Lecture 9/5

Supervised Learning (Part II)

DS 6030 | Fall 2024

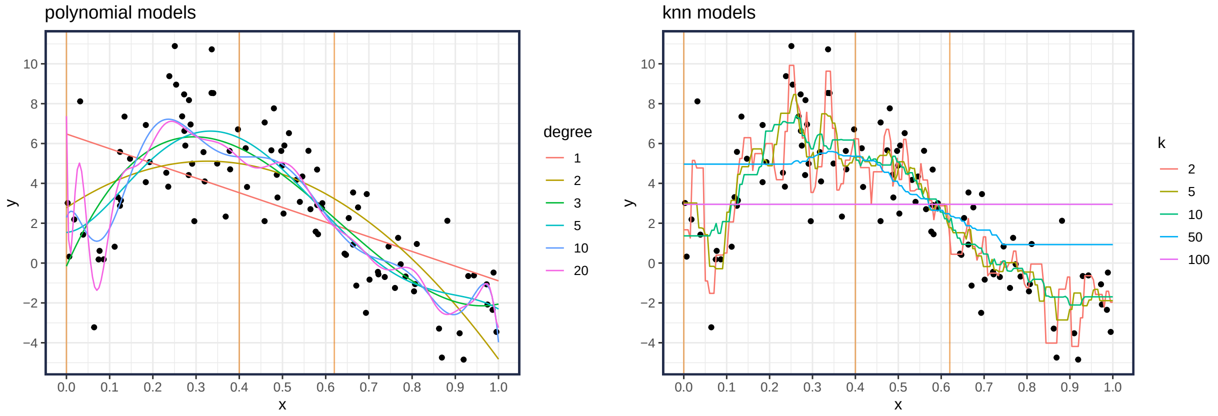
supervised_2.pdf

Contents

know exactly what the bias variance trade off is

1 Training Data and Model Fits

Recall from last class that we considered several models from two model families (nearest neighbor and polynomial).

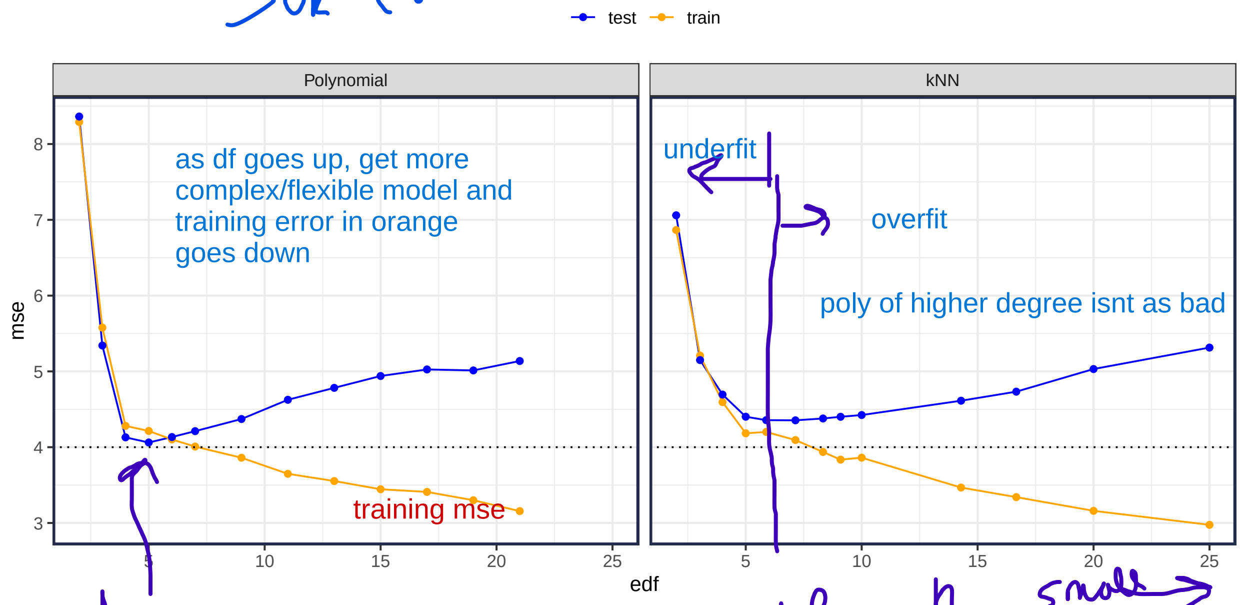


wanted model that minimized expected prediction error which was MSE in this case: E[Y-fhat(x))^2]

2 Evaluate Simulated Test Data (Best Model)

I simulated 50,000 test observations, evaluated the predictions from each model, and recorded the estimated MSE/Risk.

approx: $\frac{1}{50k}\sum(y_i - \hat{f})^2 \approx$



as df goes up, get more complex/flexible model and training error in orange goes down

training mse

underfit

overfit

poly of higher degree isnt as bad

deg = 4 = best MSE

edf = $\frac{n}{k}$

small k →

left=underfit=too simple, right=overfit, too complex

Polynomial

degree	edf	mse_train	mse_test
1	2	8.29	8.36
2	3	5.58	5.34
3	4	4.28	4.13
4	5	4.21	4.06
5	6	4.10	4.13
6	7	4.01	4.21
8	9	3.86	4.37
10	11	3.65	4.63
12	13	3.55	4.78
14	15	3.44	4.94
16	17	3.41	5.03
18	19	3.30	5.01
20	21	3.16	5.14

smallest

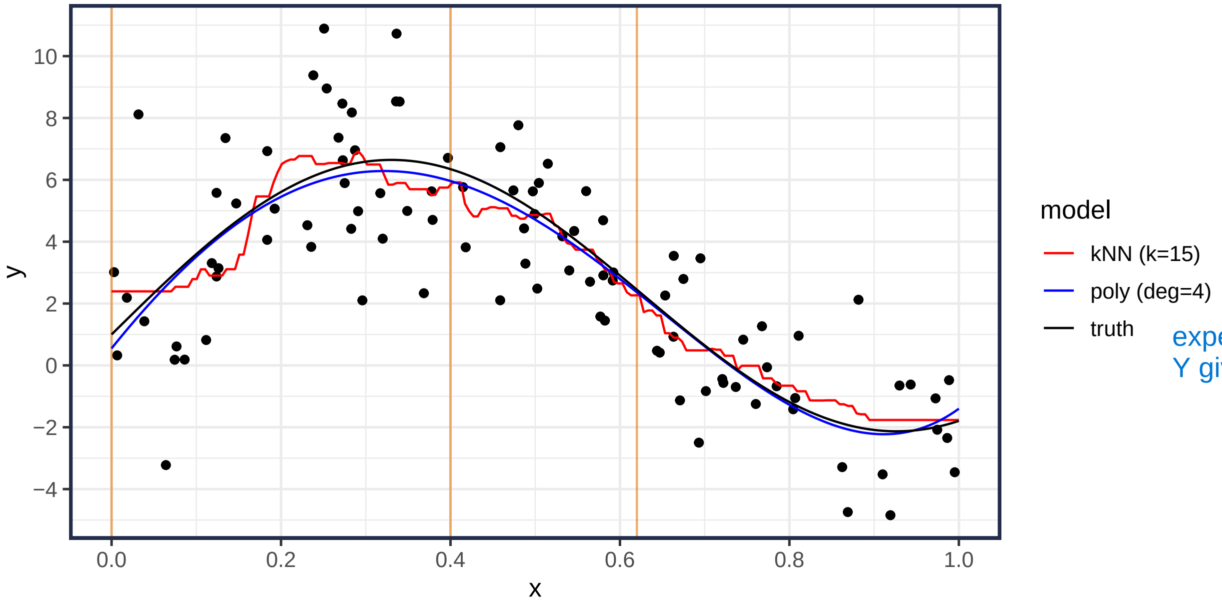
Nearest Neighbor

k	edf	mse_train	mse_test
50	2.00	6.87	7.06
33	3.03	5.21	5.15
25	4.00	4.59	4.69
20	5.00	4.18	4.40
17	5.88	4.20	4.36
14	7.14	4.09	4.35
12	8.33	3.94	4.38
11	9.09	3.84	4.40
10	10.00	3.86	4.42
7	14.29	3.47	4.61
6	16.67	3.34	4.73
5	20.00	3.16	5.03
4	25.00	2.98	5.31

optimal, smallest

Observations:

- as the flexibility increases, both classes of models *overfit*.
  - overfit means model too complex
  - underfit means model is not complex enough
  - see discrepancy between training and test performance
- The polynomial with degree = 4 has the best test performance with an approximate $\text{MSE} = 4.06$.
- The optimal $\text{MSE} = 4$.
  - I only know this because I know how the data was generated



expected value of Y given X: E[Y|X]

3 Ensemble Models

Can we use the collective *wisdom of the crowds* to help make a better prediction?

