Supervised Learning (Part I)

EN5422/EV4238 | Fall 2023
w02_supervised_1.pdf
(Week 2 – 1/2)

Contents

1 Supervised Learning Intro

1.1 Survey & Quizzes

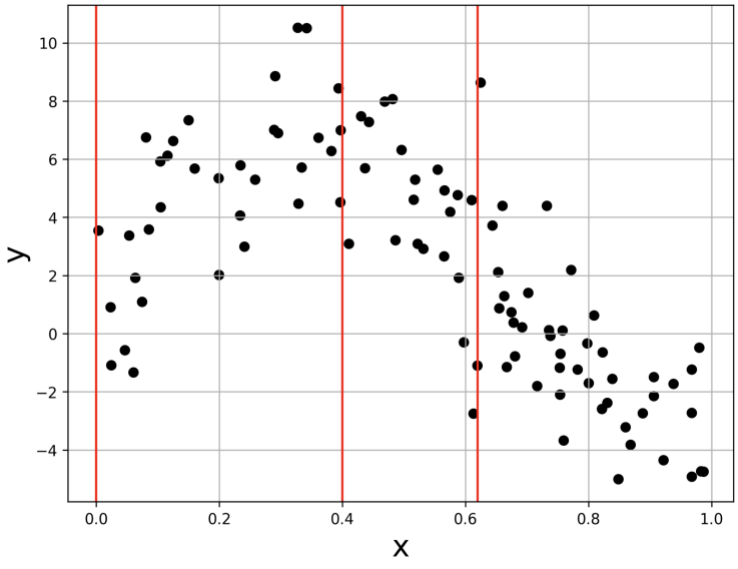
During the lecture, there will be a survey and quizzes. You can access the survey on Slido.com using the # code provided during the lecture.

1.2 Supervised Learning

- In *supervised* learning, each observation can be partitioned into two sets: the predictor variables and the outcome variable(s).
    - Predictor variables are sometimes called 1) independent, 2) feature, and 3) predictor variables.
    - Outcome variables are sometimes called 1) target, 2) labels, 3) response, 4) dependent, and 5) reference variables.
- Usually, the predictor variables are represented by X and the response variables represented by Y.
- The goal in supervised learning is to find the patterns and relationships between the predictors, X, and the response, Y.
    - Usually, the goal is to predict the value of Y given X.
- Later in the course, we will explore the *unsupervised learning* topics of association analysis, density estimation, and clustering which do not have any outcomes (i.e., no Y's)

2 Example Data

consider some data $D = \{(X_i, Y_i)\}_{i=1}^{n}$ with $Y_i \in \mathbb{R}$, $X_i \in [0,1]$ and $n = 100$.



Your Turn #1

The goal is to predict new *Y* values if we are given the *X*'s.

- If $x = 0.4$, predict *Y*.
- If $x = 0$, predict *Y*.
- If $x = 0.62$, predict *Y*.
- How should we build a model that will automatically predict *Y* for any given *X*?

3 Linear Models

- Linear models refer to a class of models where the output (predicted value) is a linear combination (weighted sum) of the input variables:

$$f(x;\beta) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

where $x = [x_1, \dots, x_p]^T$ is a vector of features/variables/attributions and $\hat{Y}|x = f(x;\ \hat{\beta})$ is the precited response at $X = x$.

- The coefficients (or weights), $\hat{\beta}$ are often selected by minimizing the squared residuals of the *training data* (may also be described as *ordinary least squares; OLS*)
  - But, there are other, and better, ways to estimate the parameters in linear regression that we will discuss later in the course. (e.g., Lasso, Ridge, Robust)

3.1 Simple Linear Regression

- single predictor variable $x \in \mathbb{R}$
- $f(x;\beta) = \beta_0 + \beta_1 x$
- Use *training data*: $D_{train} = \{(x_i, y_i)\}_{i=1}^{n}$
- OLS uses the weights/coefficients that minimize the residual sum of squares (RSS) loss function over the *training data*:

$$\hat{\beta} = arg\min_{\beta} SEE(\beta)$$

where SSE is the sum of squared errors (also known as RSS):

$$SEE(\beta) = \sum_{i}^{n} \left(y_i - f(x_i;\ \beta)\right)^2 = \sum_{i}^{n} (y_i - \beta_1 x_i)^2 = \sum_{i}^{n} \hat{\epsilon}_i^2$$

where, $\hat{\epsilon} = y_i - \hat{y}_i$ is the residual.

