Model selection

Fraida Fund

Contents

Math prerequisites for this lecture: None.

A supervised machine learning "recipe"

- *Step 1*: Get labeled data: $(\mathbf{x_i}, y_i), i = 1, 2, \cdots, N$.
- *Step 2*: Choose a candidate **model** f: $\hat{y} = f(x)$.
- *Step 3*: Select a **loss function**.
- *Step 4*: Find the model **parameter** values that minimize the loss function (**training**).
- *Step 5*: Use trained model to **predict** $\hat{y}$ for new samples not used in training (**inference**).
- *Step 6*: Evaluate how well your model **generalizes**.

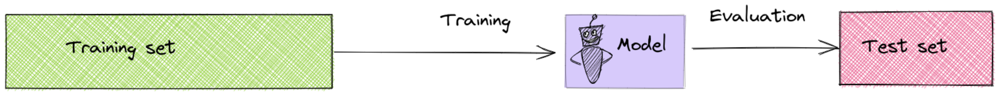

Figure 1: When we have only one model to consider, with no "hyperparameters".

Model selection problems

Model selection problem: how to select the $f()$ that maps features X to target y?

We'll look at a few examples of model selection problems (polynomial order selection, selecting number of knots and degrees in spline features, selecting number of features), but there are many more.

Problem 1: Polynomial order selection problem

- Given data $(x_i, y_i), i = 1 \cdots, N$ (one feature)
- Polynomial model: $\hat{y} = w_0 + w_1 x + \cdots + w_d x^d$
- d is degree of polynomial, called **model order**
- **Model order selection problem**: choosing d

Using training loss for polynomial order selection?

Suppose we would "search" over each possible d:

- Fit model of order d on training data, get $\mathbf{w}$
- Compute predictions on training data
- Compute loss function on training data: $MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
- Select d that minimizes loss on training set

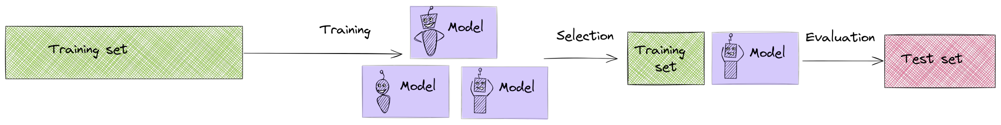

Figure 2: This approach does *not* work, because the loss function always decreases with d (model will overfit to data, training error decreases with model complexity!)

Note that we shouldn't use the test data to select a model either - the test set must be left as an "unused" data set on which to evaluate how well the model generalizes.

Recap: Spline features (1)

Polynomial models of high d are actually bad, usually -

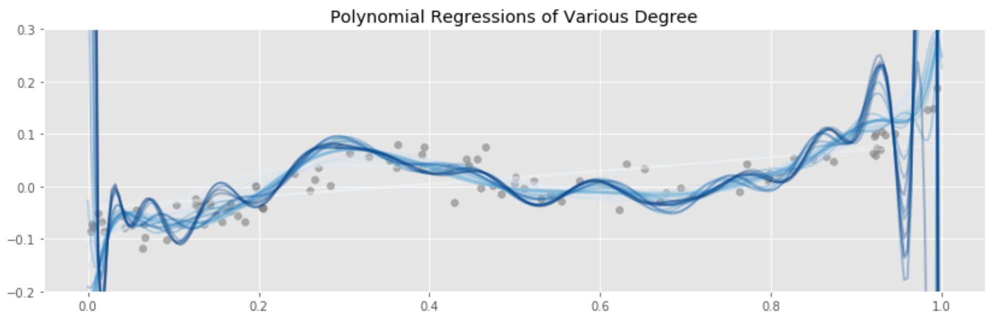

Figure 3: Polynomial model - note the boundary behavior. (Image source)

- tends to get kind of weird at the boundaries of the data (Runge's phenomenon)
- really bad if you need to extrapolate past the range of the training data
- acts *globally* when different regions of the data might have different behavior

Recap: Spline features (2)

Instead, we tend to prefer lower-d piecewise functions, so we can fit *local* behavior:

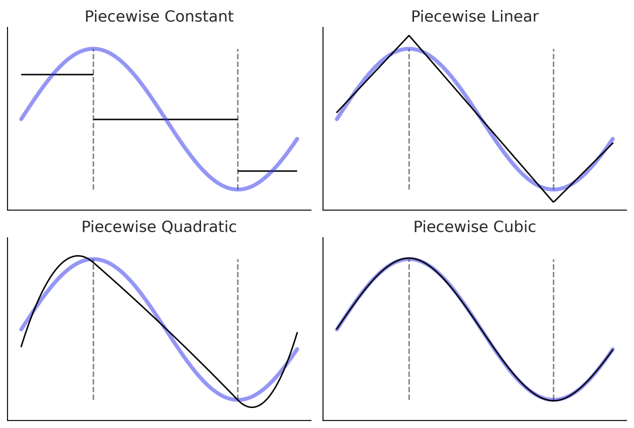

Figure 4: The blue line is a true function we are trying to approximate. The black lines are piecewise polynomials of increasing order. (Image source)

The feature axis is divided into breakpoints - we call each one a "knot" - and then we define basis functions that are equal to a polynomial function of the feature between two knots.