Here is our survey answers

Last class you gave your votes for which model you thought was best:

model	edf	number of votes
nearest neighbor (k=5)	20.0	14
polynomial (deg=3)	4.0	9
polynomial (deg=2)	3.0	7
nearest neighbor (k=10)	10.0	7
polynomial (deg=5)	6.0	6
polynomial (deg=10)	11.0	4
nearest neighbor (k=20)	5.0	2
nearest neighbor (k=50)	2.0	1
nearest neighbor (k=25)	4.0	1
nearest neighbor (k=15)	6.7	1
polynomial (deg=6)	7.0	1
nearest neighbor (k=13)	7.7	1
nearest neighbor (k=9)	11.1	1
polynomial (deg=11)	12.0	1
polynomial (deg=12)	13.0	1
polynomial (deg=16)	17.0	1
polynomial (deg=20)	21.0	1
nearest neighbor (k=4)	25.0	1
nearest neighbor (k=2)	50.0	1

An *ensemble model* is one that combines several models together. One approach is to create a ensemble model which uses as its prediction a weighted sum of the individual model predictions.

$$f_w(x) = \sum_{j=1}^{p} w_j f_j(x)$$

weighted sum of multiple predictions from different models

In our specific example, we will use your votes as the weights to create an ensemble

$$\hat{f}_w(x) = \frac{14}{61} f_{\text{knn}}(x, k=5) + \frac{9}{61} f_{\text{poly}}(x, deg=3) + \frac{7}{61} f_{\text{knn}}(x, k=10) + \ldots + \frac{1}{61} f_{\text{poly}}(x, deg=20)$$

prop who picked that is the weights

linear combo of the predictions from several different models

and the corresponding **test** performance is given by

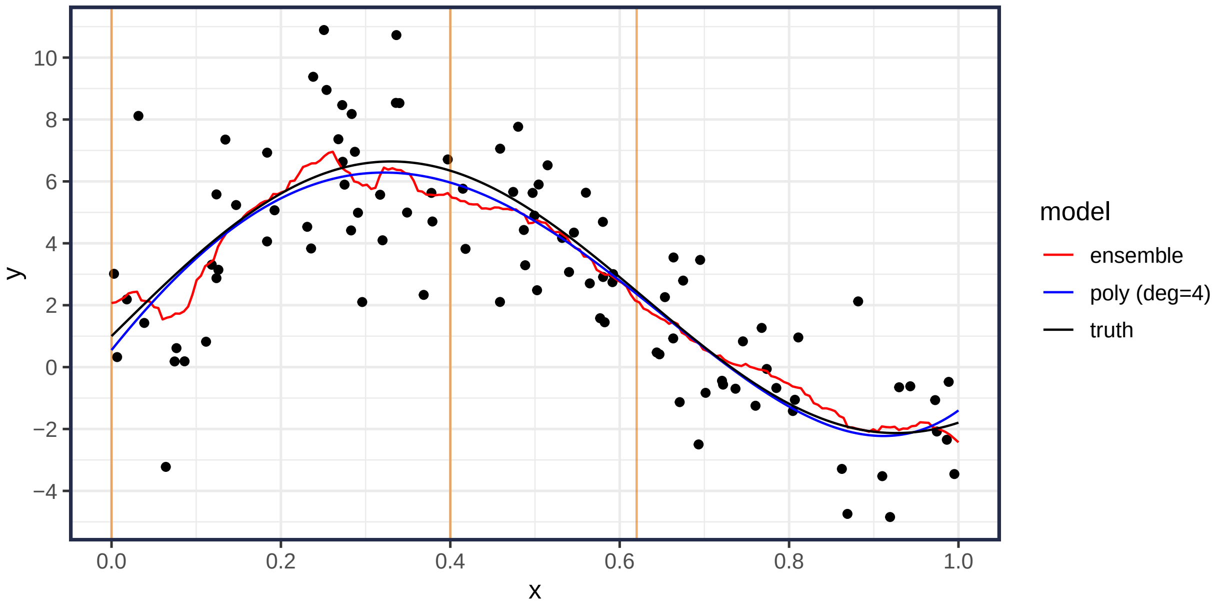
$$\text{Test MSE} = \frac{1}{M} \sum_{j=1}^{M} (y_j - \hat{f}_w(x_j))^2$$

where M is the number of test observations.

This gives a **test** MSE of 4.28, which is better than most individual models will try stacking

model	n	w	edf	mse
polynomial(deg=3)	9	0.15	4.00	4.13
polynomial(deg=5)	6	0.10	6.00	4.13
polynomial(deg=6)	1	0.02	7.00	4.21
ensemble	NA	NA	NA	4.28
nearest neighbor(k=13)	1	0.02	7.69	4.35
nearest neighbor(k=15)	1	0.02	6.67	4.37
nearest neighbor(k=20)	2	0.03	5.00	4.40
nearest neighbor(k=10)	7	0.11	10.00	4.42
nearest neighbor(k=9)	1	0.02	11.11	4.46
polynomial(deg=10)	4	0.07	11.00	4.63
polynomial(deg=11)	1	0.02	12.00	4.64
nearest neighbor(k=25)	1	0.02	4.00	4.69
polynomial(deg=12)	1	0.02	13.00	4.78
polynomial(deg=16)	1	0.02	17.00	5.03
nearest neighbor(k=5)	14	0.23	20.00	5.03
polynomial(deg=20)	1	0.02	21.00	5.14
nearest neighbor(k=4)	1	0.02	25.00	5.31
polynomial(deg=2)	7	0.11	3.00	5.34
nearest neighbor(k=2)	1	0.02	50.00	6.57
nearest neighbor(k=50)	1	0.02	2.00	7.06

this is the combined model approach



4 Bias-Variance Trade-off

This section explores the bias-variance trade-off for the examples we covered last class. This involves examining the theoretical properties of an estimator.

true reg function

4.1 Data Generating Functions

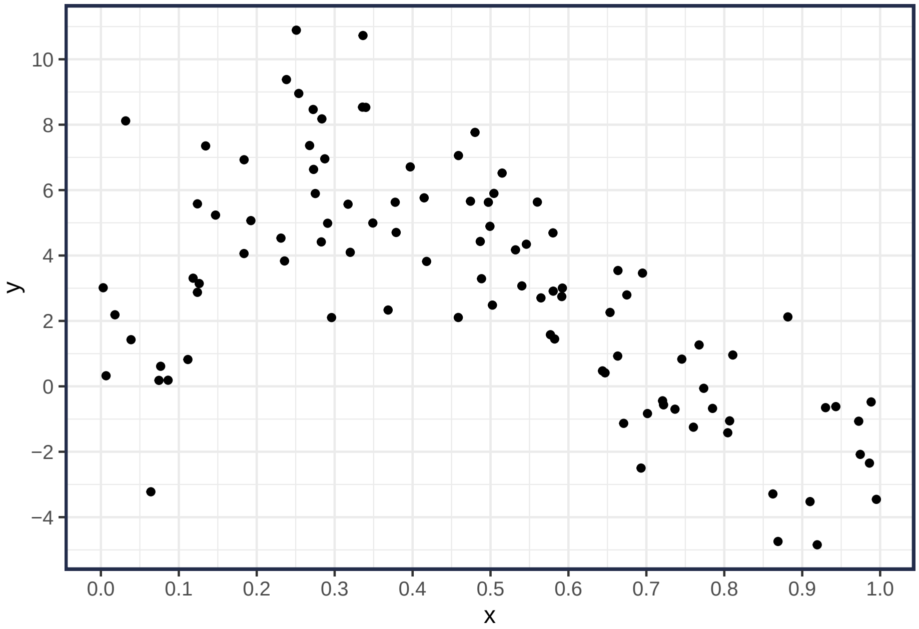
Here we set the data generation functions. $X \sim U[0,1]$ and $f(x) = 1 + 2x + 5\sin(5x)$ and $y(x) = f(x) + \epsilon$, where $\epsilon \overset{\text{iid}}{\sim} N(0,2)$.

optimal model is $\hat{y}$=f(x)= E[Y|X=x]

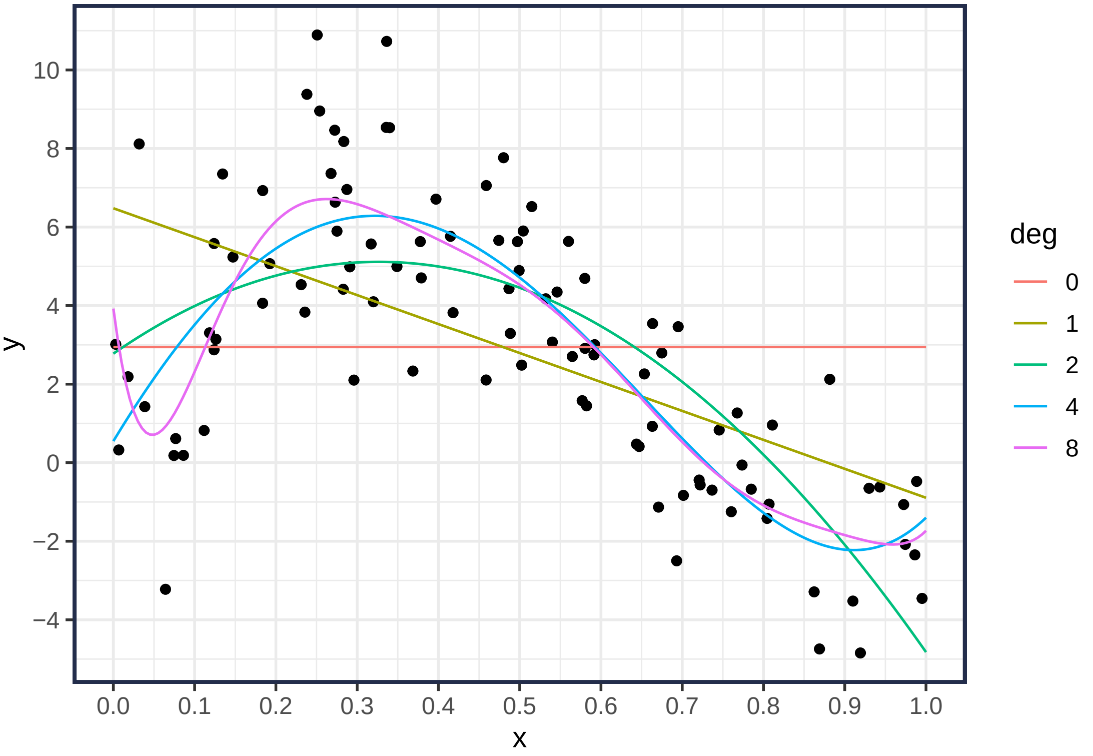
```
#- Simulation functions
sim_x <- function(n) runif(n)               # U[0,1]
f <- function(x) 1 + 2*x + 5*sin(5*x)       # true mean function
sim_y <- function(x, sd){                   # generate Y|X from N{f(x),sd}
  n = length(x)
  f(x) + rnorm(n, sd=sd)
}
#- Simulation settings
n = 100                                     # number of obs
sd = 2                                      # stdev for error
```

4.2 One Realization

Last class, we explored one realization from this system.