- The solutions are:

$$\hat{\beta}_0 = \bar{y} - \beta\bar{x}$$

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

- Definitions:

$$\begin{aligned}\mathrm{MSE}(\beta) &= \frac{1}{n}\mathrm{SSE}(\beta) \\ &= \frac{1}{n}\sum_{i=1}^{n}\big(y_i - f(x_i;\beta)\big)^2\end{aligned}$$

$$\mathrm{RMSE} = \sqrt{\mathrm{MSE}} = \sqrt{\mathrm{SSE}}/\sqrt{n}$$

3.2 OLS Linear Models in Python

3.2.1 Estimation with `statsmodels`

In **Python**, the library `statsmodels` fits an OLS linear model

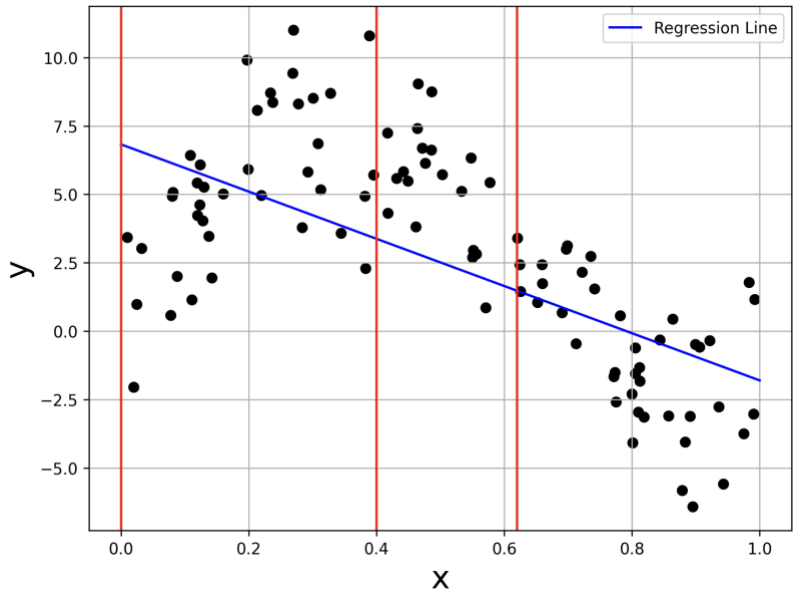
```
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

x = x.flatten()
y = y.flatten()

# Create a DataFrame
data_train = pd.DataFrame({'x': x, 'y': y})

# Fit the OLS model
model = smf.ols('y ~ x', data=data_train).fit()

# Print the summary
print(model.summary())

# Coefficients in DataFrame format
print(model.params)

# Other model properties
print("R-squared:", model.rsquared)
print("Adjusted R-squared:", model.rsquared_adj)
print("Standard error:", model.mse_resid ** 0.5)
print("F-statistic:", model.fvalue)
print("p-value of F-statistic:", model.f_pvalue)
print("AIC:", model.aic)
print("BIC:", model.bic)
print("Degrees of freedom of residuals:", model.df_resid)
print("Number of observations:", model.nobs)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.394
Model:                            OLS   Adj. R-squared:                  0.388
Method:                 Least Squares   F-statistic:                     63.68
Date:                Sun, 20 Aug 2023   Prob (F-statistic):           2.80e-12
Time:                        08:20:29   Log-Likelihood:                -252.22
No. Observations:                 100   AIC:                             508.4
Df Residuals:                      98   BIC:                             513.6
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      6.8279      0.652     10.470      0.000       5.534       8.122
x             -8.6249      1.081     -7.980      0.000     -10.770      -6.480
==============================================================================
Omnibus:                        0.194   Durbin-Watson:                   1.783
Prob(Omnibus):                  0.908   Jarque-Bera (JB):                0.051
Skew:                          -0.054   Prob(JB):                        0.975
Kurtosis:                       3.023   Cond. No.                         4.63
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
Intercept    6.827862
x           -8.624911
dtype: float64
R-squared: 0.3938577650885271
Adjusted R-squared: 0.3876726402424917
Standard error: 3.044531794318476
F-statistic: 63.678223947408824
p-value of F-statistic: 2.8006122101855e-12
AIC: 508.4368613513172
BIC: 513.6472017232934
Degrees of freedom of residuals: 98.0
Number of observations: 100.0
```

