Polynomial Regression

Overlearning, Training, Testing, and Validation Academic Year 1447 Term 1 Dr. Jomana Bashatah Slides adapted from Casten Lange

Overview

You will learn about:

- Overlearning in detail.
- Circumstances that make *overlearning* more likely to occur.
- Consequences of overlearning when predicting new data.
- Hyper-parameter tuning to avoid *overlearning*.
  - Validation
  - Cross Validation

Overlearning

If a model performs well when approximating the training data but does not perform well when it faces new data to predict outcomes.

Overlearning is one of the most pressing and still not fully solved problems in machine learning.

Circumstances that Can Lead to Overlearning

- If the training dataset does not have a sufficient number of observations.
- If the model considers many variables and thus contains many parameters to calibrate.
- If the underlying machine learning model is highly non-linear.

The Data

In what follows we use the Kings County Real Estate dataset.

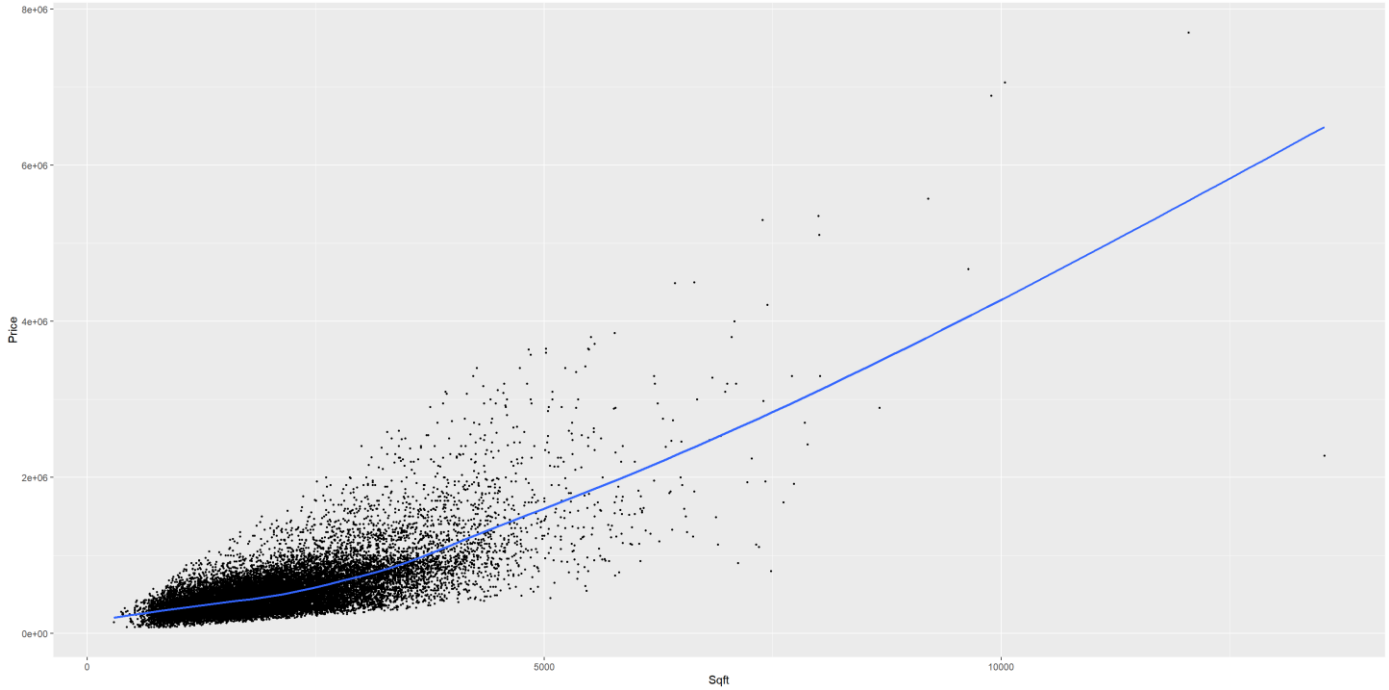
```
library(tidymodels); library(rio); library(janitor)

DataHousing = import("https://raw.githubusercontent.com/jomanab-cloud/ISE351-25-Presentations/80f7609c6bcb4887517dcaff62908459cee33e33/HousingData.csv") |>
  clean_names("upper_camel") |>
  select(Price, Sqft=SqftLiving)
```

We want to demonstrate *overlearning*. Therefore, we ceate conditions that **likely trigger overlearning**. Consequently, we work only with a very small training dataset (20 observations=0.1% of total observations. All other observations become testing data:

```
set.seed(777)
# initial_split(prop = 0.001, ...) randomly chooses 20 training observations
Split001=DataHousing %>%
  initial_split(prop = 0.001, strata = Price, breaks = 5)
DataTrain=training(Split001)
DataTest=testing(Split001)
```

Data Visualization

There seems to be a non-liner trend:



Data Structure

Price	Sqft
221900	1180
538000	2570
180000	770
604000	1960
510000	1680
1230000	5420

Polynomial Regression

Regular univariate prediction equation:

$$\widehat{Price} = \beta_1 Sqft + \beta_2$$

Polynomial univariate prediction equation (degree 5):

$$\widehat{Price} = \beta_1 Sqft + \beta_2 Sqft^2 + \beta_3 Sqft^3 + \beta_4 Sqft^4 + \beta_5 Sqft^5 + \beta_6$$

Polynomial Regression

Polynomial univariate prediction equation (degree 5):

$$\widehat{Price} = \beta_1 Sqft + \beta_2 Sqft^2 + \beta_3 Sqft^3 + \beta_4 Sqft^4 + \beta_5 Sqft^5 + \beta_6$$

We create $Sqft^2$, $Sqft^3$, $Sqft^4$, and $Sqft^5$ as new variables in the data and treat them as they were separate variables in a multivariate regression.