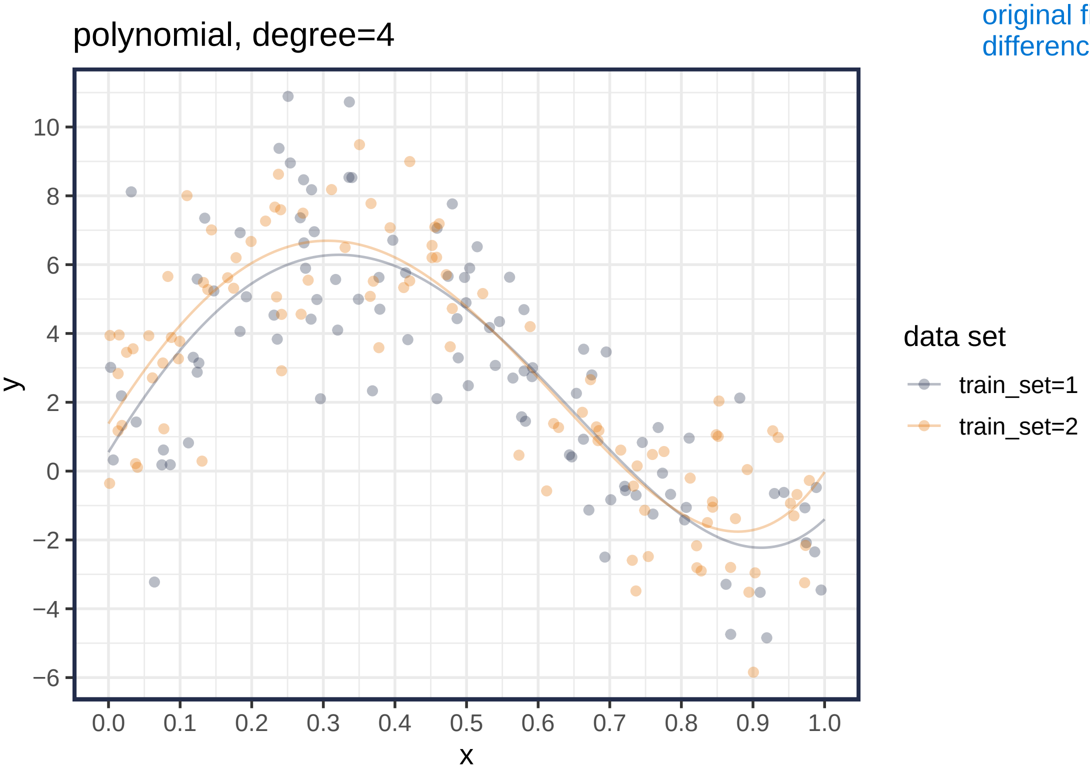
And then fit several polynomial regression models. Recall by polynomial regression I mean using a predictor function $\hat{y}(x) = f(x,d) = \sum_{j=0}^{d} x^j \beta_j$, where $d \in \{0, 1, \ldots\}$ is the degree.



4.3 A second realization

Suppose we drew another training set (using same distributions and sample size n):

generated another 100 random points, close to original fit but has some differences



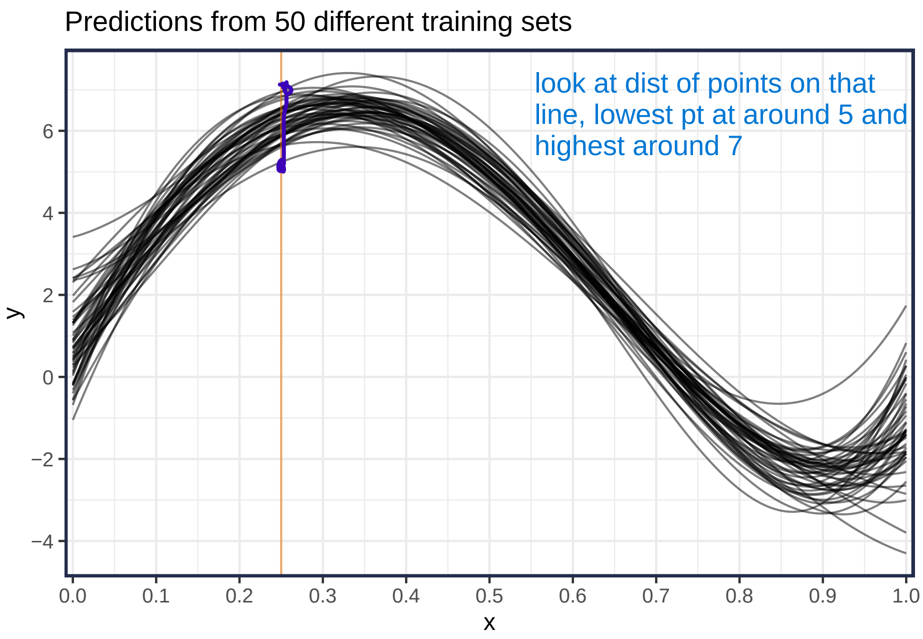
- We get another fitted curve using the new training data.
- While the two curves are visually similar, they are not identical.
- If we took more training samples, we would get more fitted curves
- What we want to study in this section is the likelihood that we will happen to get a *good* fit given a single training data set.

4.4 Bias, Variance, and Mean Squared Error (MSE)

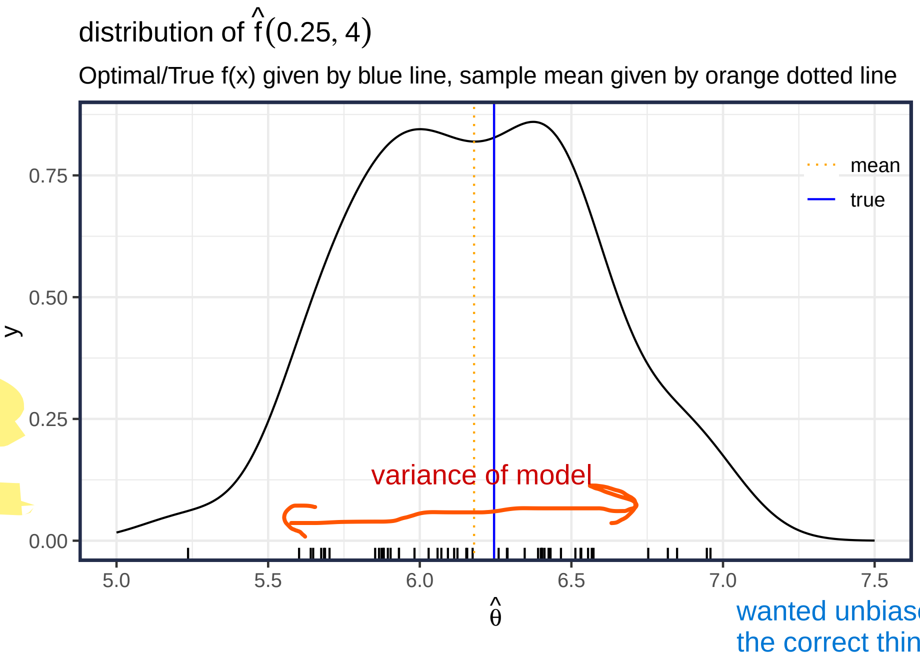
- The statistical properties of an estimator can help us understand its potential performance
- Let $D = [(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)]$ be *training data*
- Let $\hat{\theta} = \hat{\theta}(D)$ be the estimated value *calculated from the training data* D
  - E.g. $\theta = f(x)$, $\hat{\theta} = \hat{f}(x \mid D)$
  - $\hat{\theta}$ is a *random variable* because we are treating the data as random; it has a distribution.

4.4.1 Distribution of $\hat{\theta}$

- Consider the distribution of $\hat{\theta} = \hat{f}_{\text{poly}}(0.25, d = 4)$.
  - This is the distribution of the fit at $x = 0.25$ from a polynomial (of degree 4) using different *training* sets
- I generated 50 different training data sets (each with $n = 100$), fit a polynomial (deg=4) model to each data set, and recorded the estimate at $x = 0.25$.



look at dist of points on that line, lowest pt at around 5 and highest around 7



dotted line is the mean/average
the blue line is f(0.25) the actual

bias is difference between the true value and the average (expected val) from the model

plotted on the intersection points on that line

is kernal density plot
is geom_density

wanted unbiased= on average, it predicts the correct thing,
we don't care as much,

we want small MSE!!!