- `statsmodels` has an `ols` function, which includes the intercept by default.
    - Some examples of using formulas as well as getting the underlying X (model/design matrix) can be found here.

3.2.2 Prediction with `statsmodels`

```
import numpy as np
import pandas as pd

# Add a constant term (for intercept) to x data
X = sm.add_constant(x)

# Fit the OLS model
model = sm.OLS(y, X)
result = model.fit()

# Create a prediction dataset
xseq = np.linspace(0, 1, 200)  # sequence of 200 equally spaced values
between 0 and 1
xeval = sm.add_constant(xseq)

# Predict using the fitted model
yhat1 = result.predict(xeval)
```



3.2.3 Questions

Your Turn #2
1. How did we do? If X_{new} is close to 0, or close to 0.4, or close to 0.62? 2. How to make it better?

4 Polynomial inputs

- In the *simple* linear regression model, we had 2 parameters that we needed to estimate, β_0 and β_1. Thus, the model complexity is minimal.
  - The only thin simpler is an intercept only model.
- But the data appears to have a more *complex* structure than linear.
- A *parametric approach* to add complexity is to incorporate *polynomial terms* in the model.
  - A quadratic model is $f(x;\beta) = \beta_0 + \beta_1 x + \beta_2 x^2$

4.1 Estimation

- OLS uses the weights/coefficients that minimize the SSE loss function over the training data:

$$\hat{\beta} = arg\min_{\beta} SEE(\beta) \quad Note: \beta \text{ in this problem is a } vector$$

$$= arg\min_{\beta} \sum_{i}^{n} \left(y_i - f(x_i;\, \beta)\right)^2$$

$$= arg\min_{\beta} \sum_{i}^{n} (y_i - \beta_0 - \beta_1 x - \beta_2 x^2)^2$$

4.1.1 Estimation

- **Model**

$$f(x_i;\, \beta) = \mathbf{x}^{\mathrm{T}}\beta$$

$$\mathbf{x} = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix} \qquad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}$$

Your Turn #3

Solve for $\hat{\beta}$ using matrix notation.

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} \quad X = \begin{bmatrix} 1 & X_1 & X_1^2 \\ 1 & X_2 & X_2^2 \\ \vdots & \vdots & \vdots \\ 1 & X_n & X_n^2 \end{bmatrix} \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}$$

Sol)

Loss function:

$$RSS(\beta) = (Y - X\beta)^{\intercal}(Y - X\beta)$$
$$= Y^{\intercal}Y - \beta^{\intercal}X^{\intercal}Y - Y^{\intercal}X\beta + \beta^{\intercal}X^{\intercal}X\beta$$
$$= Y^{\intercal}Y - 2\beta^{\intercal}X^{\intercal}Y + \beta^{\intercal}X^{\intercal}X\beta$$

Note: see below how $\frac{\partial \beta^{\intercal}X^{\intercal}X\beta}{\partial \beta}$ can be calculated.

$$\frac{\partial RSS(\beta)}{\partial \beta} = -2X^{\intercal}Y + 2X^{\intercal}X\beta = 0$$

$$\hat{\beta} = (X^{\intercal}X)^{-1}X^{\intercal}Y$$ >> This is the least square solution.

Given the expression $f(\beta) = \beta^{\intercal}X^{\intercal}X\beta$, where β is a column vector of dimension $p \times 1$ and X is a matrix of dimension $p \times 1$, you want to differentiate f with respect to β.

To differentiate this, we'll use the following properties of matrix derivatives:

1. $\frac{\partial}{\partial \mathrm{X}}(X^{\intercal}AX) = (A + A^{\intercal})X$ where A is a symmetric matrix
2. $\frac{\partial}{\partial \mathrm{X}}(X^{\intercal}A) = A$ where A is a matrix.