If we constrain the piecewise function to meet at the knots, we call these splines - basis splines or "B splines".

Recap: Spline features (3)

For constant functions (degree 0) - given "knots" at positions k_t, k_{t+1}:

$$\phi_{t,0}(x_{i,j}) = \begin{cases} 1, & k_t \leq x < k_{t+1} \\ 0, & \text{otherwise} \end{cases}$$

Recap: Spline features (4)

For degree $p > 0$, defined recursively:

$$\phi_{t,p}(x) := \frac{x - k_t}{k_{t+p} - k_t}\phi_{t,p-1}(x) + \frac{k_{t+p+1} - x}{k_{t+p+1} - k_{t+1}}\phi_{t+1,p-1}(x)$$

You won't need to compute this yourself - use `SplineTransformer` in `sklearn`.

Problem 2: Selecting number of knots, degree for spline features

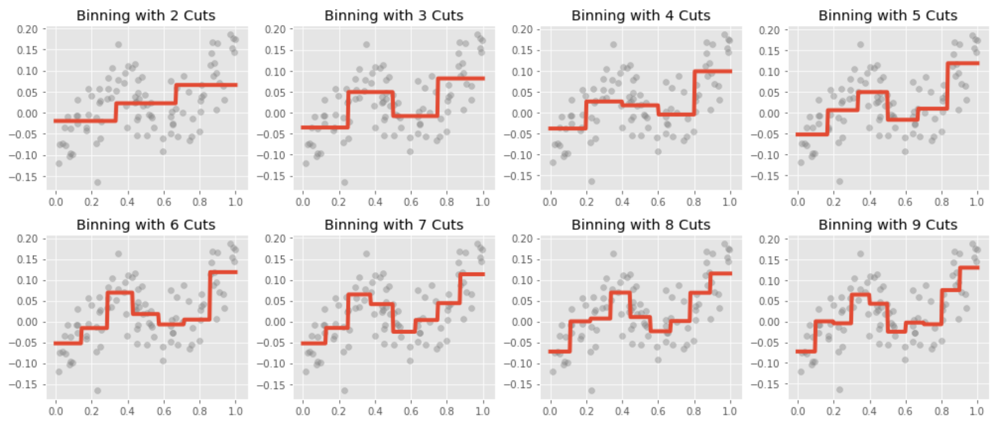

Figure 5: Increasing number of knots makes the model more complex. (Image source.)

Now we have two "knobs" for tuning model complexity:

- the degree of the splines,
- and the number of knots!

The number of features will be: number of knots + degree - 1

We have the same problem as before:

- if we select the number of knots, degree to minimize error on the training set - we will always pick the model with the most knots, highest degree
- if we choose the model that minimizes error on the test set, we don't have a held-out set on which to evaluate our model on new, unseen data. (If we use the test data for model selection, then again for evaluation, we will have an overly optimistic evaluation of model performance on unseen data.)

Problem 3: Feature selection (1)

Given high dimensional data $\mathbf{X} \in R^{n \times d}$ and target variable y,

Linear model: $\hat{y} = w_0 + \sum_{j=1}^{d} w_j x_j$

- Many features, only some are relevant (you don't know which, or how many!)
- **Feature selection problem**: fit a model with a small number of features

Problem 3: Feature selection (2)

Select a subset of $k << d$ features, $\mathbf{X}_S \in R^{n \times k}$ that is most relevant to target y.

Linear model: $\hat{y} = w_0 + \sum_{x \in \mathbf{X}_S} w_j x_j$

Why use a subset of features?

- High risk of overfitting if you use all features!
- For linear regression, when $N \geq p$, variance increases linearly with number of parameters, inversely with number of samples. (Refer to extra notes posted after class at home.)

Today we consider the challenge of selecting the number of features k, in a future lesson we will discuss how to decide $which$ features to include.

Once again:

- if we select the number of features to minimize error on the training set - we will always pick the model with the most features. Our model will happily overfit to the "noise" of irrelevant features!
- and we also shouldn't use our test set.