4.4.2 Some properties of an estimator

- **Bias** of an estimator is defined as $\mathrm{E}_D[\hat{\theta}] - \theta$
- **Variance** of an estimator is defined as $\mathrm{V}_D[\hat{\theta}] = \mathrm{E}_D[\hat{\theta}^2] - \mathrm{E}_D[\hat{\theta}]^2$
- **MSE** of an estimator is defined as $\mathrm{MSE}(\hat{\theta}) = E_D[(\hat{\theta} - \theta)^2]$

D stands for training data=random

X = R.V.

E[X^2]=V(x)+E[x]^2

note: E[X^2]:
X=thetahat-theta
expected val of constant is constant, variance of constant is 0

$$
\begin{aligned}
\mathrm{MSE}(\hat{\theta}) &= E_D[(\hat{\theta} - \theta)^2] \\
&= V_D[\hat{\theta} - \theta] + E_D[\hat{\theta} - \theta]^2 \\
&= V_D[\hat{\theta}] + E_D[\hat{\theta} - \theta]^2
\end{aligned}
$$

- Estimators are often evaluated based on MSE, being unbiased, and/or having minimum variance (out of all unbiased estimators)
- These properties are based on the *distribution of an estimate*.
  - Once we observe the training data, the resulting estimate may be great or horrible.
  - However these theoretical properties provide insight into what we can expect and how much confidence we can have in the estimates.

4.5 Estimating the Bias, Variance, and Mean Squared Error (MSE)

- Last class, we examined the Risk/EPE (e.g., MSE) *conditioning on the training data* (See Section 6.2.1)
- Now we will relax this and bring in the uncertainty in the training data D

Under a squared error loss function $L(Y, f(X)) = (Y - f(X))^2$, the *overall* EPE (or EPE before we see any training data) at a particular $X = x$ is

$$\begin{aligned}\mathrm{MSE}_x(f) &= \mathrm{E}_{DY|X}[(Y - \hat{f}_D(x))^2 \mid X = x] \\ &= \mathrm{V}[Y \mid X = x] + \mathrm{V}[\hat{f}_D(x) \mid X = x] + \Big(\mathrm{E}[\hat{f}_D(x) \mid X = x] - f(x)\Big)^2 \\ &= \text{irreducible error} + \text{model variance} + \text{model squared bias}\end{aligned}$$

where D is the training data, f is the true model, and $\hat{f}_D(x)$ is the prediction at $X = x$ estimated from the training data D.

Note

$$\begin{aligned}\mathrm{MSE}_x(f) &= \mathrm{E}_{DY|X}[(Y - \hat{f}_D(x))^2 \mid X = x] \\ &= \mathrm{E}_{DY|X}[(Y - f(x) + f(x) - \hat{f}_D(x))^2 \mid X = x] \\ &= \mathrm{E}_{DY|X}[(Y - f(x))^2] + \mathrm{E}_{DY|X}(f(x) - \hat{f}_D(x))^2 + \mathrm{E}_{DY|X}\, 2(Y - f(x))(f(x) - \hat{f}_D(x)) \\ &= V[Y \mid X = x] + \mathrm{E}_{DY|X}(f(x) - \hat{f}_D(x))^2 + 0 \\ &= V[Y \mid X = x] + \mathrm{V}_{DY|X}(\hat{f}_D(x)) + \mathrm{E}_{DY|X}(f(x) - \hat{f}_D(x))^2\end{aligned}$$

not required to derive it

- We can estimate the model variance and bias with simulation
  - Generate new data $D_m = \{(Y_i, X_i)\}_{i=1}^n$ for simulations $m = 1, 2, \dots, M$ (use the same sample size n)
  - Fit the models with data D_m getting $\hat{f}_{D_m}(\cdot)$
  - Now we can estimate the items of interest:

$$\mathrm{E}[\hat{f}_D(x)] \approx \bar{f}(x) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}_{D_m}(x)$$

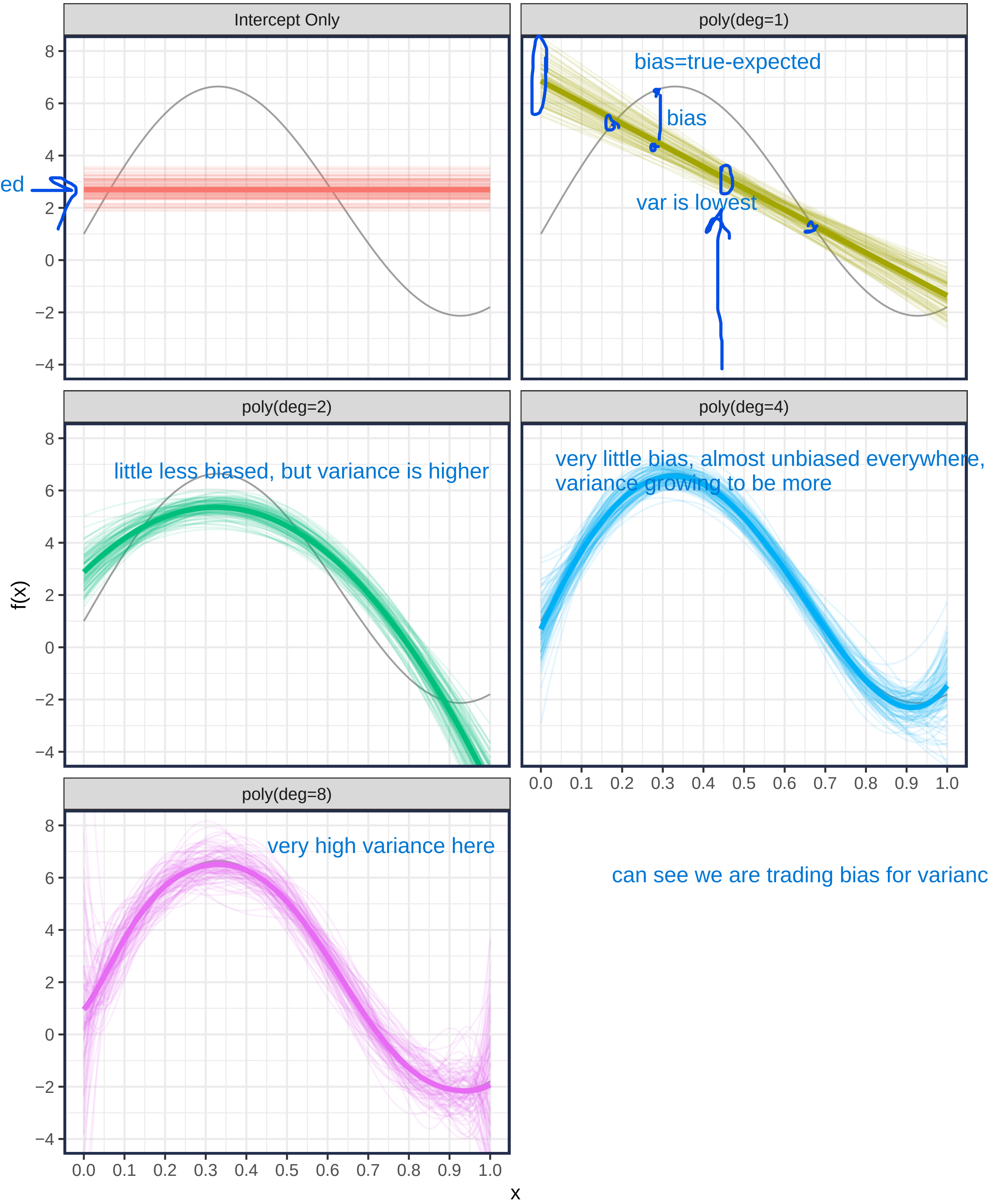
over training data=D

$$\mathrm{V}[\hat{f}_D(x)] \approx s_f^2(x) = \frac{1}{M-1}\sum_{m=1}^{M} (\hat{f}_{D_m}(x) - \bar{f}(x))^2$$

generated 2000 training datasets, there are 2000 lines per plot

4.5.1 Simulation

I ran 2000 simulations to generate $\{\hat{f}_m(x, deg = d) : d \in \{0, 1, 2, 4, 8\}, m \in \{1, 2, \ldots, 2000\}\}$.