- This makes the regression **linear in variables** but **non-linear in data**.
  - Because *SQFT* is not only used in its original form, but also with various powers (squared, cubic, quartic, and quintic), the model is non-linear (in the data).
  - If we treat the variables (sqft) the same as any other variable in a multivariate linear OLS model, then each variable is multiplied by a parameter (beta) and added to the equation. Thus the model is a linear function as long as we interpret the different powers of sqft as separate variables.

Consequently we can OLS to find the optimal βs.

HOW THE DATA WOULD LOOK LIKE

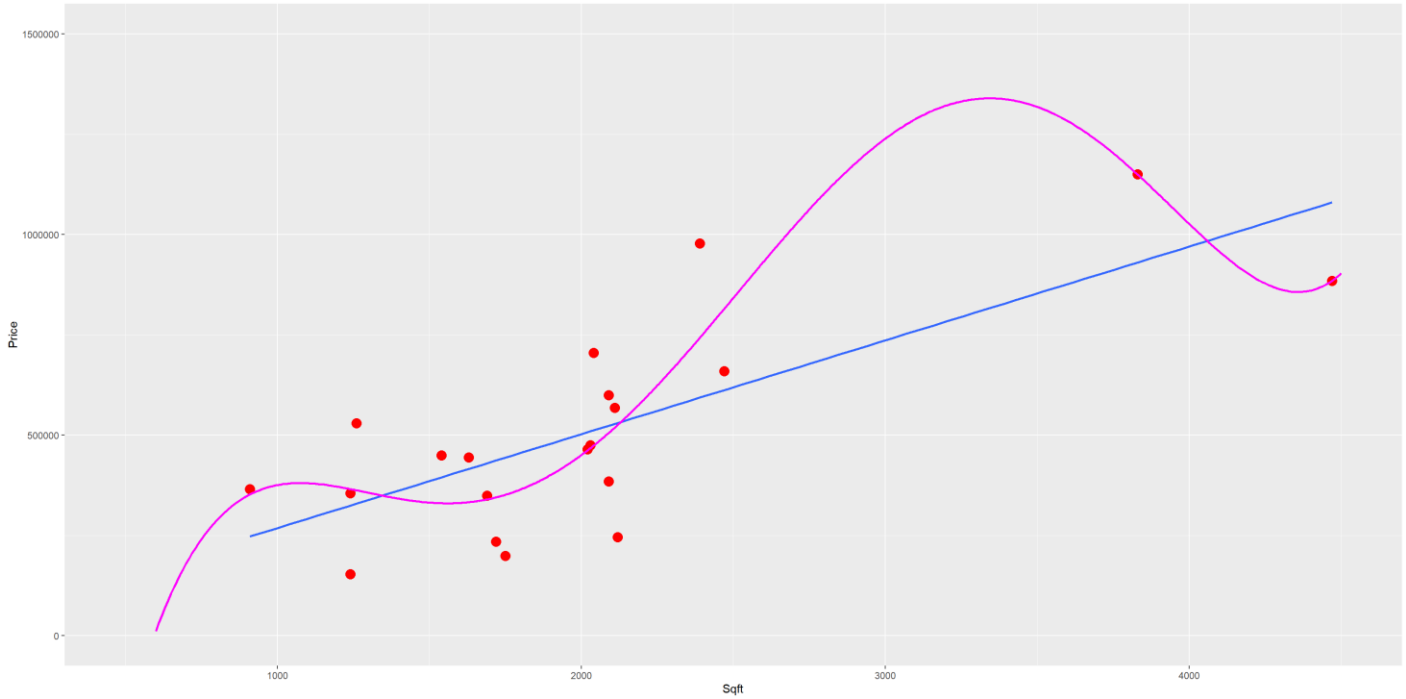
Price	Sqft	Sqft2	Sqft3	Sqft4	Sqft5
221900	1180	1392400	1643032000	1.938778e+12	2.287758e+15
538000	2570	6604900	16974593000	4.362470e+13	1.121155e+17
180000	770	592900	456533000	3.515304e+11	2.706784e+14
604000	1960	3841600	7529536000	1.475789e+13	2.892547e+16
510000	1680	2822400	4741632000	7.965942e+12	1.338278e+16
1230000	5420	29376400	159220088000	8.629729e+14	4.677313e+18