Validation

We will discuss a few types of validation:

- Hold-out validation
- K-fold cross validation
- Leave-p-out validation

Hold-out validation (1)

- Divide data into training, validation, test sets
- For each candidate model, learn model parameters on training set
- Measure error for all models on validation set
- Select model that minimizes error on validation set
- Evaluate *that* model on test set

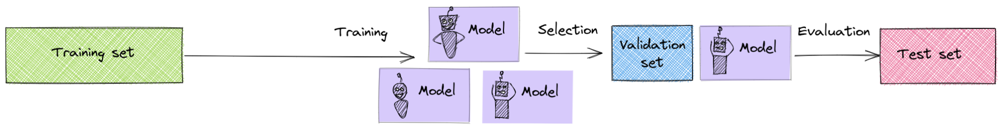

Figure 6: Model selection with a validation set.

Note: sometimes you'll hear "validation set" and "test set" used according to the reverse meanings.

Hold-out validation (2)

- Split X, y into training, validation, and test.
- Loop over models of increasing complexity: For $p = 1, \ldots, p_{max}$,
  - **Fit**: $\hat{w}_p = \text{fit}_p(X_{tr}, y_{tr})$
  - **Predict**: $\hat{y}_{v,p} = \text{pred}(X_v, \hat{w}_p)$
  - **Score**: $S_p = \text{score}(y_v, \hat{y}_{v,p})$

Hold-out validation (3)

- Select model order with best score (here, assuming "lower is better"):

$$p^* = \underset{p}{\text{argmin}}\, S_p$$

- Evaluate:

$$S_{p^*} = \text{score}(y_{ts}, \hat{y}_{ts,p^*}), \quad \hat{y}_{ts,p^*} = \text{pred}(X_{ts}, \hat{w}_{p^*})$$

Problems with hold-out validation

- Fitted model (and test error!) varies a lot depending on samples selected for training and validation.
- Fewer samples available for estimating parameters.
- Especially bad for problems with small number of samples.

K-fold cross validation

Alternative to single split:

- Divide data into K equal-sized parts (typically 5, 10)
- For each of the "splits": evaluate model using $K-1$ parts for training, last part for validation
- Average the K validation scores and choose based on average

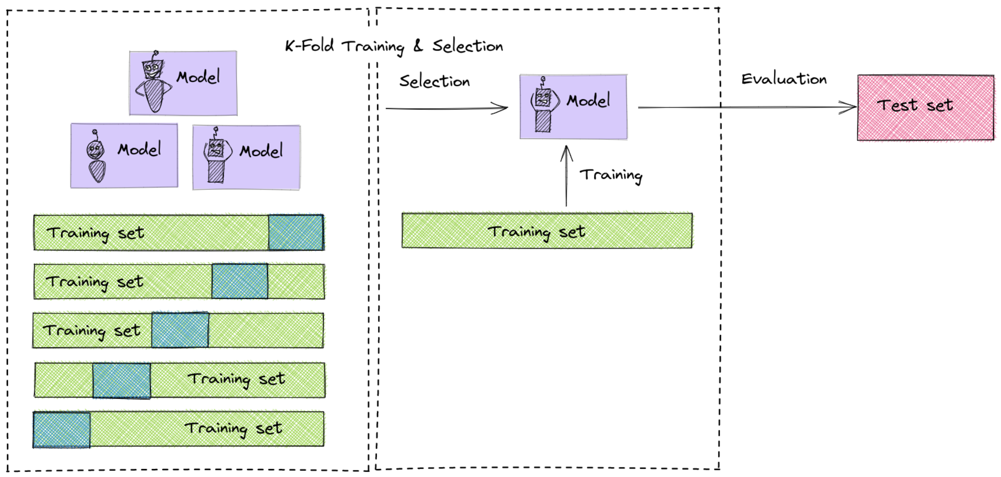

Figure 7: K-fold CV for model selection.