expected

bias=true-expected

bias

var is lowest

grey is true value f(x)

little less biased, but variance is higher

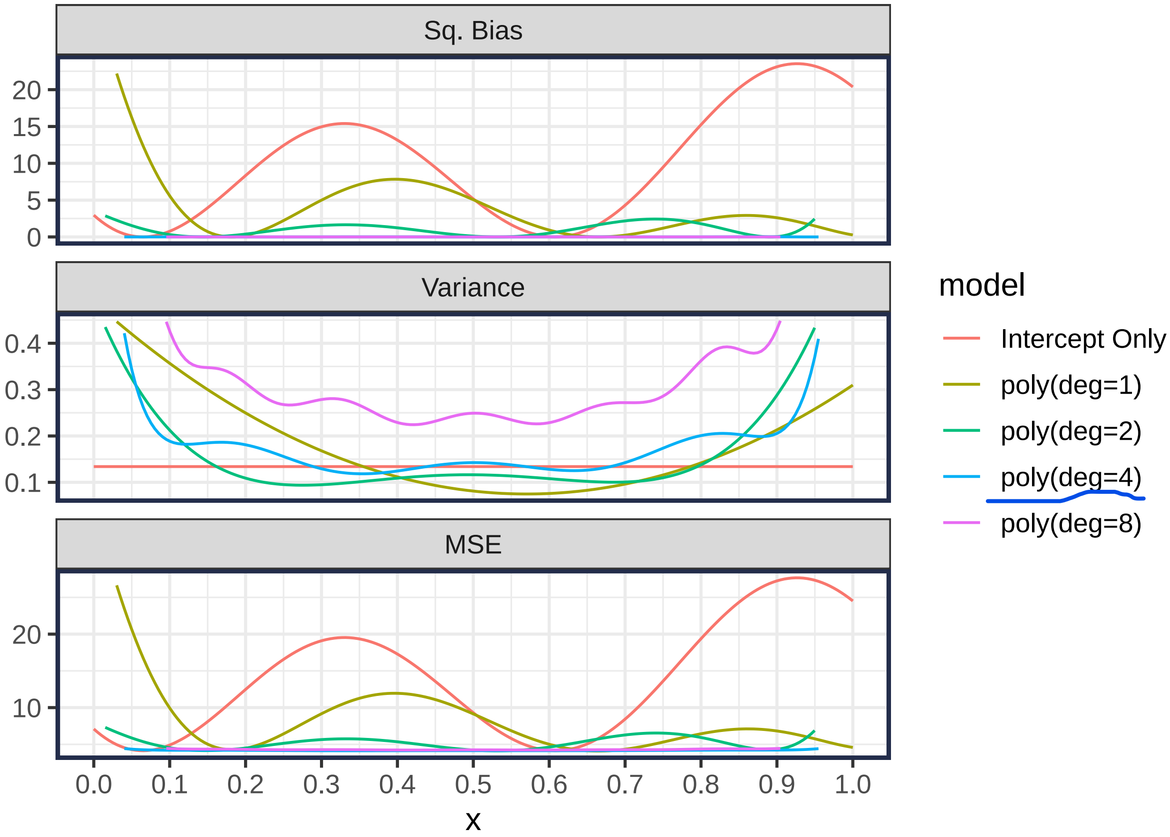
very little bias, almost unbiased everywhere, but variance growing to be more

very high variance here

can see we are trading bias for variance

4.5.2 Observations

- This shows the bias and variance of each model.
- You can see that as the flexibility (e.g., degree) of the model increases, the bias decreases but the variance (especially at the edges) increases.
  - The **bias** is the difference between the true regression function (dark gray line) and the model mean (dark colored line).
  - The **variation** is seen in the width of the transparent curves, one for each simulation.



lowest MSE, and looks to be best model theoretically as well

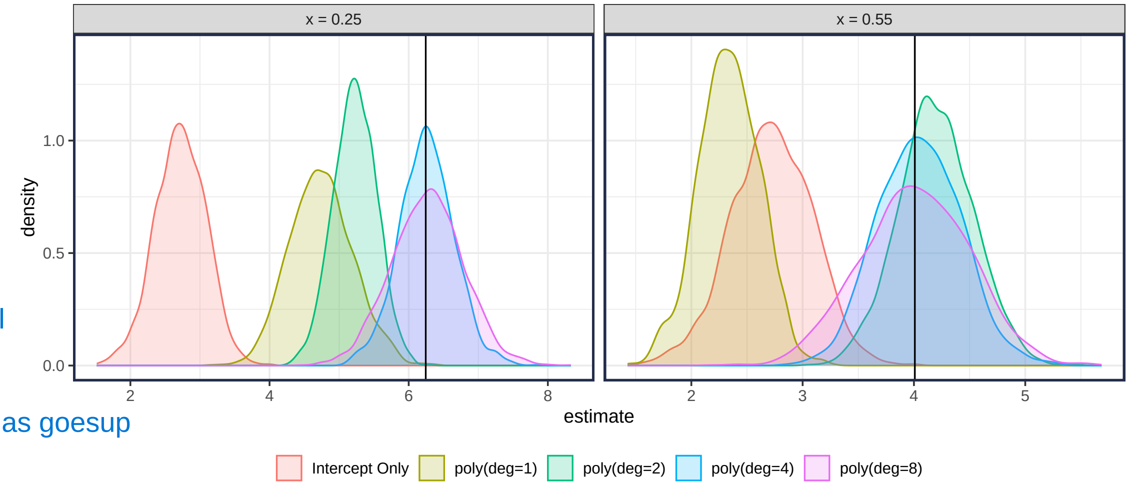
4.5.3 Bias, Variance, and MSE at a single input

- Notice that model variance and model bias vary over x.
- To help see what is going on, we now look at the distributions at $x = 0.25$ and $x = 0.55$.

bias and variance are part of feature space

more flexible model is higher the variance will be

less complex=bias goesup



slices through those graphs

4.5.4 Integrated MSE

overall

The above analysis examines the $\mathrm{MSE}_x(f)$ over a set of x's. However, in a real setting, the overall test error will be based on *all* of the actual test X values. So we are usually more interested in the *integrated* MSE:

$$\begin{aligned}\mathrm{MSE}(f) &= \mathrm{E}_{DYX}[(Y - \hat{f}_D(X))^2] \\ &= E_X[\mathrm{MSE}_X(f)] \\ &= \int \mathrm{MSE}_x(f)\,\Pr(dx)\end{aligned}$$

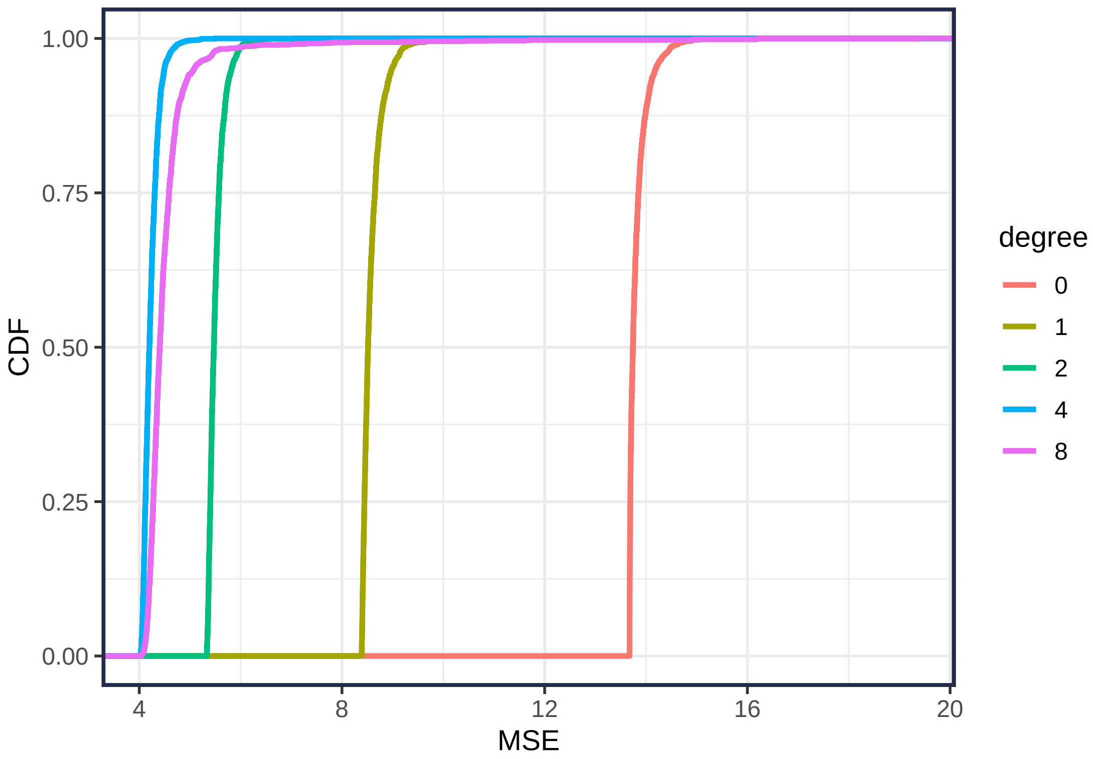
deg	bias.sq	var	mse
0	9.68	0.13	13.81
1	4.39	0.19	8.58
2	1.33	0.18	5.51
4	0.01	0.22	4.23
8	0.00	0.53	4.53

