Model selection

Fraida Fund

Contents

A supervised machine learning "recipe" . . . 2
Model selection problems . . . 2
    Model order selection problem . . . 2
    Using loss function for model order selection? . . . 2
    Feature selection problem (1) . . . 3
    Feature selection problem (2) . . . 3
Validation . . . 3
    Hold-out validation . . . 3
    Hold-out validation (1) . . . 3
    Hold-out validation (2) . . . 3
    Problems with hold-out validation . . . 4
    K-fold cross validation . . . 4
    K-fold CV - algorithm (1) . . . 4
    K-fold CV - algorithm (2) . . . 4
    Leave-p-out CV . . . 6
    Computation (leave-p-out CV) . . . 6
    Computation (K-fold CV) . . . 6
    K-fold CV - how to split? . . . 6
One standard error rule . . . 9
    One standard error rule - algorithm (1) . . . 9
    One standard error rule - algorithm (2) . . . 9
    One standard error rule - algorithm (3) . . . 10
Placing computation . . . 10
    Placement options . . . 10

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**.

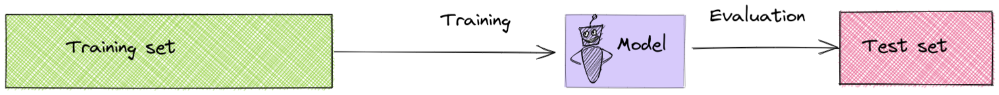

Figure 1: When we have only one model to consider, with no "hyperparameters".

Model selection problems

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

We'll look at two examples of model selection problems, but there are many more.

Model 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 loss function for model 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

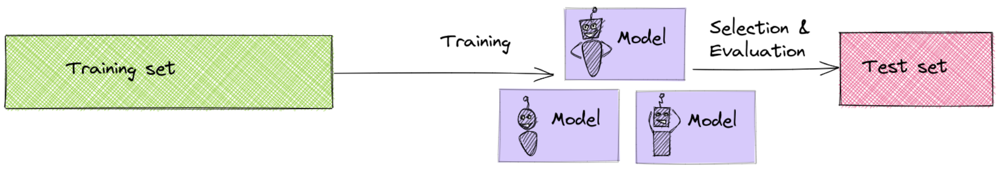

Figure 2: Selecting the best of multiple models - this approach does *not* work, because the loss function always decreasing with d (training error decreases with model complexity!)

Note that we shouldn't use the test data to select a model either - then we wouldn't have an "unused" data set on which to evaluate how well the model generalizes.

Feature selection problem (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
- **Feature selection problem**: fit a model with a small number of features

Feature selection problem (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, there's a unique OLS solution only if $n \geq d$
- For linear regression, when $N \geq p$, variance increases linearly with number of parameters, inversely with number of samples. (Not derived in class, but read extra notes posted after class at home.)

Validation

Hold-out validation

- 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

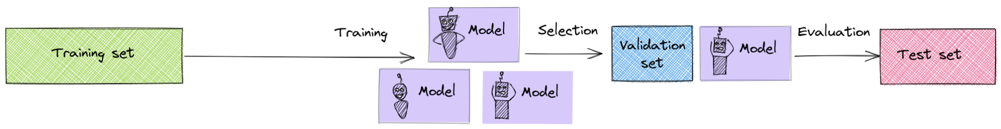

Figure 3: Model selection with a validation set.

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

Hold-out validation (1)

- 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 (2)

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

$$p^* = \operatorname*{argmin}_p S_p$$

- 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 simple 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