K-fold CV - algorithm (1)

Outer loop over folds: for $i = 1$ to K

- Get training and validation sets for fold i:
- **Inner loop** over models of increasing complexity: For $p = 1$ to p_{max},
  - **Fit**: $\hat{w}_{p,i} = \text{fit}_p(X_{tr_i}, y_{tr_i})$
  - **Predict**: $\hat{y}_{v_i,p} = \text{pred}(X_{v_i}, \hat{w}_{p,i})$
  - **Score**: $S_{p,i} = \text{score}(y_{v_i}, \hat{y}_{v_i,p})$

K-fold CV - algorithm (2)

- Find average score (across K scores) for each model: $\bar{S}_p$
- Select model with best *average* score: $p^* = \text{argmin}_p \bar{S}_p$
- Re-train model on entire training set: $\hat{w}_{p^*} = \text{fit}_p(X_{tr}, y_{tr})$
- Evaluate new fitted model on test set

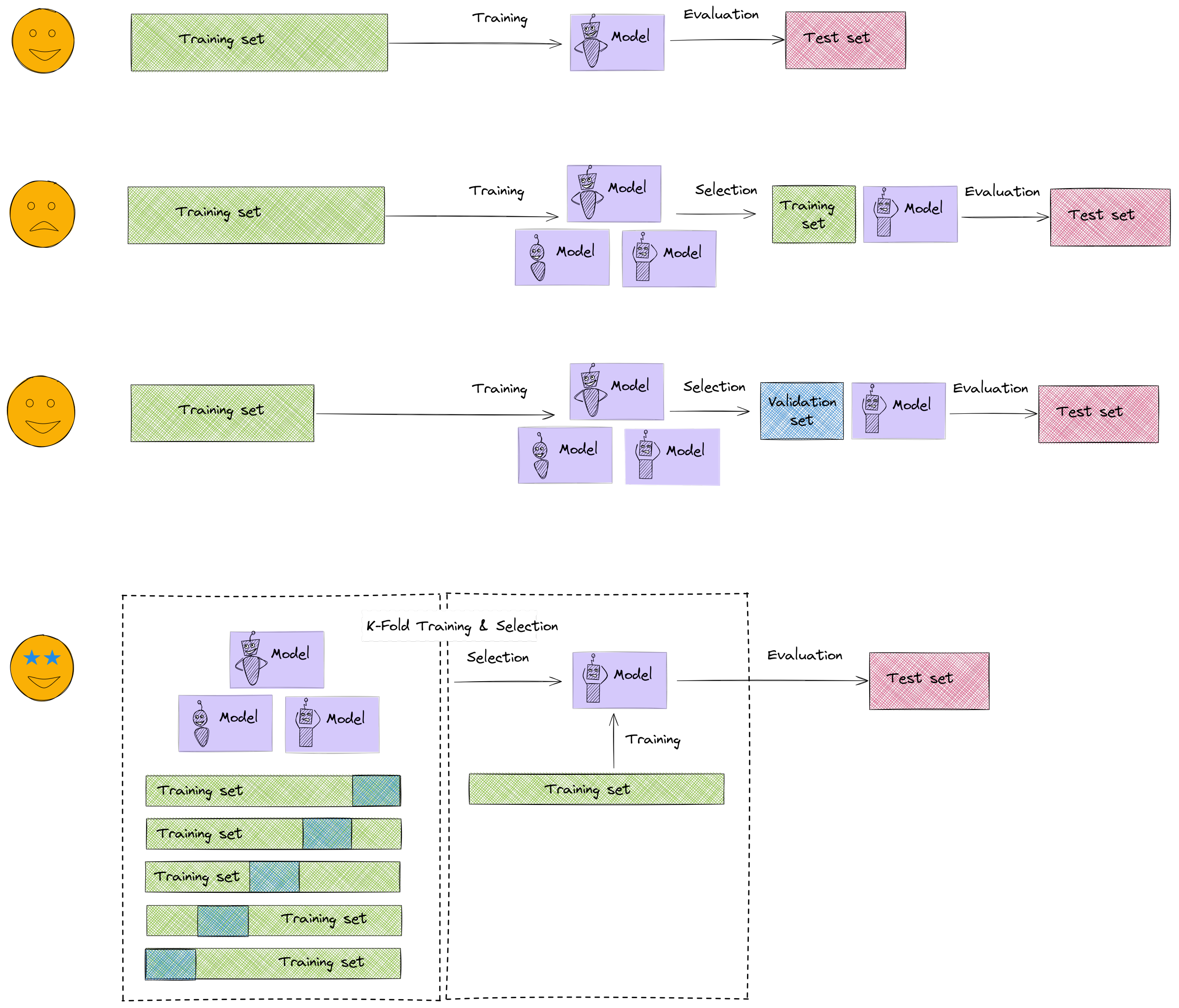

Figure 8: Summary of approaches. Source.

Leave-p-out CV

- In each iteration, p validation points
- Remaining $n-p$ points are for training
- Repeat for *all* possible sets of p validation points

This is *not* like K-fold CV which uses non-overlapping validation sets (they are only the same for $p=1$)!

Computation (leave-p-out CV)

$\binom{n}{p}$ iterations, in each:

- train on $n-p$ samples
- score on p samples

Usually, this is too expensive - but sometimes LOO CV can be a good match to the model (KNN).

