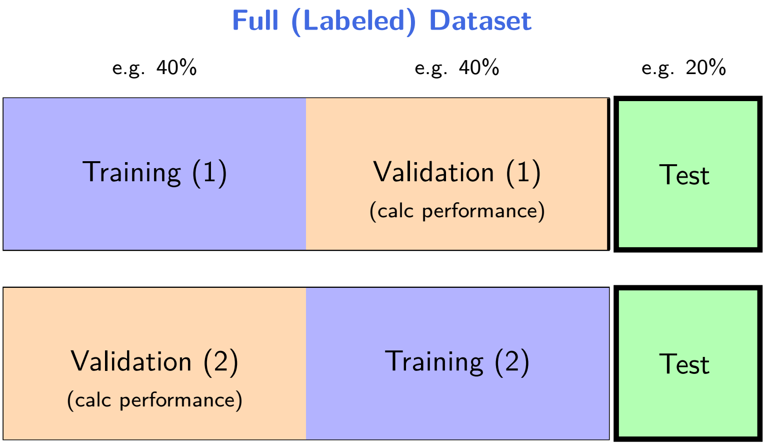
- ➜ You're guarding against drawing a "weird" validation set.

If you train with all* your data:

- ➜ You have a wider array of approaches and models you can try

{*} Note: always need to keep a test set held out and stored away

Cross-validation: the idea



Cross-validation: the idea

Model selection with **cross-validation**:

1. Split into training, validation and test
2. Pick an algorithm/model
3. Train on the training set and validate on the validation set then *switch* to train on the validation set and validate on the training set
4. Average performance metrics on validation sets
5. Repeat steps 2-4 for each algorithm/model you are considering
6. Choose the algorithm that has best average performance on validation sets
7. Quantify performance on the test set

Cross-validation

Now you've trained *and* validated the model on all the data.

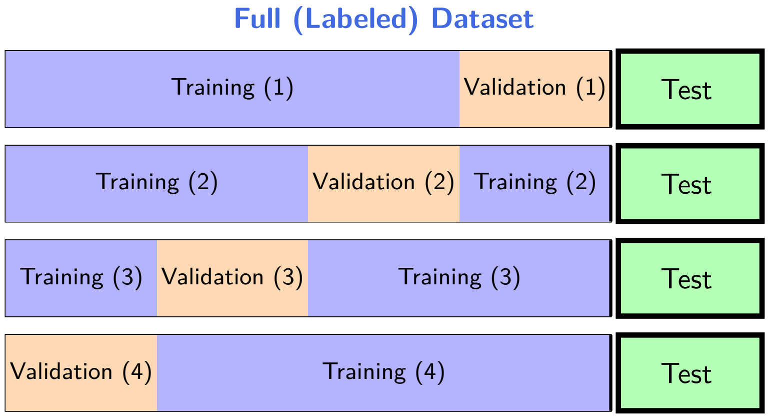
When you average out the performance across the two validation sets:

- ➜ "Unusually" high performance in one "unusual" set will be pulled down.
- ➜ "Unusually" low performance in one "unusual" set will be pulled up.

This helps you guard against overfitting during model selection.

We can do *even better*: why only split the data once?

Cross-validation: the idea



Cross-validation

Notice: each round of training uses a bigger dataset.

- This helps the machine out.
- It is easier for it to learn with more data.

I just showed you **four fold cross-validation** where you partition your data into 4 randomly created **folds**.

- For each fold k, you treat all the folds *except* k as a training set, fitting a model on it and evaluating performance on the held-out fold k.
- Repeat for each of the remaining folds.
- Average together the performance metrics across the folds.

More folds: (1) computation costs increase, but (2) each fold's training set is larger and more representative of full dataset.

Cross-validation

Some bad news...

You can still create overfitting if you use cross-validation, then tune your model and use cross-validation again (esp. with same folds).

The principle remains unchanged: you are at risk of overfitting when you tune your model in response to measured performance.