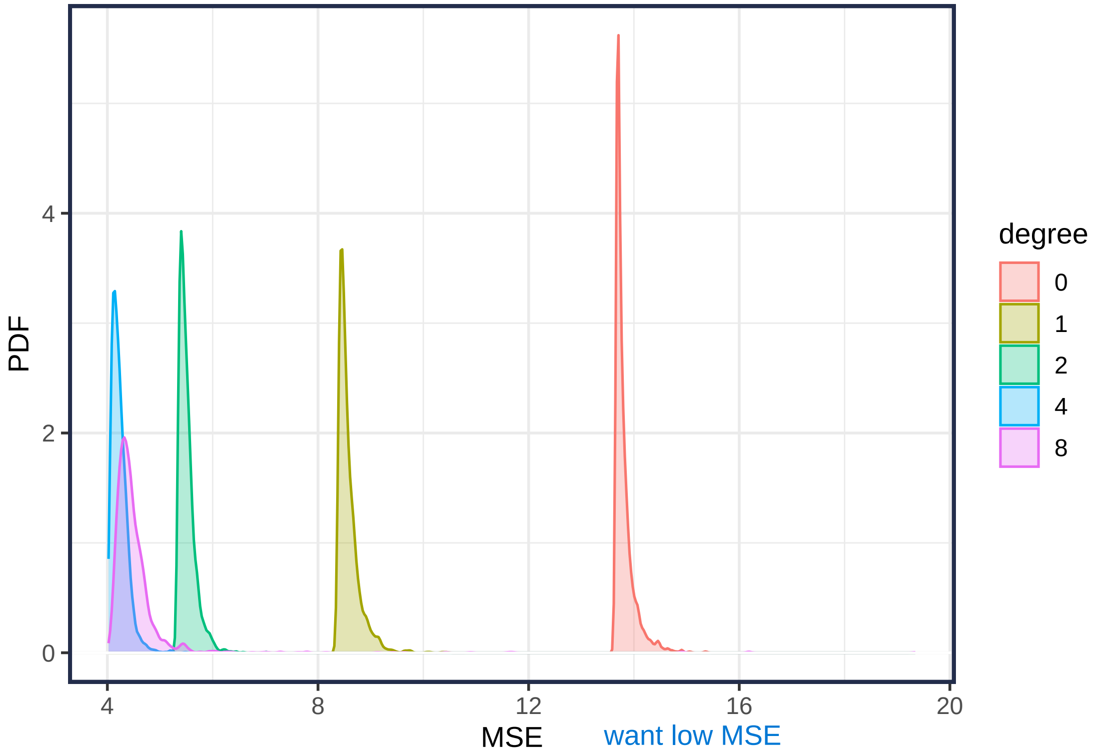
4.6 What does it all mean

The main point is that we desire to find a predictive model that has just the right *flexibility*. This will depend on both the true flexibility of the data as well as the sample size. A model that is too complex will have low bias, but potentially high variance. A model that is not complex enough will have high bias, but lower variance.

In a real setting, we will only observe one training data (a single curve) and will have to decide the optimal flexibility. The following plot shows the estimated distribution of MSE values.

more data= can have a more complex model distribution of mse





model complexity

want low MSE

While its possible that we could just happen to get a particular training data realization that favors a model other than the globally optimal model, this is unlikely for the bad models. However, it is not uncommon for "close" models.

Below is a table of the number of simulations that each model had the best MSE:

degree	n
0	0
1	0
2	0
3	397
4	1006
5	486
6	75
7	25
8	4
9	6
10	1

4th degree poly came out best the most often out of 2000 times

- While the polynomial of degree=4 is most often best model (under the squared error loss), under some training data sets, the other degree models predict better.
- **Conclusion 1**: In our toy example, a polynomial with degree=4 is the best model, in principal. However, for some data (i.e., some training data sets) the models with degree>4 and degree<4 would predict better.
    - Why do more higher degree models (degree > 4) predict better than lower degree models (degree < 4)?
- **Conclusion 2**: The above analysis is what is meant by the "bias-variance trade-off".
    - In reality, we only get to observe one realization of the training data so we can never actually estimate the bias and variance the way we did above
    - But we can still estimate the Risk (e.g., MSE) by using resampling methods like cross-validation or statistical methods like BIC.
    - More loosely, when people mention bias-variance trade-off they are referring to the principal that the best model is one that has just the right flexibility.
    - If the model is too complex, it is unlikely to produce a good estimate (across the entire range of inputs) because it is likely to stray far from the expected mean at certain values.
    - If the model is not complex enough, then it will not track with the expected mean across the range of input values and thus produce poor overall performance.
- **Conclusion 3**: Performance of a model can vary across the input features X.
    - If you are only concerned about performance in a specific range of X, then emphasize these during training (e.g., weight observations close to X more heavily during model estimation).
    - If the test X values are coming from a different distribution than the training X values, then your model may not be optimal.

predictive ability can be different across feature space

5 Appendix: R Code

HELPFUL FOR HW!!!!!

5.1 Required R Packages

We will be using the R packages of:

- `FNN` for k nearest neighbor models
- `tidyverse` for data manipulation and visualization
- `tidymodels` (for optional tidymodels approach)

```r
library(FNN)
library(tidyverse)
library(tidymodels)
```

5.2 Training and Test Data

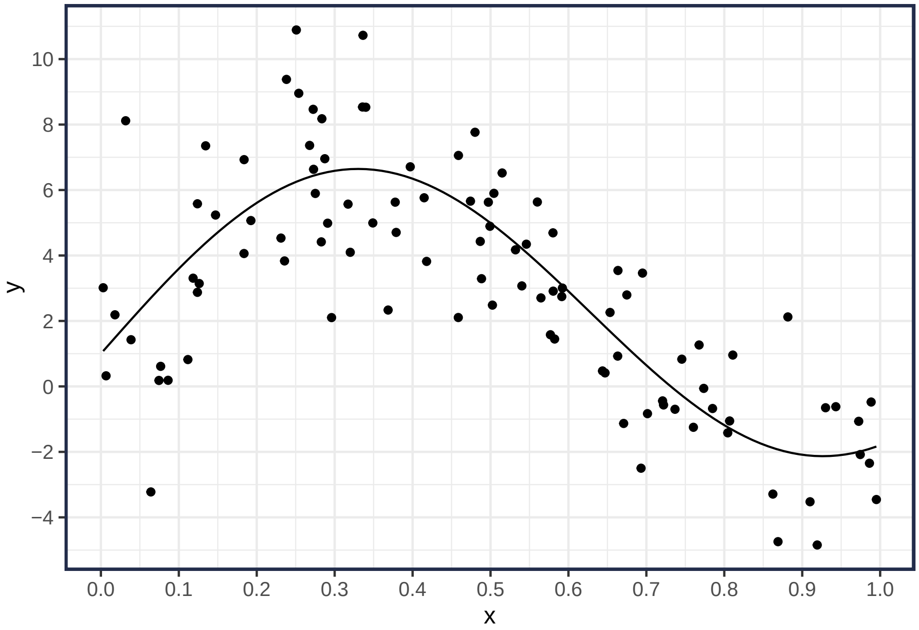
```r
#-- Functions
sim_x <- function(n) runif(n)             # U[0,1]
f <- function(x) 1 + 2*x + 5*sin(5*x)     # true mean function
sim_y <- function(x, sd){                 # generate Y|X from N{f(x),sd}
  n = length(x)
  f(x) + rnorm(n, sd=sd)
}

#-- Settings
n = 100                                   # number of observations
sd = 2                                    # stdev for error

#-- Generate Training Data
set.seed(825)                             # set seed for reproducibility
x = sim_x(n)                              # get x values
y = sim_y(x, sd=sd)                       # get y values
data_train = tibble(x, y)                 # training data tibble

#-- Generate Test Data
ntest = 50000                             # Number of test samples
set.seed(235)                             # set *different* seed
xtest = sim_x(ntest)                      # generate test X's
ytest = sim_y(xtest, sd=sd)               # generate test Y's
data_test = tibble(x=xtest, y=ytest)      # test data
```

```r
#-- plot points and true regression function
ggplot(data_train, aes(x,y)) +
  geom_point() +
  geom_function(fun = f) +
  scale_x_continuous(breaks=seq(0, 1, by=.10)) +
  scale_y_continuous(breaks=seq(-6, 10, by=2))
```



5.3 Evaluating Polynomial Models

We want to evaluate several polynomial models. For each tuning parameter (i.e., polynomial degree), we need to fit the model on the training data, make predictions on the test data, and score those predictions. Since we are going to repeat these steps for multiple tuning parameters, its a good time to write a function!

```
#: Function to fit, predict, and evaluate a polynomial regression model
poly_eval <- function(deg, data_train, data_test){
  #: fit model with training data
  if(deg==0) m = lm(y~1, data=data_train)   # intercept only model
  else m = lm(y~poly(x, degree=deg), data=data_train) # polynomial
  p = length(coef(m))                       # number of parameters
  #: calculate training MSE
  mse_train = mean(m$residuals^2)           # training MSE
  #: calculate test MSE
  yhat = predict(m, data_test)              # predictions at test X's
  mse_test = mean( (data_test$y - yhat)^2 )# test MSE
  #: output a data frame of relevant info
  tibble(degree=deg, edf=p, mse_train, mse_test)
}
```

Let’s try it out

```
poly_eval(deg = 2, data_train, data_test)
```

```
#> # A tibble: 1 x 4
#>   degree   edf mse_train mse_test
#>    <dbl> <int>     <dbl>    <dbl>
#> 1      2     3      5.58     5.34
```

Looks good! Now we can iterate over a degree sequence.