Computation (K-fold CV)

K iterations, in each:

- train on $n-n/k$ samples
- score on n/k samples

K-fold CV - how to split?

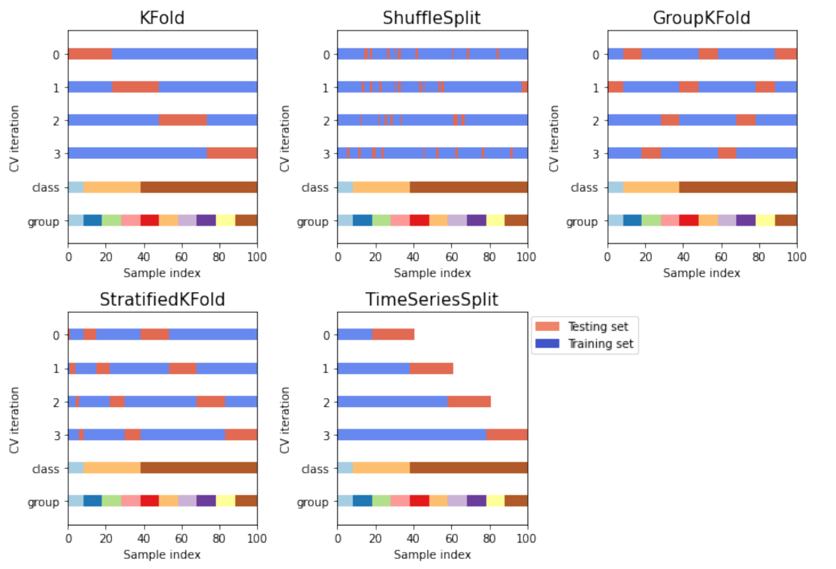

Figure 9: K-fold CV variations.

Selecting the right K-fold CV is very important for avoiding data leakage! (Also for training/test split.)

- if there is no structure in the data - shuffle split (avoid accidental patterns)
- if there is group structure - use a split that keeps members of each group in either training set, or validation set, but not both
- for time series data - use a split that keeps validation data in the future, relative to training data

Think about the task that the model will be asked to do in "production," relative to the data it is trained on! Refer to the function documentation for more examples.

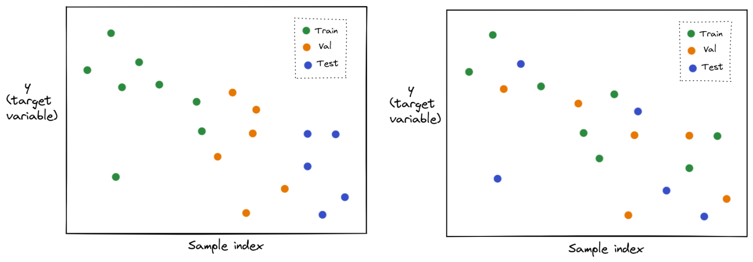

Figure 10: Example 1: The data is not homogeneous with respect to sample index, so splitting data data as shown on left would be a very bad idea - the training, validation, and test sets would not be similar! Instead, we should shuffle the indices before splitting the data, as shown on right.

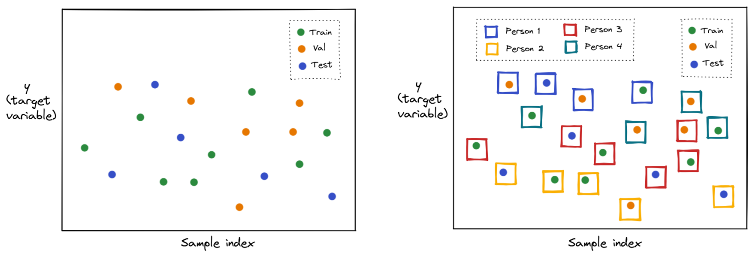

Figure 11: Example 2: The split on the left seems OK, unless (as shown on the right), each person contributes several samples to the dataset, and the value of y is similar for different samples from the same person. This is an example of data leakage. The model is learning from data from an individual, then it is validated and evaluated on data from the same individual - but in production, the model is expected to make predictions about individuals it has never seen. The training, validation, and evaluation process will have overly optimistic performance compared to production (and the model may overfit).