Polynomial Regression (degree=5) vs. Regular OLS

Aproximation of the Training Data

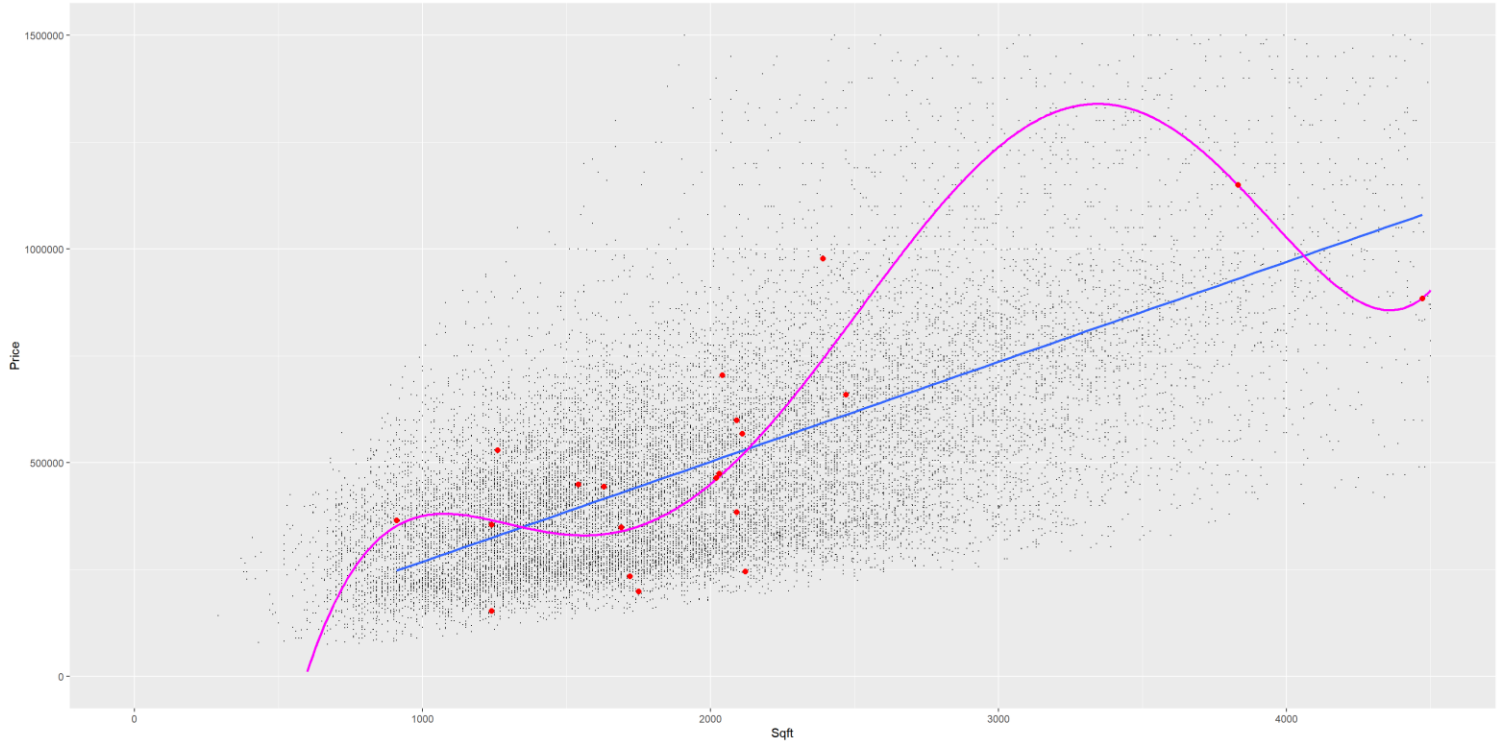
Polynomial Regression (degree=5) vs. Regular OLS

Aproximation of the Training Data



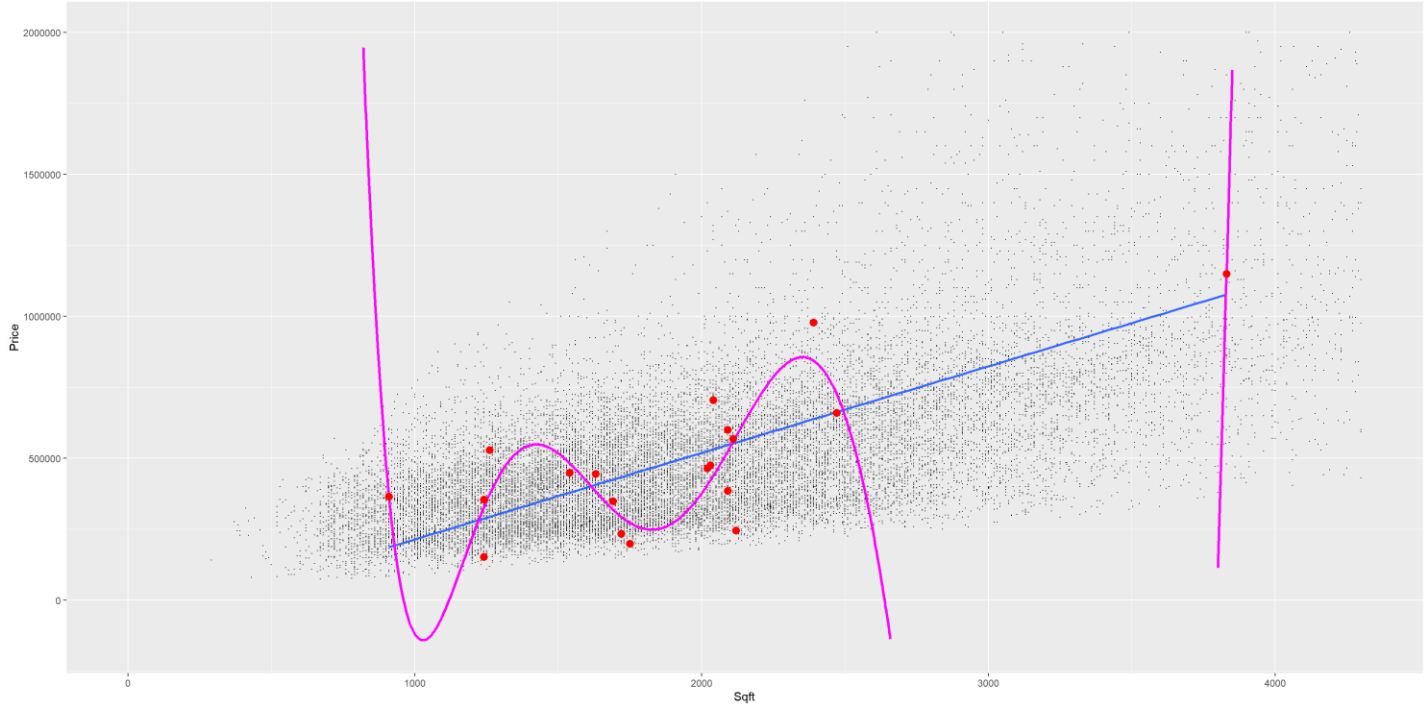
Polynomial Regression (degree=5) vs. Regular OLS

Training and Testing Data Performance



Polynomial Regression (degree=10) vs. Regular OLS

Training and Testing Data Performance



What do the Metrics Say?