- E.g., iteratively training on same training set in response to performance statistics from previous rounds of training.

But: cross-validation is an aggressive attempt to reduce overfitting without causing underfitting:

- Use larger training sets in each round (reduce underfitting)
- Average performance across each fold's validation set (guard against overfitting)

Techniques for performance improvements

Tuning

Obviously you want to pick good models. So far that has meant:

- Choosing a good algorithm (regression, NB, etc.)
- Choosing a DGP you give the machine (e.g. regression equation)

But even after you commit to a general approach, you likely will need ti need to **tune** your models.

Term is a bit ambiguous, but common agreement: it includes choosing **hyperparameters**.

- Hyperparameters control overall behavior of your algorithms.
- E.g., instructions you supply about how to prune a tree (α)

If you choose badly: underfitting or overfitting.

Tuning

There are some common strategies, which we won't explore in detail:

- ➜ **Grid search**: systematic, exhaustive
- ➜ **Random search**: faster, often good enough
- ➜ **Bayesian search**: learns where to look next

Typically: start broad and/or coarse and zoom in.

Don't waste time tuning a model that's too simple or complex.

Cross-Validation to tune a hyperparameter

1. **Choose the hyperparameter**: e.g., α for decision trees
2. **Define a grid of candidate values**: such as $\alpha \in \{0.01, 0.1, 0.2, 0.5, 1\}$
3. **Set up k-fold cross-validation**: split training data into k folds
4. **Evaluate each candidate** value of the hyperparameter:
   - ➜ Train on $k-1$ folds
   - ➜ Validate on the held-out fold
   - ➜ Repeat for all k folds
   - ➜ Average the performance across folds
5. **Select best hyperparameter**: choose value with best average validation performance
6. **Re-train the final model**: use the full training set and the selected hyperparameter

Choosing from models with good properties

One way to try to guard against overfitting when choosing your models is to only choose from models you know to be pretty good.

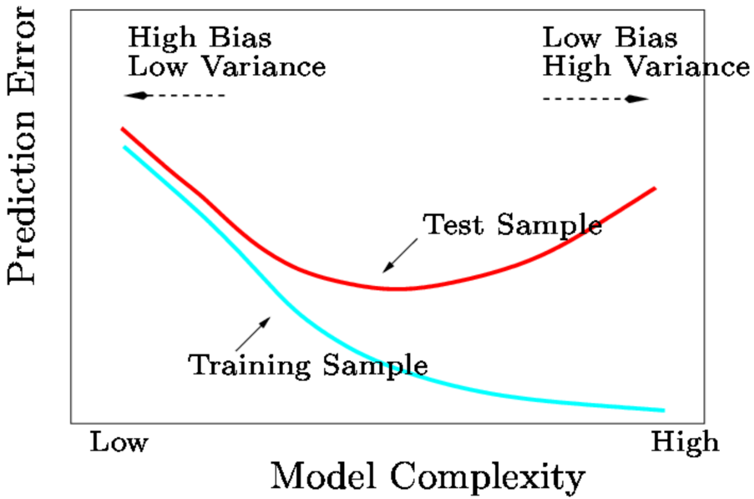
An obvious example: plot your data and choose DGPs for regressions that visually look right.

Important:

- ➜ We spent a lot of time worrying about misleading performance in sample.
- ➜ But, performance in the training sample is very useful for initial decisions about choosing which algorithms and models to run.

Recall the schema for the bias-variance tradeoff…

Choosing from models with good properties



Choosing from models with good properties

So: yes, you have to use your judgment (and your gut) to pick some algorithms and approaches you want to use.

But there are some other more sophisticated things you can do to improve performance before looking at your data.

We'll cover two here:

1. **Regularization**: literally make your model "more regular" by penalizing excessive complexity.
2. **Ensemble methods**: blend together multiple models to get the best of all of them.

Regularized regression

Recall that model complexity can create overfitting.