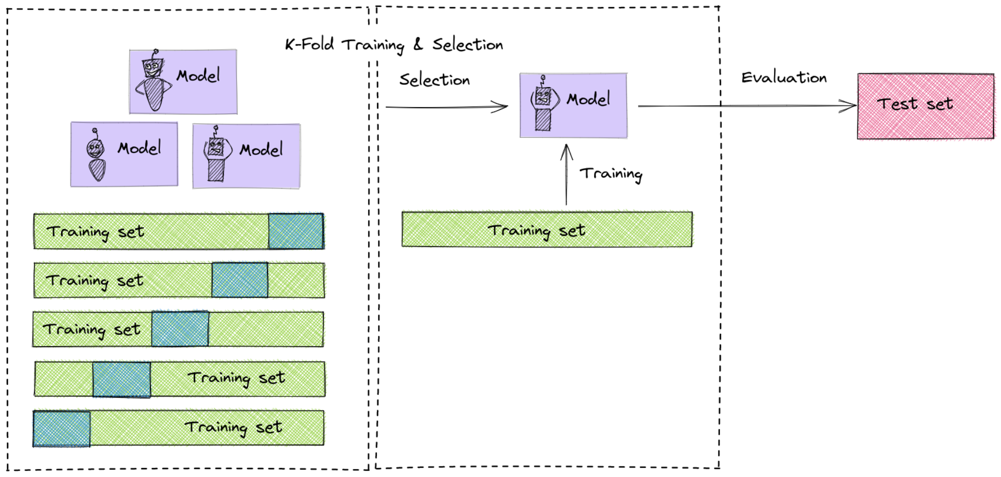

Figure 4: K-fold CV for model selection.

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**: $S_{p,i} = score(y_{v_i}, \hat{y}_{v_i,p})$