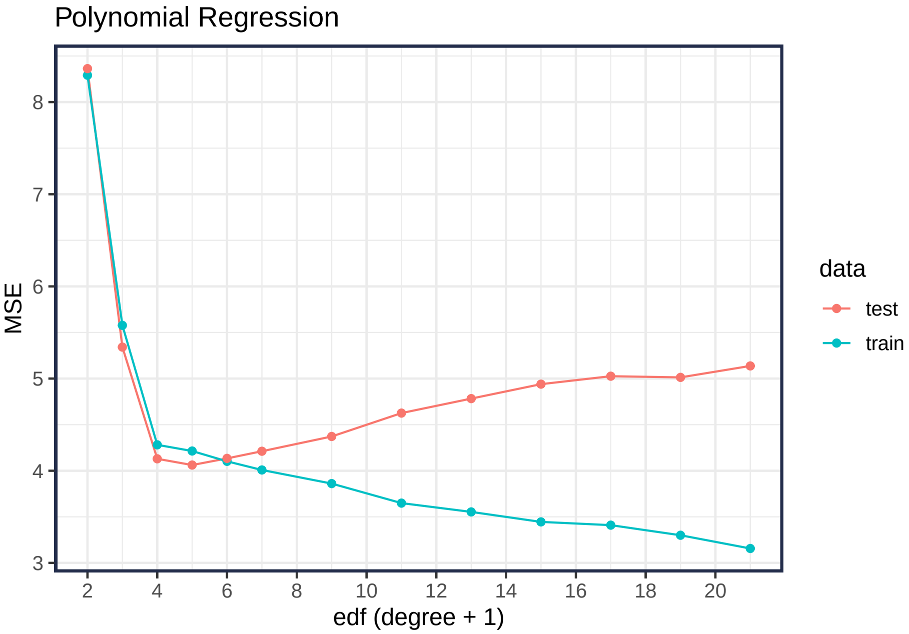
```
#: degree sequence
deg_seq = c(1:6, seq(8, 20, by=2))    # sequence of Degrees

# Using purrr:map_df()
perf_poly = map_df(deg_seq, ~poly_eval(., data_train=data_train, data_test=data_test))
```

```
# Using a loop
perf_poly = vector("list", length(deg_seq))
for(deg in deg_seq){
  tmp = poly_eval(deg, data_train=data_train, data_test=data_test)
  perf_poly = bind_rows(perf_poly, tmp)
}
```

Now for the plots

```
perf_poly %>%
  # make long:
  pivot_longer(starts_with("mse"), names_to="data", values_to="mse") %>%
  # remove 'mse_' from values:
  mutate(data = str_remove(data, "mse_")) %>%
  # plot:
  ggplot(aes(edf, mse, color=data)) + geom_line() + geom_point() +
  labs(title="Polynomial Regression", x = 'edf (degree + 1)', y="MSE") +
  scale_x_continuous(breaks=seq(2, 30, by=2))
```



And we can find the best polynomial degree for our test data

```
perf_poly %>% slice_min(mse_test)
```

```
#> # A tibble: 1 x 4
#>   degree   edf mse_train mse_test
#>    <dbl> <int>     <dbl>    <dbl>
#> 1      4     5      4.21     4.06
```

5.3.1 Tidymodels

We can repeat the process with tidymodels. The tidymodels approach has a nice helper function called `tune_grid()` that lets us fit and evaluate models in one function.

One small nuisance step. We have to create a `rset` object that contains the information about the training and testing sets. In the future we will use functions like `vfold_cv()` instead of the manual process we need to do now.

```
library(tidymodels)

#: manually create an `rset` object
# Note: normally we would use initial_split() to make train/test split.
#       However since we are doing things a bit differently (i.e., we can
#       generate as much evaluation data as we want), we have to force things
#       a bit.
#       Long answer short: you probably won't have to do this step for any
#       real analysis.
splits = make_splits(data_train, data_test) %>%
  list %>%
  manual_rset("train/test")
```

Next we set up a *workflow* for polynomial regression with tunable degree. Notice it the `step_poly()` function that `degree = tune()` which indicates the degree needs to be specified.

```
poly_wf = workflow(
  preprocessor = recipe(y~x, data = data_train) %>%
    step_poly(x, degree = tune()), # note the tune() for degree
  spec = linear_reg()
)
```

Now we will use the `tune_grid()` function. All tunable parameters (i.e., those indicated by tune()) need to be supplied in the `grid` = argument. In this case, it is `degree`.

```
poly_tune = tune_grid(
  #: specify workflow
  object = poly_wf,

  #: specify the grid for all parameters that have tune().
  grid = expand_grid(degree = c(1:6, seq(8, 20, by=2))),

  #: pass in the `rset` object
  resamples = splits,

  #: specify the metrics to calculate. The package doesn't give use MSE,
  #  so we'll use RMSE and later take the squared value.
  metrics = metric_set(rmse)
)
```

The output doesn't look very informative at this point.

```
poly_tune
```

```
#> # A tibble: 1 x 4
#>   splits                id         .metrics           .notes
#>   <list>                <chr>      <list>             <list>
#> 1 <split [100/50000]> train/test <tibble [13 x 5]> <tibble [0 x 3]>
```

Notice that we have a tibble with *list columns*. We can extract the relevant information using `collect_metrics()`.

```
poly_tune %>%
  collect_metrics(summarize = FALSE)
```

```
#> # A tibble: 13 x 6
#>   id         degree .metric .estimator .estimate .config
#>   <chr>       <dbl> <chr>   <chr>          <dbl> <chr>
#> 1 train/test      1 rmse    standard        2.89 Preprocessor01_Model1
#> 2 train/test      2 rmse    standard        2.31 Preprocessor02_Model1
#> 3 train/test      3 rmse    standard        2.03 Preprocessor03_Model1
#> 4 train/test      4 rmse    standard        2.02 Preprocessor04_Model1
#> 5 train/test      5 rmse    standard        2.03 Preprocessor05_Model1
#> 6 train/test      6 rmse    standard        2.05 Preprocessor06_Model1
#> # i 7 more rows
```

The relevant columns are `degree` and `.estimate`. But remember the function is returning RMSE, but we want our results in MSE = RMSEˆ2.

```
poly_tune %>%
  collect_metrics(summarize = FALSE) %>%
  mutate(
    mse = .estimate^2   # recall the estimate is RMSE
  ) %>%
  select(degree, mse)
```

```
#> # A tibble: 13 x 2
#>   degree   mse
#>    <dbl> <dbl>
#> 1      1  8.36
#> 2      2  5.34
#> 3      3  4.13
#> 4      4  4.06
#> 5      5  4.13
#> 6      6  4.21
#> # i 7 more rows
```

5.4 Evaluating k nearest neighbor models

Here is a function that fits a knn model on the training data for a particular k, makes predictions on the test data, and evaluates the MSE.

```
#: Function to evaluate kNN
knn_eval <- function(k, data_train, data_test){
  # fit model and eval on training data
  knn = knn.reg(data_train[,'x', drop=FALSE],
                y = data_train$y,
                test = data_train[,'x', drop=FALSE],
                k = k)
  r = data_train$y-knn$pred        # residuals on training data
  mse_train = mean(r^2)            # training MSE

  # fit model and eval on test data
  knn.test = knn.reg(data_train[,'x', drop=FALSE],
                     y = data_train$y,
                     test = data_test[,'x', drop=FALSE],
                     k = k)
  r_test = data_test$y - knn.test$pred # residuals on test data
  mse_test = mean(r_test^2)            # test MSE
  # results
  edf = nrow(data_train)/k         # effective dof (edof)
  tibble(k = k, edf = edf, mse_train, mse_test)
}
```

Let's try it out

```
knn_eval(k = 10, data_train, data_test)
```

```
#> # A tibble: 1 x 4
#>       k   edf mse_train mse_test
#>   <dbl> <dbl>     <dbl>    <dbl>
#> 1    10    10      3.86     4.42
```

Set up a grid of neighborhood size (k).

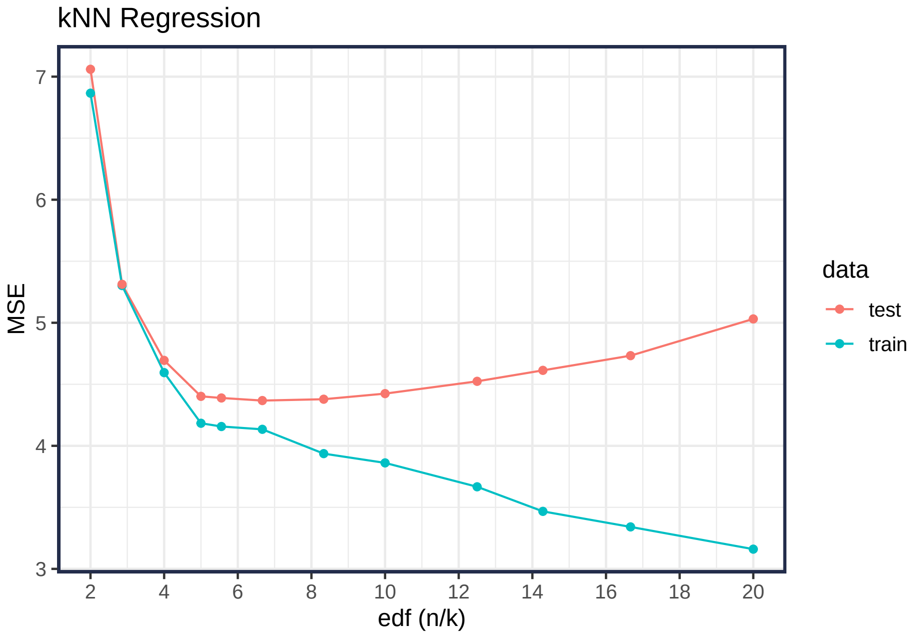
```
K = c(50, 35, 25, 20, 18, 15, 12, 10, 8, 7, 6, 5)
```

And let it run

```
# Using purrr:map_df()
perf_knn = map_df(K, knn_eval, data_train=data_train, data_test=data_test)
```

```
# Alternatively, using a loop
perf_knn = tibble()
for(k in K){
  tmp = knn_eval(k, data_train=data_train, data_test=data_test)
  perf_knn = bind_rows(perf_knn, tmp)
}
```

```
#: Plot Results
perf_knn %>%
  # make long:
  pivot_longer(starts_with("mse"), names_to="data", values_to="mse") %>%
  # remove 'mse_' from values:
  mutate(data = str_remove(data, "mse_")) %>%
  # plot:
  ggplot(aes(edf, mse, color=data)) + geom_line() + geom_point() +
  labs(title="kNN Regression", x='edf (n/k)', y="MSE") +
  scale_x_continuous(breaks=seq(2, 30, by=2))
```



And we can find the best neighborhood size for our test data

```
#: best knn
perf_knn %>% slice_min(mse_test)
```

```
#> # A tibble: 1 x 4
#>       k   edf mse_train mse_test
#>   <dbl> <dbl>     <dbl>    <dbl>
#> 1    15  6.67      4.13     4.37
```

5.4.1 Tidymodels

We can slightly modify the `tune_grid()` we used for polynomials.