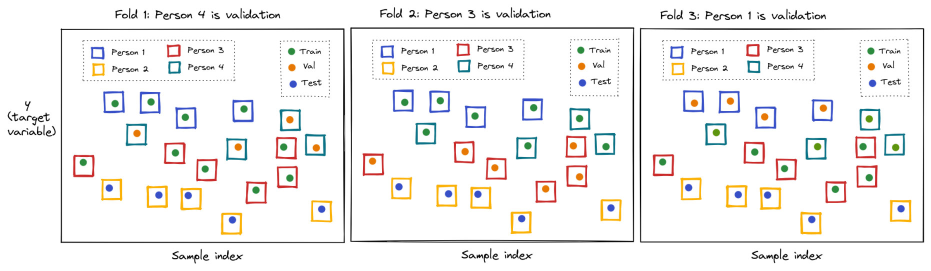

Figure 12: Example 2 - continued: Instead, we should make sure that each person is *only* in one type of “set” at a time (e.g. with GroupKFoldCV or equivalent).

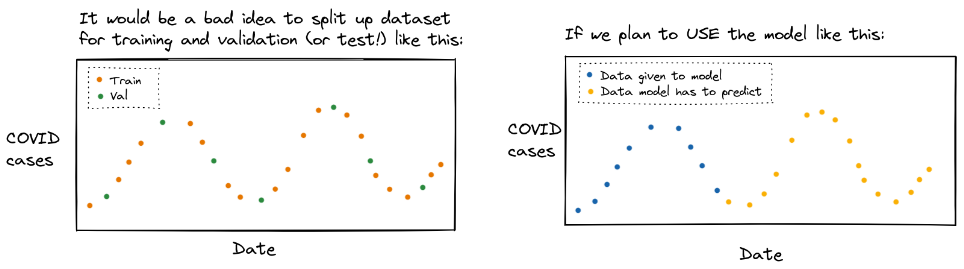

Figure 13: Example 3: if we would split this time series data as shown on the left, we would get overly optimistic performance in training/validation/evaluation, but then much worse error in production! (This is also an example of data leakage: the model learns from future data, and from adjacent data points, in training - but that data is not available during production.)

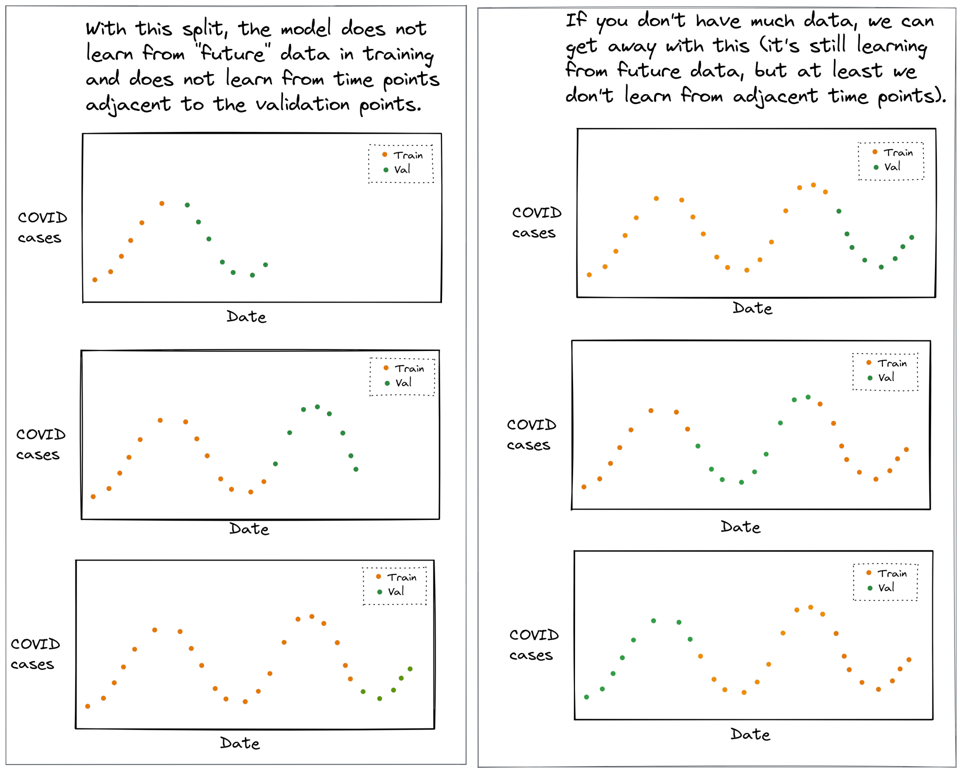

Figure 14: A better way would be to train and validate like this (example shown is 3-fold CV).