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

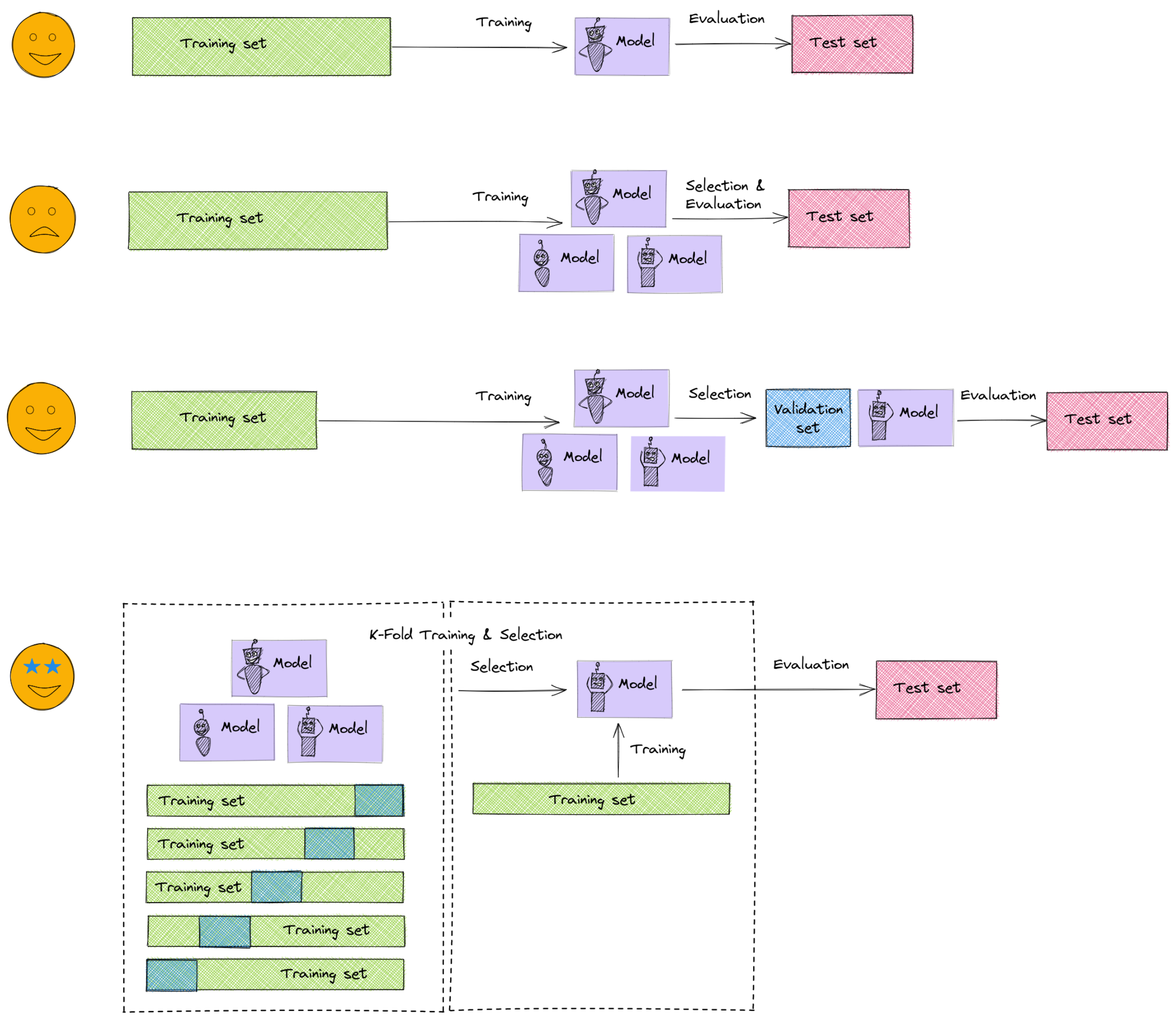

Figure 5: Summary of approaches. Source.

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?

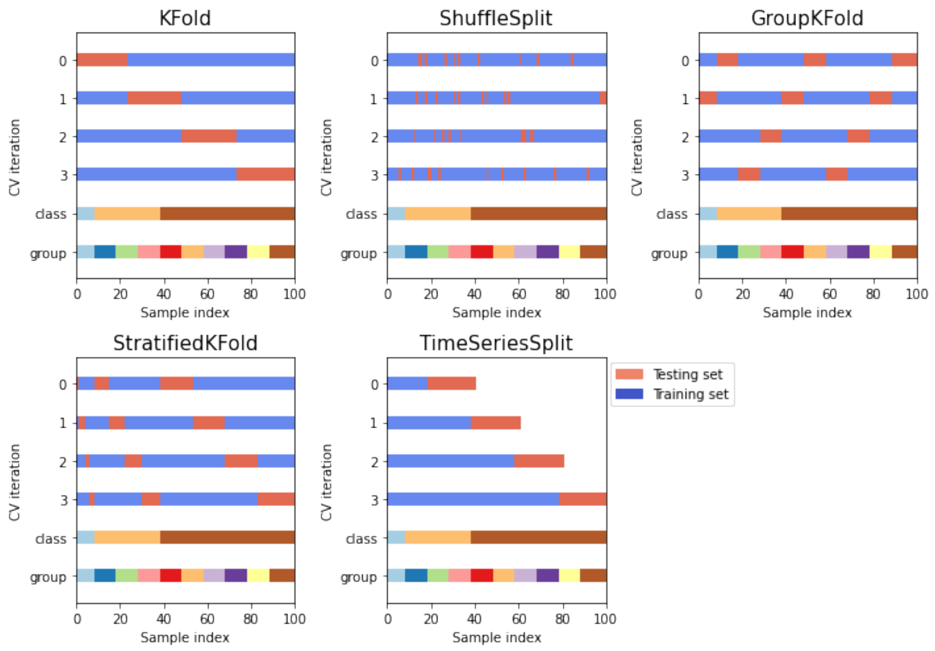

Figure 6: K-fold CV variations.

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

Refer to the function documentation for more examples.

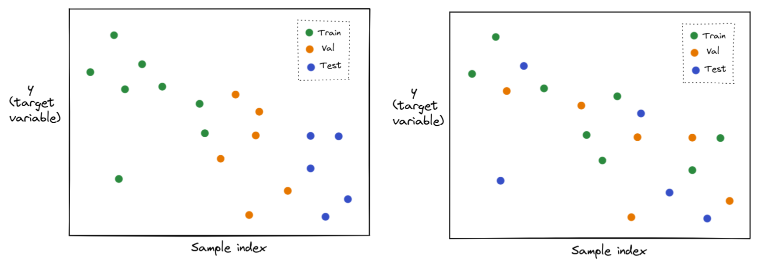

Figure 7: Example 1: The data is not homogeneous with respect to sample index, so splitting data data as shown on left would be a very bad idea - the training, validation, and test sets would not be similar! Instead, we should shuffle the indices before splitting the data, as shown on right.