First we will specify the knn workflow. In this workflow, the model specification has the tunable parameter `neighbors`.

```
knn_wf = workflow(
  preprocessor = recipe(y~x, data = data_train),
  spec = nearest_neighbor(neighbors = tune(),  # tune this parameter
                          mode = "regression", # regression / classification
                          weight_func = "rect" # using unweighted knn
                          )
)
```

```
knn_tune = tune_grid(

  #: specify model
  object = knn_wf,

  #: specify the grid for all parameters that have tune().
  grid = expand_grid(neighbors = c(50, 35, 25, 20, 18, 15, 12, 10, 8, 7, 6, 5)),

  #: pass in the `rset` object
  resamples = splits,

  #: specify the metrics to calculate. The package doesn't give use MSE,
  #  so we'll use RMSE and later take the squared value.
  metrics = metric_set(rmse)
)
```

The output doesn't look very informative at this point.

```
knn_tune
```

```
#> # A tibble: 1 x 4
#>   splits               id         .metrics          .notes
#>   <list>               <chr>      <list>            <list>
#> 1 <split [100/50000]> train/test <tibble [12 x 5]> <tibble [0 x 3]>
```

Notice that we have a tibble with *list columns*. We can extract the relevant information using `collect_metrics()`.

```
knn_tune %>%
  collect_metrics(summarize = FALSE)
```

```
#> # A tibble: 12 x 6
#>   id         neighbors .metric .estimator .estimate .config
#>   <chr>          <dbl> <chr>   <chr>          <dbl> <chr>
#> 1 train/test         5 rmse    standard        2.24 Preprocessor1_Model01
#> 2 train/test         6 rmse    standard        2.18 Preprocessor1_Model02
#> 3 train/test         7 rmse    standard        2.15 Preprocessor1_Model03
#> 4 train/test         8 rmse    standard        2.13 Preprocessor1_Model04
#> 5 train/test        10 rmse    standard        2.10 Preprocessor1_Model05
#> 6 train/test        12 rmse    standard        2.09 Preprocessor1_Model06
#> # i 6 more rows
```

Extract relevant columns and transform RMSE to MSE.

```
knn_tune %>%
  collect_metrics(summarize = FALSE) %>%
  mutate(
    mse = .estimate^2   # recall the estimate is RMSE
  ) %>%
  select(neighbors, mse)
```

```
#> # A tibble: 12 x 2
#>   neighbors   mse
#>       <dbl> <dbl>
#> 1         5  5.03
#> 2         6  4.73
#> 3         7  4.61
#> 4         8  4.52
#> 5        10  4.42
#> 6        12  4.38
#> # i 6 more rows
```

5.5 Comparison of best models

The optimal models (i.e., the model with optimal tuning parameters) have about the same edf.

```
(best_poly = perf_poly %>% slice_min(mse_test))
```

```
#> # A tibble: 1 x 4
#>   degree   edf mse_train mse_test
#>    <dbl> <int>     <dbl>    <dbl>
#> 1      4     5      4.21     4.06
```

```
(best_knn = perf_knn %>% slice_min(mse_test))
```

```
#> # A tibble: 1 x 4
#>       k   edf mse_train mse_test
#>   <dbl> <dbl>     <dbl>    <dbl>
#> 1    15  6.67      4.13     4.37
```

Let's see how the resulting predictions appear. We will re-fit each model on the training data and make predictions on a grid.

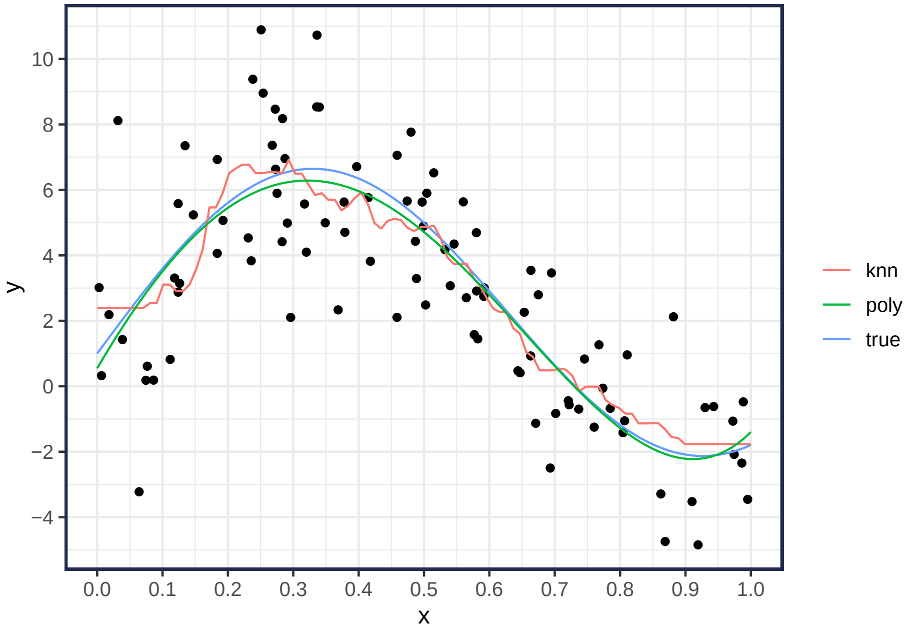
```
data_grid = tibble(x = seq(0, 1, length = 100))
```

```
#: optimal polynomial model
opt_poly = lm(y~poly(x, degree=best_poly$degree), data=data_train) # polynomial
yhat_poly = predict(opt_poly, data_grid)              # predictions at test X's

#: optimal knn
opt_knn = FNN::knn.reg(select(data_train, x),
                       y = data_train$y,
                       k = best_knn$k,
                       test = data_grid)
yhat_knn = opt_knn$pred
```

```
preds = data_grid %>%
  mutate(
    poly = yhat_poly,
    knn = yhat_knn
  ) %>%
  pivot_longer(cols = c(poly, knn), names_to = "model", values_to = ".pred")
```

```
ggplot(data_train, aes(x,y)) +
  geom_point() +
  geom_function(fun = f, aes(color = "true")) +
  geom_line(data = preds, aes(x = x, y = .pred, color=model)) +
  scale_x_continuous(breaks=seq(0, 1, by=.10)) +
  scale_y_continuous(breaks=seq(-6, 10, by=2)) +
  labs(color = "")
```



5.5.1 Tidymodels

Use `show_best()` to see the best models.

```
show_best(poly_tune, metric = "rmse")
```

```
#> # A tibble: 5 x 7
#>   degree .metric .estimator  mean     n std_err .config
#>    <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
#> 1      4 rmse    standard    2.02     1      NA Preprocessor04_Model1
#> 2      3 rmse    standard    2.03     1      NA Preprocessor03_Model1
#> 3      5 rmse    standard    2.03     1      NA Preprocessor05_Model1
#> 4      6 rmse    standard    2.05     1      NA Preprocessor06_Model1
#> 5      8 rmse    standard    2.09     1      NA Preprocessor07_Model1
```

```
show_best(knn_tune, metric = "rmse")
```

```
#> # A tibble: 5 x 7
#>   neighbors .metric .estimator  mean     n std_err .config
#>       <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
#> 1        15 rmse    standard    2.09     1      NA Preprocessor1_Model07
#> 2        12 rmse    standard    2.09     1      NA Preprocessor1_Model06
#> 3        18 rmse    standard    2.09     1      NA Preprocessor1_Model08
#> 4        20 rmse    standard    2.10     1      NA Preprocessor1_Model09
#> 5        10 rmse    standard    2.10     1      NA Preprocessor1_Model05
```

Use `finalize_workflow()` and `select_best()` to make the final models.

```
poly_wf %>%
  finalize_workflow(
    parameters = select_best(poly_tune, metric = "rmse")
  ) %>%
  fit(data = data_train) %>%
  predict(data_grid)
```

```
#> # A tibble: 100 x 1
#>    .pred
#>    <dbl>
#> 1 0.549
#> 2 0.890
#> 3 1.22
#> 4 1.54
#> 5 1.86
#> 6 2.16
#> # i 94 more rows
```

```
knn_wf %>%
  finalize_workflow(
    parameters = select_best(knn_tune, metric = "rmse")
  ) %>%
  fit(data = data_train) %>%
  predict(data_grid)
```

```
#> # A tibble: 100 x 1
#>    .pred
#>    <dbl>
#> 1   2.39
#> 2   2.39
#> 3   2.39
#> 4   2.39
#> 5   2.39
#> 6   2.39
#> # i 94 more rows
```