In regression: complexity can arise from too many predictors.

We can add a **penalty for model complexity** (e.g., too many predictors!), such that we now minimize:

$$\mathrm{SSR} + \lambda \sum_{j=1}^{J} \beta_j^2 \qquad \textbf{ridge regression}$$

or

$$\mathrm{SSR} + \lambda \sum_{j=1}^{J} |\beta_j| \qquad \textbf{lasso regression}$$

λ is the **penalty parameter** (a hyperparameter chosen with CV!)

➜ The lasso and ridge penalties are often referred to as L1 and L2

Lasso vs. ridge regression

How does the penalty work?

- Algorithm is trying to *minimize* SSR but λ is increasing it.
- The optimal way to minimize SSR, given the penalty: push down coefficients that create large penalties.
- If $\lambda = 0$, both are equivalent to standard OLS.

Both types of regression penalize large regression coefficients.

- The idea: we're worried about some features causing our model predictions to swing wildly (too much variance).
- Features with huge effects are pushed toward zero.

Lasso regression pushes some coefficients to zero; ridge does not.

- Lasso can be used for feature selection and to identify a regression when $J > n$.

Advantages and disadvantages of regularized regression

Advantages of regularized regression:

- ➜ Easy to interpret
- ➜ Works well when features are each independently informative
- ➜ Works well when data is limited

Disadvantages:

- ➜ Does not work well when features are, e.g., jointly informative
- ➜ Often outperformed by more complex methods

Ensembles (I): random forests

Key problem: want to make sensitive trees more robust.

Random forests improve on decision trees in two main ways:

1. **Bagging** (short for “bootstrap aggregating”)
   - ➜ Train many decision trees on different bootstrap samples.
   - ➜ Each tree sees a slightly different subset.
   - ➜ Final prediction calculated across all trees. (average for regression, majority vote for classification)

Reduces variance through averaging.

2. Random feature selection
   - ➜ At each split in each tree, only a (small) random subset of predictors is used.
   - ➜ Forces trees to explore different parts of the predictor space.

Reduces correlation between trees, improving power of averaging.

Ensembles (I): random forests

Under the hood:

1. Each tree votes for a class label for each observation.
2. The forest counts votes, estimates the proportion of votes for each class.
3. The predicted class is the argmax of those proportions.

This is an ensemble method because the algorithm is combining results from a large number of trees.

- ➜ Creating a *forest* of trees
- ➜ Using “wisdom of the crowd” (or rather, the forest) to make better predictions.

What is the “model” of a estimated RF?

This is different than what we’ve seen before.

A few important distinctions:

1. Since this algorithm uses trees, there’s no assumed DGP.
2. Since this averages over randomly generated trees, there’s no overall tree to look at.
3. Since there is no overall tree, it’s difficult (nearly impossible) to write out a model.
4. The algorithm initially produces predicted classifications, you have to approximate predicted probabilities if you want them.

➜ For binary response, typically: % of random trees that classify as 1.

Ensembles (II): stacking

Another common ensemble approach is called **stacking**.

One example, called **Super Learner**:

1. Train multiple base models (e.g., logistic regression, random forest, naive Bayes, etc.).
2. For each observation, collect predicted probabilities (or predictions) from each base model.
3. Create a new dataset:
   - ➜ Predictors = model predictions
   - ➜ Respomse = true labels
4. Fit a “meta-model” (often a simple linear regression or logistic regression) using this new dataset.
5. The meta-model’s output is the final prediction.

Ensembles (II): stacking

Other kinds of ensembles (e.g., RF) assume that all the base models are similar

- ➜ Stacking does not – diversity is encouraged.
- ➜ Wider range of models allows for more opportunity to improve performance

The meta-model learns to trust some models more than others

- ➜ E.g., if the meta-model is a linear regression, $\hat{\beta}$s are the weight the meta-model puts on each base model.

- ➜ Stacking is especially useful when base models are complementary such that they make different types of errors.