OLS - training dataset

```
metrics(DataTrainWithPredBenchmOLS,
truth=Price, estimate=.pred)
```

.metric	.estimator	.estimate
rmse	standard	1.702083e+05
rsq	standard	5.570751e-01
mae	standard	1.393518e+05

OLS - testing dataset

```
metrics(DataTestWithPredBenchmOLS,
truth=Price, estimate=.pred)
```

.metric	.estimator	.estimate
rmse	standard	2.658887e+05
rsq	standard	4.928454e-01
mae	standard	1.677258e+05

What do the Metrics Say?

ploynomial (degree 5) - training dataset

```
metrics(DataTrainWithPredPolynomOLS
, truth=Price, estimate=.pred)
```

.metric	.estimator	.estimate
rmse	standard	1.364317e+05
rsq	standard	7.154233e-01
mae	standard	1.040470e+05

ploynomial (degree 5) - testing dataset

```
metrics(DataTestWithPredPolynomOLS,
truth=Price, estimate=.pred)
```

.metric	.estimator	.estimate
rmse	standard	9.994024e+07
rsq	standard	2.148350e-02
mae	standard	1.719470e+06

What do the Metrics Say?

ploynomial (degree 10) - training dataset

```
metrics(DataTrainWithPredPolynom10OLS, truth=Price, estimate=.pred)
```

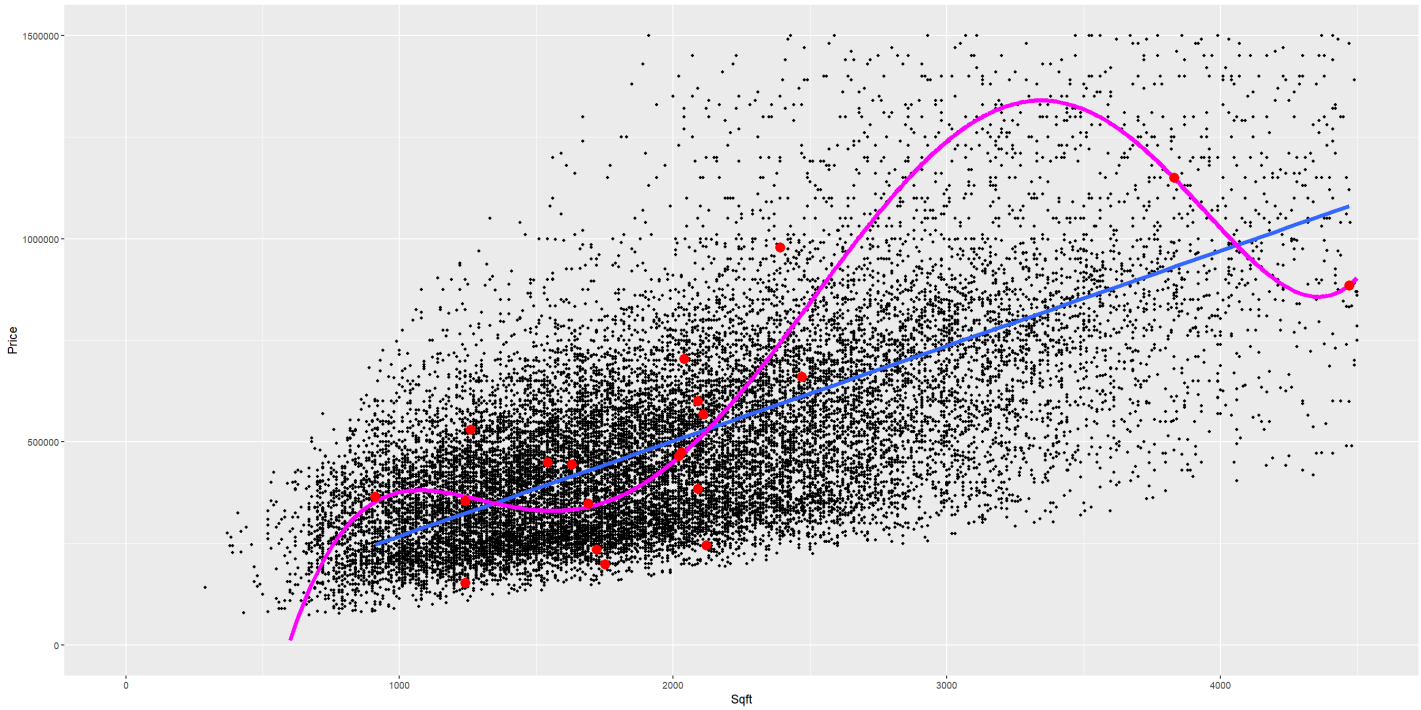
.metric	.estimator	.estimate
rmse	standard	1.234627e+05
rsq	standard	7.669549e-01
mae	standard	8.758509e+04

ploynomial (degree 10) - testing dataset

```
metrics(DataTestWithPredPolynom10OLS, truth=Price, estimate=.pred)
```

.metric	.estimator	.estimate
rmse	standard	1.437378e+11
rsq	standard	3.551700e-03
mae	standard	1.255949e+09

Why?