One standard error rule

- Model selection that minimizes mean error often results in too-complex model
- One standard error rule: use simplest model where mean error is within one SE of the minimum mean error

One standard error rule - algorithm (1)

- Given data X, y
- Compute score $S_{p,i}$ for model p on fold i (of K)
- Compute average ($\bar{S}_p$), standard deviation σ_p, and standard error of scores:

$$SE_p = \frac{\sigma_p}{\sqrt{K-1}}$$

One standard error rule - algorithm (2)

"Best score" model selection: $p^* = \text{argmin}_p \bar{S}_p$

One SE rule for "lower is better" scoring metric: Compute target score: $S_t = \bar{S}_{p^*} + SE_{p^*}$

then select simplest model with score lower than target:

$$p^{*,\text{1SE}} = \min\{p | \bar{S}_p \leq S_t\}$$

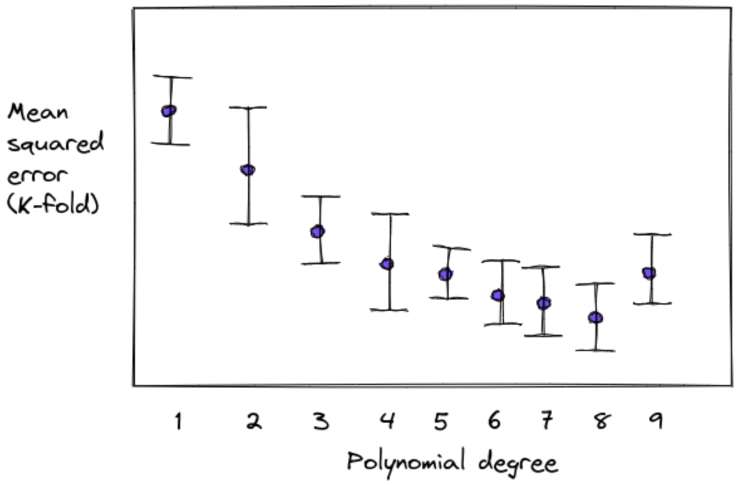

Figure 15: Model selection using one SE rule on MSE. The best scoring model is $d = 8$, but $d = 6$ is simplest model within one SE of the best scoring model, and so $d = 6$ would be selected according to the one-SE rule.

Note: this assumes you are using a "smaller is better" metric such as MSE. If you are using a "larger is better" metric, like R2, how would we change the algorithm?

One standard error rule - algorithm (3)

"Best score" model selection: $p^* = \text{argmax}_p \bar{S}_p$

One SE rule for "higher is better" scoring metric: Compute target score: $S_t = \bar{S}_{p^*} - SE_{p^*}$

then select simplest model with score higher than target:

$$p^{*,1\text{SE}} = \min\{p | \bar{S}_p \geq S_t\}$$

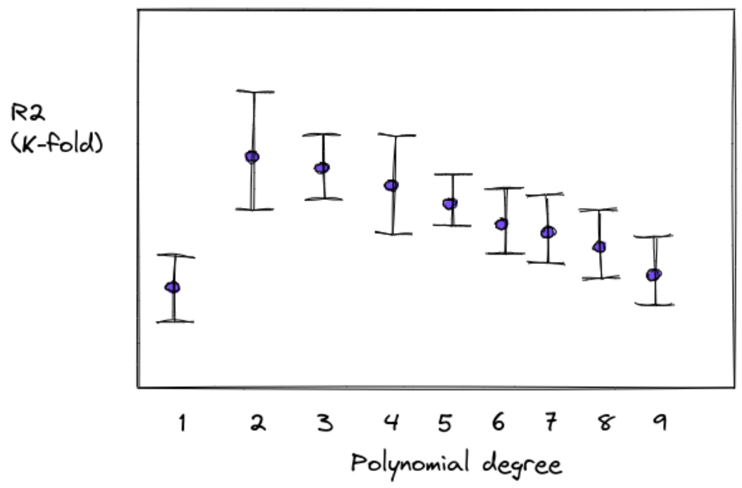

Figure 16: Model selection using one SE rule on R2. In this example, the best scoring model is $d = 2$, and there is no simpler model within one SE, so the one-SE rule would also select $d = 2$.

Placing computation

Placement options

Any "step" could be placed:

- before the train/test split
- after the split, outside loop
- in the first (outer) loop
- in the second (inner) loop

We want to place each "step" in the appropriate position to minimize the computation required, but also in order to use the split effectively! In placing a "step", we need to ask ourselves:

- does the result of this computation depend on the train/test split?
- does the result of this computation depend on the first loop variable?
- does the result of this computation depend on the second loop variable?

Note: the order of the loops (first loop over models, then over splits; or first loop over splits, then over models) is up to us - we can select the order that is most efficient for computation.