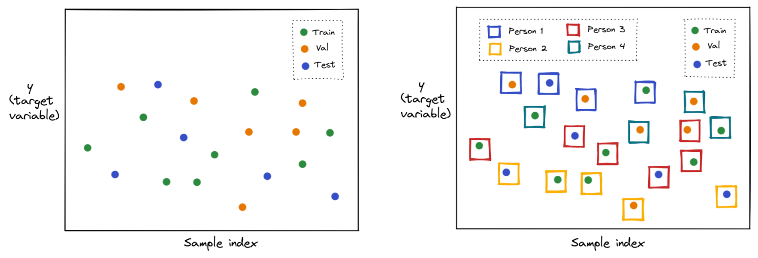

Figure 8: Example 2: The split on the left seems OK, unless (as shown on the right), each person contributes several samples to the dataset, and the value of y is similar for different samples from the same person. This is an example of data leakage. The model is learning from data from an individual, 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).

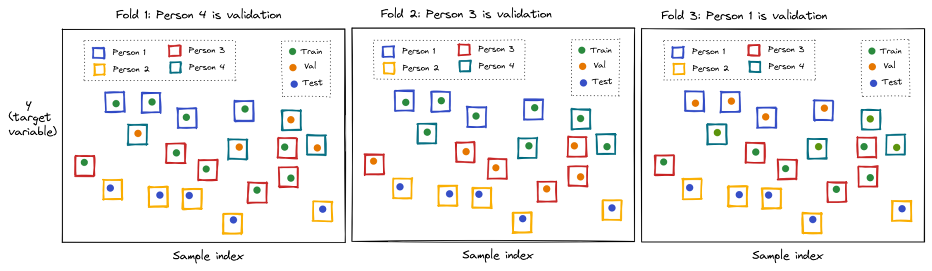

Figure 9: Example 2 - continued: Instead, we should make sure that each person is *only* in one type of “set” at a time (e.g. with GroupKFoldCV or equivalent).

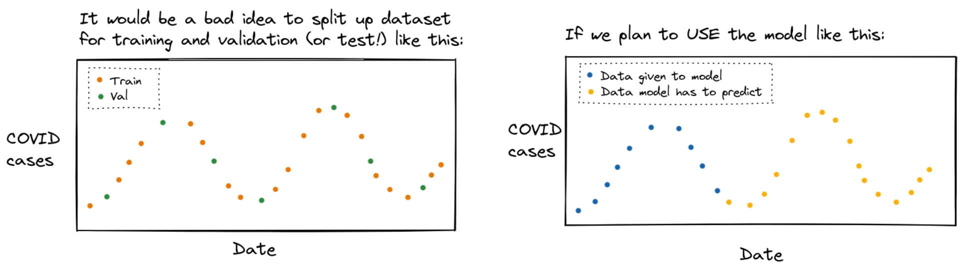

Figure 10: Example 3: if we would split this time series data as shown on the left, we would get overly optimistic performance in training/validation/evaluation, but then much worse error in production! (This is also an example of data leakage: the model learns from future data, and from adjacent data points, in training - but that data is not available during production.)