SUMMARY: POLYNOMIAL REGRESSION

- Overfitting occurs when a highly non-linear (flexible) model adjusts almost perfectly to a relatively small number of training observations, but the adjustment is so specific that the model fails when predicting the testing data.
  - Overfitting is sometimes compared to human learning behavior, when somebody learns facts by heart rather than understanding the underlying theory.
- If we do not have enough data, a polynomial regression with a high degree might lead to overlearning
- What is the right degree?
- We could try different degrees (e.g., 2, 3, 4, … 10) and see which model performs best.
- Which data are we using to measure performance? Training data (overlearning) and testing data (cannot be used for model optimization) are out.
- We could split off data from the training dataset (**validation data**). These *validation data* are not used to calculate the Bs. Instead, they are used to find the best setting for the degree of polynomial regression (aka hyper-parameter of polynomial regression).

Hyper-Parameters

- Hyper-Parameters are parameters other than the β parameters, because they can not be optimized by the optimizer.
- Hyper-Parameters are like settings for a machine learning model such as the number of polynomials (e.g., $Sqft^N$) to be considered for polynomial regression. Another example are the number of k Nearest Neighbors.
- Hyper parameters often make a model more or less complex and thus influence the quality of predicting but also the chance of overlearning.
- Hyper-parameters cannot be determined with an Optimizer based on training data like a model???s beta parameters. Therefore,we must utilize a trial and-error process to find suitable hyper-parameters.
  - This process is called hyper-parameter tuning.

- We cannot utilize the testing data for hyper-parameter tuning because this can extend overfitting into the testing data

Hyper-Parameter Tuning

Creating the Tuning Workflow

```
library(tidymodels); library(rio); library(janitor)
DataHousing=
import("https://raw.githubusercontent.com/jomanab-cloud/ISE351-25-
Presentations/80f7609c6bcb4887517dcaff62908459cee33e33/HousingData.csv") |>
clean_names("upper_camel") |>
select(Price, Sqft=SqftLiving)
set.seed(987)
Split80=DataHousing |>
initial_split(prop=0.8, strata=Price, breaks=5)
DataTrain=training(Split80)
DataTest=testing(Split80)
```

Building the workflow follows the same steps as before. We first create the recipe then the model design, then we combine both in a workflow

```
RecipeHousesPolynomOLS=recipe(Price~., data=DataTrain) |>
step_poly(Sqft, degree=tune(),
options=list(raw=TRUE))
ModelDesignLinRegr=linear_reg() |>
set_engine("lm") |>
set_mode("regression")
TuneWFModelHouses=workflow() |>
add_model(ModelDesignLinRegr) |>
add_recipe(RecipeHousesPolynomOLS)
```

Note:
1. In step_poly() where the argument degree= determines the highest power of sqft in the prediction equation, we do not assign a number for the degree.
- This makes sense because the aim of hyper-parameter tuning is to find this number (the degree of the Polynomial Model).
- Since the argument degree= needs to be determined, we use `tune()` as a placeholder
2. The workflow does not contain a fit() command to fit the model parameters to the training data.
- This also makes sense because the degree for the polynomial function is not determined, fitting the model is not possible.

Hyper-Parameter Tuning

- we want to evaluate the performance of several different degrees for the polynomial model.
  - we first have to decide which degrees we want to try out - arbitrary decision

- For example, we will try 1:10
  - in the tuning process, each value will be pushed to the tuning workflow one-by-one (replace the “tune()” placeholder)
  - each workflow will be fitted, and its predictive quality will be evaluated
  - the workflow with the best performance will be the best model

Hyper-Parameter Tuning

Validating the Tuning Results

- The question is: Which dataset will you use to validate different values for the hyper-parameters?
- You may be tempted to use the testing dataset, but keep in mind that the testing dataset should never be used for any type of optimization, including hyper-parameter tuning.
- Using the complete training dataset to find the best hyper-parameter value is also not an option because the best performing hyper-parameter value would be the one that triggers the highest degree of overfitting.
- We will look at two strategie to assess values of hyper-parameters without the testing dataset or the full training dataset
  - Validation Dataset
  - Cross Validation

Validation Dataset

- Randomly choose a number of observations from the training dataset, exclude them from the training, and assign them to an additional holdout dataset called the **validation dataset**
  - We use the observations from the validation dataset to assess the predective performance of the different hyper-parameter values

- Similar in concept to the testing dataset, but used in different stages of model development
  - Validation dataset: used during model design stage to find best hyper-parameter values

  - Testing dataset: used after the development of the model to assess overall prediciton quality

Validation Dataset in R

```
set.seed(879)
DataValidate=validation_split(DataTrain,
prop = 0.85, strat=Price)
```

DataValidate includes the complete training dataset, but observations are internally earmarked with *analysis* to indicate that an observation will be used for training, and with *assessment* to indicate that the observation will be used as validation data to assess hyper-parameter performance.

Validation Dataset - Pros and Cons

Using a validation dataset is appropriate for lareg datasets. For smaller datasets it comes with 2 disadvantages

1. Excluding observations from the training process and earmarking them for hyper-parameter assessment reduces the number of observations that are available for training.

2. The observations used for the assessment of hyper-parameters are randomly chosen. This bears the risk that, by accident, an unusual validation dataset might be created (the risk is higher for smaller training datasets).Evaluating hyper-parameters based on unusual assessment observations might lead to a sub-par choice of hyper-parameter values.

Cross Validation

- Instead of using one dataset where observations are earmarked for training or hyper-parameter assessment,the Cross-Validation procedure creates multiple training/assessment datasets called folds or resamples.

- These folds differ only by which observations are chosen for training and which ones are used for hyper-parameter assessment.