Let's differentiate the expression step by step:

Given:
$f(\beta) = \beta^{\intercal}X^{\intercal}X\beta$

Let $A = X^{\intercal}X$. Then, $f(\beta) = \beta^{\intercal}A\beta$.

Using the first property:
$\frac{\partial f}{\partial \mathrm{X}} = (A + A^{\intercal})\beta$

Since $A = X^{\intercal}X$ is symmetric (i.e., $A = A^{\intercal}$), the derivative simplifies to:
$\frac{\partial f}{\partial \beta} = 2A\beta$

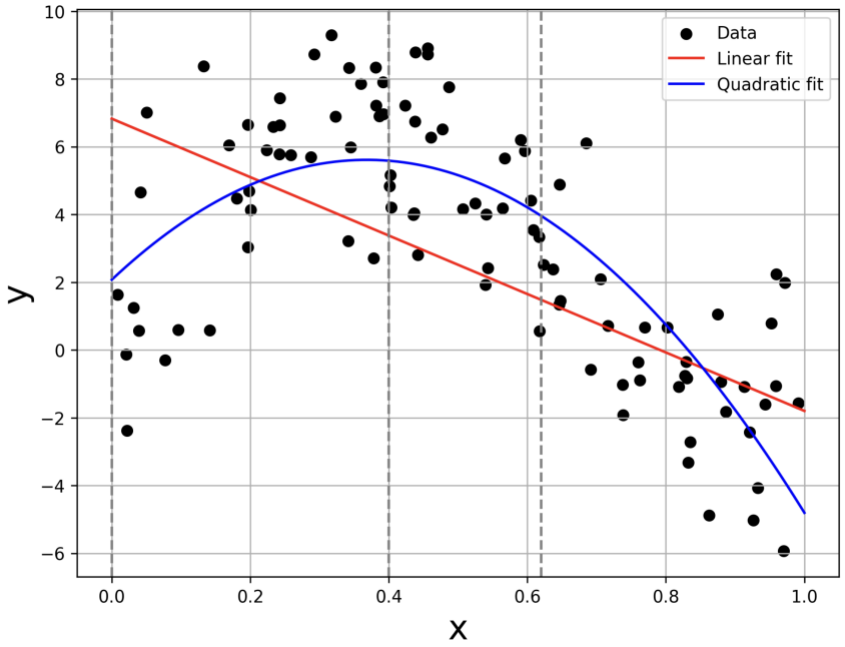
Substituting back for A:
$\frac{\partial f}{\partial \beta} = 2X^{\intercal}X\beta$

So, the gradient of $f(\beta)$ with respect to β is $2X^{\intercal}X\beta$.

4.1.2 Python implementation

In Python, the `OLS` function in `statsmodels.api` is a convenient way to get polynomial terms.

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Assuming sim_x and sim_y are already defined elsewhere, and you've already
set up data_train as before
#-- Model Settings
n = 100
sd = 2
x = sim_x(n)
y = sim_y(x, sd)

x = x.flatten()
y = y.flatten()

data_train = pd.DataFrame({'x': x, 'y': y})

# Create the polynomial features for the model
data_train['x2'] = data_train['x']**2
X = sm.add_constant(data_train[['x', 'x2']])