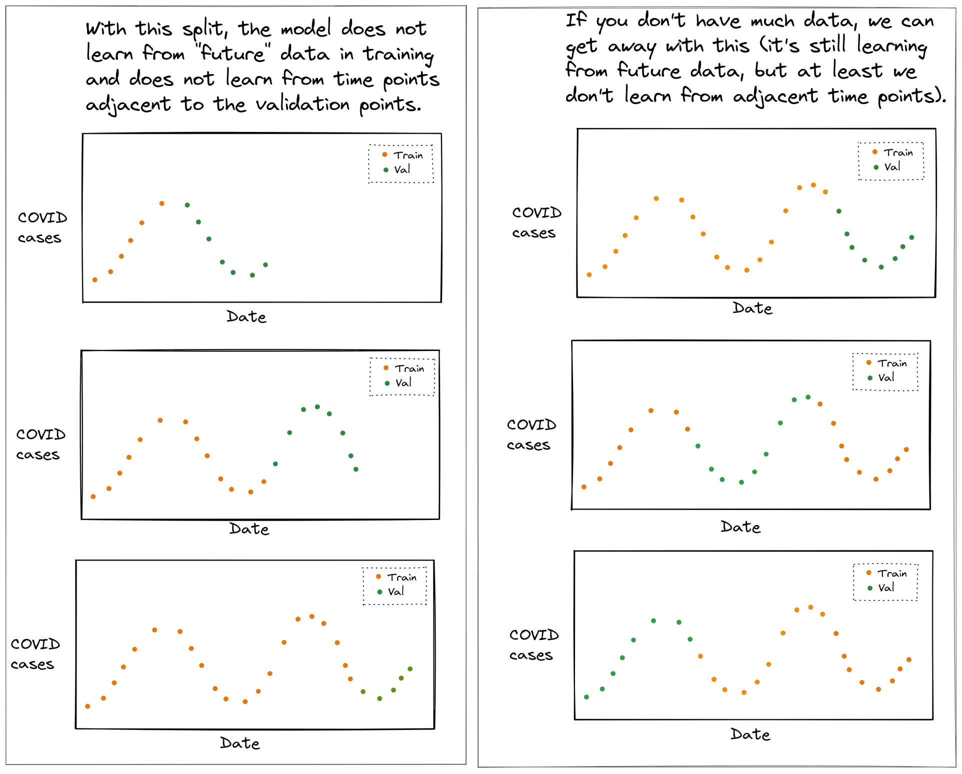

Figure 11: A better way would be to train and validate like this (example shown is 3-fold CV).

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:

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

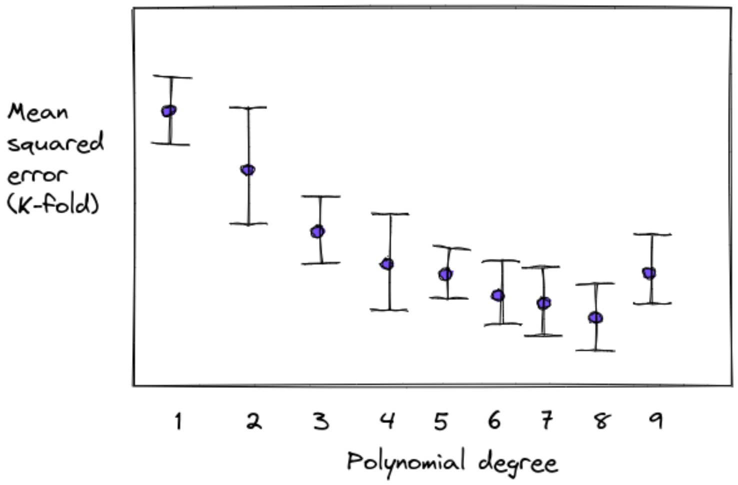

Figure 12: Model selection using one SE rule on MSE. The best scoring model is $d = 8$, but $d = 6$ is simplest model within one SE of the best scoring model, and so $d = 6$ would be selected according to the one-SE rule.

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:

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

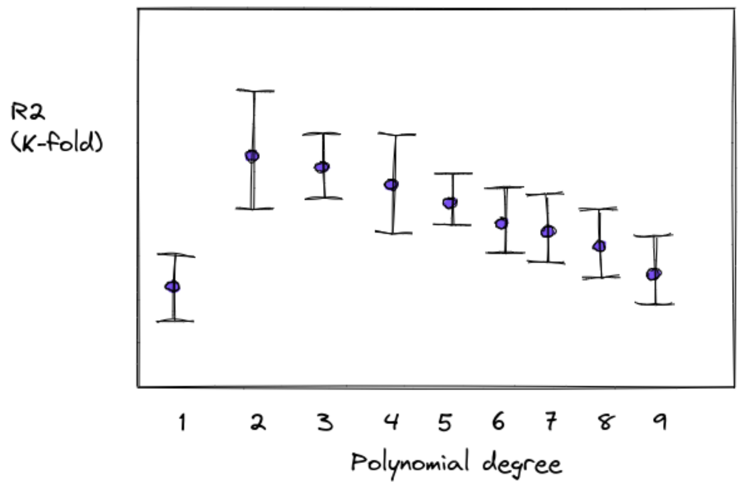

Figure 13: Model selection using one SE rule on R2. In this example, the best scoring model is $d = 2$, and there is no simpler model within one SE, so the one-SE rule would also select $d = 2$.

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.

Data pre-processing should be considered part of model training, so steps that depend on the data should not come before the train/test split. Examples:

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

The test data should never be used to compute these statistics.