CROSS VALIDATION (4-FOLD)

For each hyper-parameter setting:

1. Splits off validation data from training data (e.g. last quarter)

2. Runs the model and calculates metrics based on validation data.

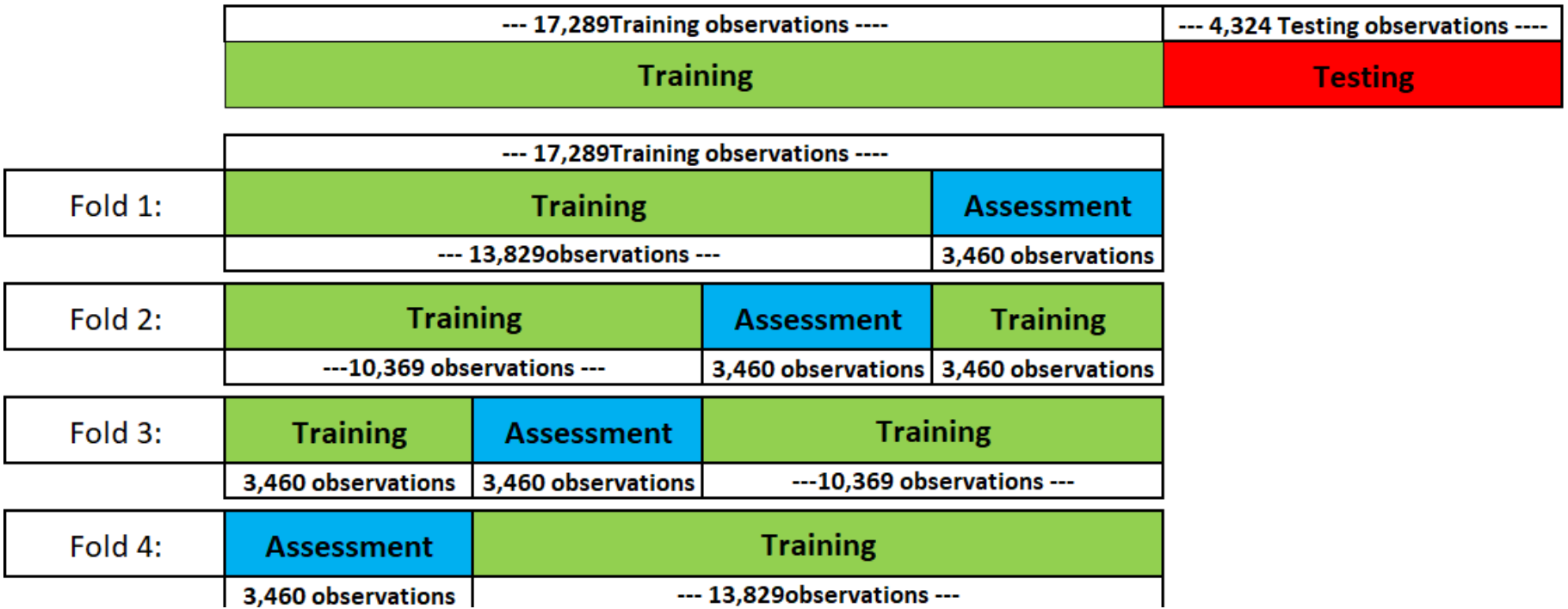
3. Splits off validation data from training data (next quarter)

4. Repeats steps 2 – 3 four times.

. . .

We end up with four results for each hyper-parameter setting. We calculate the average of the four results as an result for that specific hyper parameter.

Crossvalidation — The Idea Behind It



Crossvalidation - Pros and Cons

- The advantage of Cross-Validation is that different sets of observations are used for assessment (the mean prediction error is used to assess overall performance) and all observations of the training data at some stage of model assessment are used for validation. Therefore, the risk of an unusual assessment dataset is mitigated.
- The disadvantage of Cross-Validation is that each hyper-parameter setup needs to be trained and assessed separately for each of the folds. Computation time increases exponentially with the number of hyper-parameters tried out and proportionally with the number of folds used.

Cross-Validation in R

```
set.seed(987)
FoldsHouses=vfold_cv(DataTrain, v=4,
strata=Price)
```

10 Steps to Create a Model, Tune it, and Predict

The **10 general steps** are:

1. Generating **training and testing data** with `initial_split(),training(),testing()`.

```
set.seed(987)
Split80=MyData |>
initial_split(prop=0.8, strata=OUTCOME_VARIABLE, breaks=5)
DataTrain=training(Split80)
DataTest=testing(Split80)
```

2. Create **recipe** to determine predictor and outcome variables. Optionally add one or more `step_X()` commands.

```
Recipe=recipe(<OUTCOME VARIABLE>~<PREDICTOR VARIABLE(S)>,
data=DataTrain) |>
step_<NAME OFSTEP>(<ARGUMENT(S) OFSTEP>)
```

3. Create **model design** and mark parameters to be `tune()` ed. without `fit()`

```
ModelDesign=<NAME OFML-COMMAND>(<ARGUMENT(S) OFCOMMAND>) |>
set_engine("<PACKAGE NAME>") |>
set_mode("<MODE>")
```

10 Steps to Create a Model, Tune it, and Predict

4. Create **workflow** by `add_recipe()` and `add_model()`

```
TuneWFModel=workflow() |>
add_recipe(Recipe) |>
add_model(ModelDesign)
```

5. Create a **hyper-parameter grid** containing the hyper-parameter combinations to be validated.

```
ParGrid=data.frame(<HYPER-PAR1>=c(<LIST OFVALUES>),
<HYPER-PAR2>=c(<LIST OFVALUES>), <ETC>)
ParGrid=data.frame(neighbors=c(1,3,6))
```

6. Create **cross validation datasets** (aka *resamples*) containing the folds (use commands `vfold()`).

```
FoldsForTuning=vfold_cv(DataTrain, v=10, strata=<OUTCOME
VARIABLE>)
```