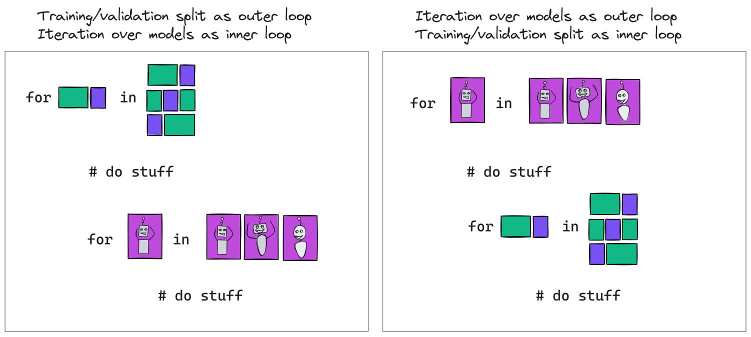
Training/validation split as outer loop
Iteration over models as inner loop
for
in
# do stuff
for
in
# do stuff
Iteration over models as outer loop
Training/validation split as inner loop
for
in
# do stuff
for
in
# do stuff

Figure 17: Possible arrangements of inner vs outer loop.

Data pre-processing should be considered part of model training, so steps where the value after pre-processing depends on the data, should use *only* the training data. For example -

- filling missing values with some statistic from the data (mean, median, max, etc.)
- standardizing (removing mean and scaling to unit variance) or other scaling

Example: design matrix for n-way interactions

Suppose we want to evaluate models for increasing n, where the model of n includes n-way interaction features. For example:

- Model 1: x_1, x_2, x_3
- Model 2: $x_1, x_2, x_3, x_1 \times x_2, x_1 \times x_3, x_2 \times x_3$
- Model 3: $x_1, x_2, x_3, x_1 \times x_2, x_1 \times x_3, x_2 \times x_3, x_1 \times x_2 \times x_3$

If we place the computation of the interaction features in the innermost loop, we will compute the same values repeatedly - and we don't need to! We can actually compute the entire design matrix *outside* the entire K-fold CV, and inside the K-fold CV -

- select from this overall matrix, the columns corresponding to the current model
- select from this overall matrix, the rows corresponding to the training/validation data for this split

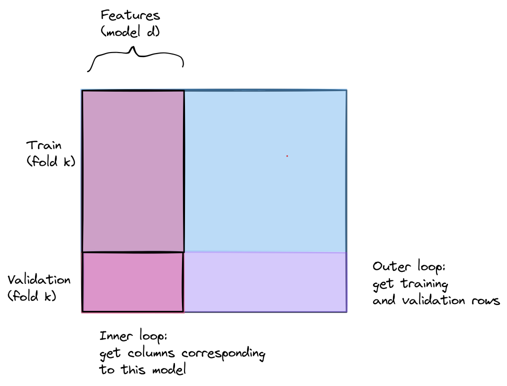

Figure 18: Slicing rows and columns from an "overall" matrix.

Example: design matrix for models with spline features

Suppose we want to evaluate models on a spline transformation of the data, with a fixed degree (e.g. $d = 2$) and an increasing number of "knots". For example:

- Model 1: 3 knots
- Model 2: 4 knots
- Model 3: 5 knots

(we will specify the positions of the knots ourselves, rather than having them be inferred from the data.)

In this example, we cannot put the computation of the spline features outside the K-fold CV - *all* the values in the "transformed" dataset are different in each model. (Unlike the previous example, there is no "repetition" of features from one model to the next.)

However, we can consider two *valid* ways to place the computation of spline features:

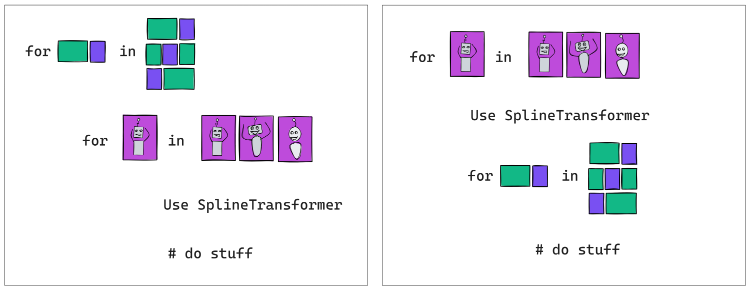

Figure 19: Either of these are valid ways to compute the spline features.

- In the first (left) case, however, we re-compute the spline features repeatedly for the same samples - we don't need to!
- Instead, we should use the second (right) loop order, and then in the inner loop, just select the rows corresponding to training and validation from the design matrix *for the given model*.