# Fit the quadratic regression model
model2 = sm.OLS(data_train['y'], X)
result2 = model2.fit()

# Create a prediction dataset
xseq = np.linspace(0, 1, 200)  # sequence of 200 equally spaced values
between 0 and 1
xeval2 = pd.DataFrame({'x': xseq, 'x2': xseq**2})
xeval2 = sm.add_constant(xeval2)
yhat2 = result2.predict(xeval2)
```

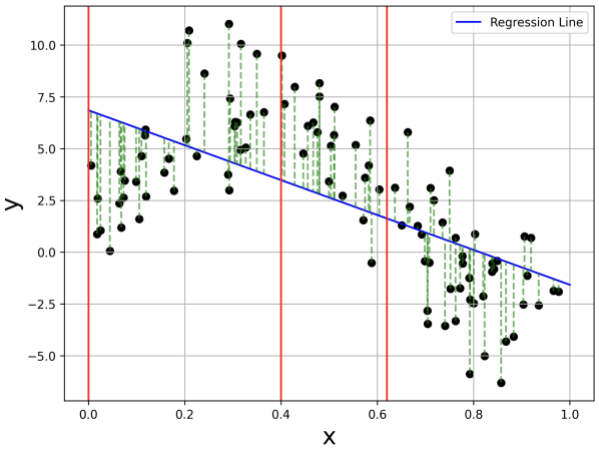


Your Turn #4

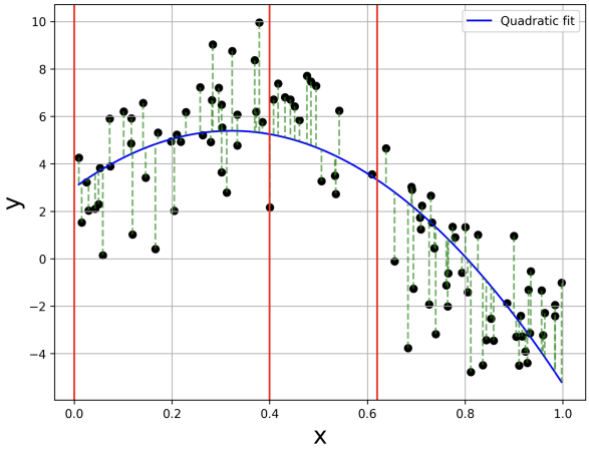
1. How did we do? If X_{new} is close to 0, or close to 0.4, or close to 0.62?
2. But does the quadratic model fit better *overall*?
3. What is the *complexity* of the quadratic model?
   (The effective degrees of freedom is essentially the number of parameters you're estimating.)

Degree	MSE	# of params
1	8.3	2
2	5.5	3

Linear Fit



Quadratic Fit



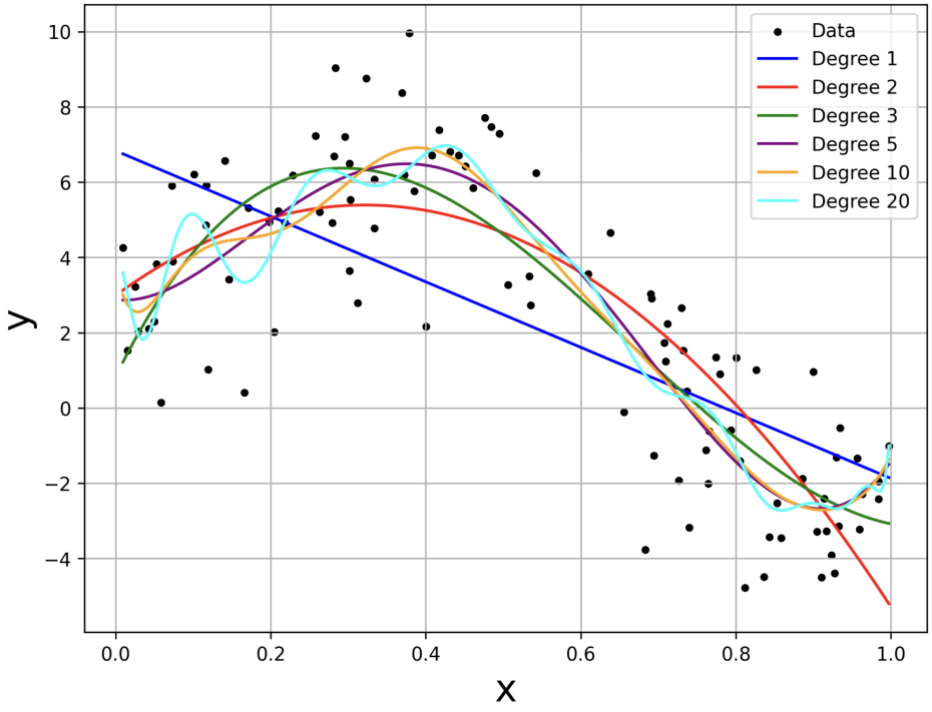
As kids always reason, “if a little is good, than a lot much be better”> So why not try more complex models by increasing the polynomial degree.

- Polynomial of degree *d*

$$f_{poly}(x;\ \beta, d) = \beta_0 + \sum_{j=1}^{d} \beta_j x^j$$

Degree	MSE	# of params
1	7.5	2
2	4.5	3
3	3.7	4
5	3.3	6
10	3.2	11
20	3.0	21

And its always good to observe the plot.



- For `degree=20`, the behavior at the end points are a bit erratic.
- Using a higher degree would further reduce the MSE, but the fitted curve would be more “complex” and may not be a s good for new data.

5 *k*-nearest neighbor models (*k*-NN)