10 Steps to Create a Model, Tune it, and Predict

7. **Tune** the machine learning model with `tune_grid()` and track specific metrics defined by `metric_set()`. **Runs all hyper-parameter combinations for all folds.**

```
TuneResults=tune_grid(TuneWFModel, resamples=FoldsForTuning,
grid=ParGrid, metrics=metric_set(<LIST OFMETRICS>)),
control_grid(verbose=TRUE))
```

8. **Extract the best hyper-parameter combination** from the tuning results based on selected metrics (use `select_best()`)

```
BestHyperPar=select_best(TuneResults, "<METRIC>")
```

9. **Finalize the model** by training it with the full set of training data with the best hyper-parameter combination (see `finalize_workflow() %>% fit()`).

```
BestWFModel=TuneWFModel |>
finalize_workflow(BestHyperPar) |>
fit(DataTrain)
```

10 Steps to Create a Model, Tune it, and Predict

10. **Assessing predictive quality** of the final model by using the testing dataset to predict (see `augment() %>% metrics()`).

```
DataTestWithPredBestModel=augment(BestWFModel, DataTest)
metrics(DataTestWithPredBestModel,
truth=<OUTCOME VARIABLE>,
estimate=.pred)
```

CROSS VALIDATION FOR POLYNOMIAL REGRESSION AND THE KING COUNTY REALESTATE DATASET

Splitting the Data

```
set.seed(987)

Split80=DataHousing %>%
  initial_split(prop = 0.8, strata = Price,
breaks = 5)
DataTrain=training(Split80)
DataTest=testing(Split80)

print(Split80)
<Training/Testing/Total>
<17289/4324/21613>
```

Creating the Recipe

```
RecipeHousesPolynomOLS=recipe(Price~.,
data=DataTrain) |>
step_poly(Sqft, degree=tune(),
options=list(raw=TRUE))
```

Creating the Model Design

```
ModelDesignLinRegr=linear_reg() |>
set_engine("lm") |>
set_mode("regression")
```

Creating the Workflow

```
TuneWFModelHouses=workflow() |>
add_model(ModelDesignLinRegr) |>
add_recipe(RecipeHousesPolynomOLS)
```

Creating the Hyper-Parameter Grid

```
ParGridHouses=data.frame(degree=c(1:10))
print(ParGridHouses)
   degree
1       1
2       2
3       3
4       4
5       5
6       6
7       7
8       8
9       9
10     10
```

Creating the Cross-Validation Datasets

```
FoldsHouses=vfold_cv(DataTrain, v=4,
strata=Price)
```

Tune the ML Model

```
TuneResultsHouses=tune_grid(TuneWFModelHou
ses, resamples=FoldsHouses,
grid=ParGridHouses,
metrics=metric_set(rmse,rsq,mae))
```

Extract the Best Hyper-parameter Combination

```
BestHyperPar=select_best(TuneResultsHouses
, metric="rmse")
print(BestHyperPar)
# A tibble: 1 × 2
  degree .config
   <int> <chr>
1      6 Preprocessor06_Model1
```

Finalize the Model

```
BestWFModelHouses=TuneWFModelHouses |>
finalize_workflow(BestHyperPar) |>
fit(DataTrain)
print(BestWFModelHouses)
```

```
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: linear_reg()

── Preprocessor ────────────────────────────────────────────────────────────────
1 Recipe Step

• step_poly()

── Model ───────────────────────────────────────────────────────────────────────

Call:
stats::lm(formula = ..y ~ ., data = data)

Coefficients:
(Intercept)  Sqft_poly_1  Sqft_poly_2  Sqft_poly_3  Sqft_poly_4  Sqft_poly_5
 -1.297e+04    6.569e+02   -4.902e-01    2.051e-04   -3.758e-08    3.184e-12
Sqft_poly_6
 -9.933e-17
```

Assess the Predictive Quality of the Model

```
DataTestWithPredBestModel=augment(BestWFModelHouses, DataTest)
metrics(DataTestWithPredBestModel, truth=Price, estimate=.pred)
```

.metric	.estimator	.estimate
rmse	standard	2.407062e+05
rsq	standard	5.856548e-01
mae	standard	1.649870e+05

PROJECT: TUNING A K-NEAREST NEIGHBORS MODEL

Project Description

In Lecture 4, you developed a k-Nearest Neighbor model to predict the species of a penguin. We arbitrarily set *k=4* to consider the four nearest neighbors.

Start by loading the required libraries and read the data

```
# A tibble: 10 × 3
   Species BillLengthMm BodyMassG
   <fct>          <dbl>     <int>
 1 Adelie          39.1      3750
 2 Adelie          39.5      3800
 3 Adelie          40.3      3250
 4 Adelie          36.7      3450
 5 Adelie          39.3      3650
 6 Adelie          38.9      3625
 7 Adelie          39.2      4675
 8 Adelie          34.1      3475
 9 Adelie          42        4250
10 Adelie          37.8      3300
```

Step 1 - Generating Training and Testing Data

As before, we will use the penguin dataset and split the data into training (DataTrain) and testing (DataTest).
We will use the same seed (876) so the random split will be identical to the one we had with the k=4 model.

Species	BillLengthMm	BodyMassG
Adelie	40.3	3250
Adelie	36.7	3450
Adelie	39.3	3650
Adelie	38.9	3625
Adelie	39.2	4675
Adelie	42.0	4250

Step 2 - Create a Recipe

This is the same recipe as before. We will be dropping NAs and normalizing all predictor variables.

Step 3 - Create a Model Design

Use the appropriate ML model "*knn*". Remember that we will not specify the number of neighbors, but we will use `tune()` placeholder

```
K-Nearest Neighbor Model Specification
(classification)

Main Arguments:
  neighbors = tune()
  weight_func = rectangular

Computational engine: kknn
```

Step 4 - Add the Recipe and the Model Design to a Workflow

```
══ Workflow ════════════════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: nearest_neighbor()

── Preprocessor ────────────────────────────────────────────────────────────────
2 Recipe Steps

• step_naomit()
• step_normalize()

── Model ───────────────────────────────────────────────────────────────────────
K-Nearest Neighbor Model Specification (classification)

Main Arguments:
  neighbors = tune()
  weight_func = rectangular

Computational engine: kknn
```

Step 5 - Create a Hyper-Parameter Grid

Later, when tuning is executed in Step 7, values reaching from 1 ??? 15 for the hyper-parameter neighbors shall be tried out.
You need to provide these values in a data frame column that is named the same as the hyper-parameter.

```
   neighbors
1          1
2          2
3          3
4          4
5          5
6          6
7          7
8          8
9          9
10        10
11        11
12        12
13        13
14        14
15        15
```

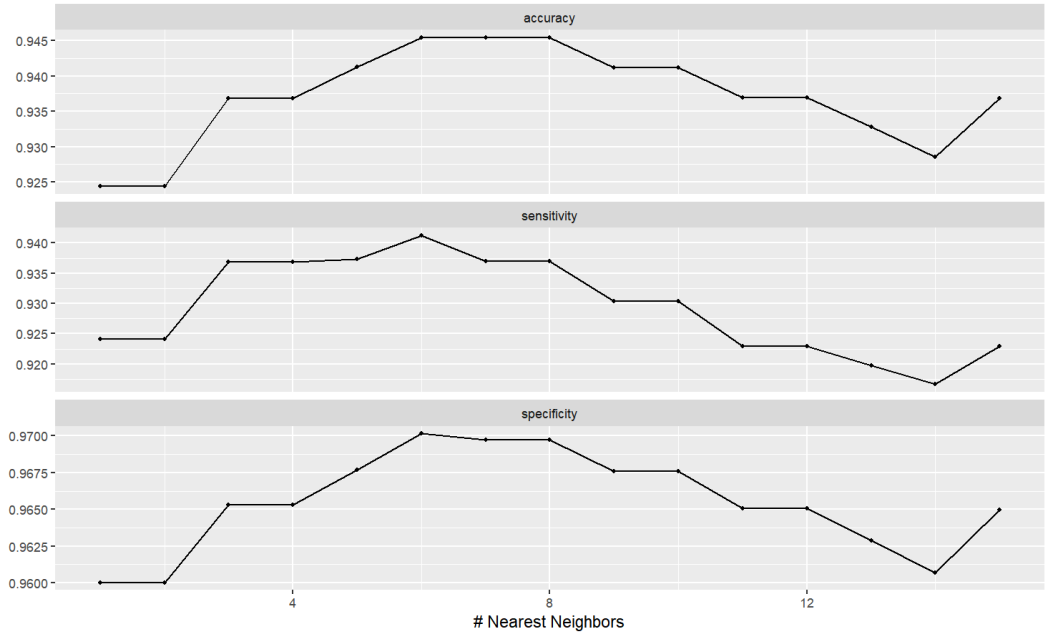
Step 6 - Creating Resamples for Cross-Validation

The values you have created above for k (hyper-parameter neighbors) will be evaluated later using five folds (resamples). Each fold contains the complete training data, but different sections are used for training and assessment in each fold.

```
#  5-fold cross-validation using stratification
# A tibble: 5 × 2
  splits           id
  <list>           <chr>
1 <split [189/49]> Fold1
2 <split [190/48]> Fold2
3 <split [191/47]> Fold3
4 <split [191/47]> Fold4
5 <split [191/47]> Fold5
```

Step 7 - Tune the Workflow and Train All Models

Now it is time to run the tuning procedure using the tune_grid() command. Be patient because it will take some time to fully execute. Since we have to try out 15 parameters and use five folds for each model, the tune_grid() command has to fit 75 models (15???5=75) .



Step 8 - Extract the Best Hyper-Parameter(s)

All assessment results for the specified metrics are stored in the tuning object TuneResults. Use the select_best() command to extract the best hyper-parameter (value for neighbors) for the metric you specified:

```
# A tibble: 1 × 2
  neighbors .config
      <int> <chr>
1         6 Preprocessor1_Model06
```

Step 9 - Finalize and Train the Best Workflow Model

The finalize_workflow() command will use the value from BestHyperPar to substitute the tune() placeholder in the R object TuneWFModel (created in Steps 2 - 4). The hyper-parameter is now set to `neighbors=6` completing the workflow.

use the command fit(DataTrain) to calibrate the workflow to the training data and save the result into WFModelBest.

```
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: nearest_neighbor()
── Preprocessor ────────────────────────────────────────────────────────────────
2 Recipe Steps
• step_naomit()
• step_normalize()
── Model ───────────────────────────────────────────────────────────────────────
Call:
kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(6L,     data, 5),
kernel = ~"rectangular")

Type of response variable: nominal
Minimal misclassification: 0.06302521
Best kernel: rectangular
Best k: 6
```

Step 10 - Assess Prediction Quality Based on the Testing Data

Since WFModelBest is a fitted model, you can use it for predictions. In this last step, you will use the augment() command to predict Species. The augment() command will then add the prediction results as column .pred to the testing data.

Use the conf_mat() command to compare the predictions in column .pred to the true values to create a confusion matrix.

```
          Truth
Prediction  Adelie Chinstrap Gentoo
  Adelie        41         1      1
  Chinstrap      2        19      0
  Gentoo         3         1     36
```