- The *k*-NN method is a non-parametric *local* method, meaning that to make a prediction $\hat{y}|x$, it only uses the training data in the vicinity of *x*.
  - contrast with OLS linear regression, which uses all *x*'s to get prediction.
- The model is simple to describe:

$$f_{\text{knn}}(x;k) = \frac{1}{k}\sum_{i:x_i \in N_k(x)} y_i = \text{Avg}\left(y_i \mid x_i \in N_k(x)\right)$$

  - $N_k(x)$ are the set of *k* nearest neighbors to *x*
  - only the *k* closest *y*'s are used to generate a prediction
  - it is a *simple mean* of the *k* nearest observations

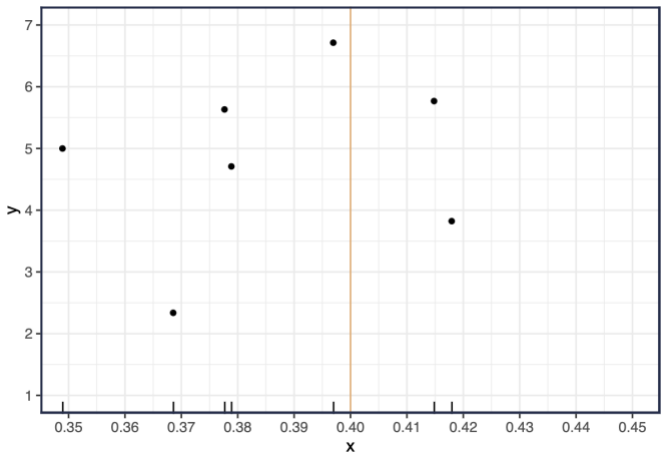
Your Turn #5

What is the estimate $f_{\text{knn}}(x; k = n)$?

5.1 Example & Note

5.1.1 Example

Let's zoom in on the region around x=0.4



x	y	k	D	$\hat{f}_{\text{knn}}(x;k)$
0.397	6.710	1	0.003	6.710
0.415	5.763	2	0.015	6.237
0.418	3.819	3	0.018	5.431
0.379	4.705	4	0.021	5.249
0.378	5.628	5	0.022	5.325
0.369	2.333	6	0.031	4.826
0.349	4.994	7	0.051	4.850

5.1.2 Note about *k*-NN

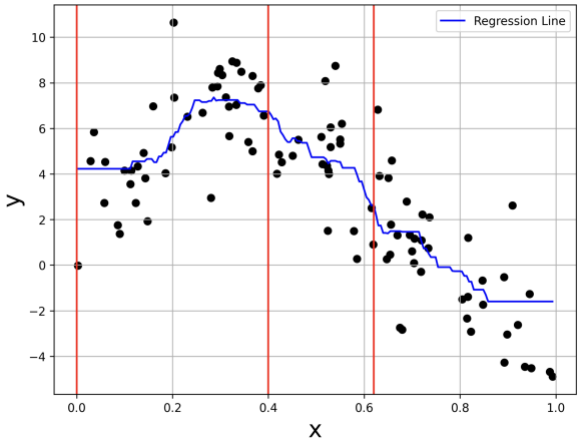
- A suitable *distance* measure (e.g., Euclidean) must be chosen.
  - And predictors are often *scaled* (same standard deviation or range) so one variable doesn't dominate the distance calculation.

- Because the distance to neighbors grows exponentially with increased dimensionality/features, the *curse of dimensionality* is often referenced with respect to *k*-NN.
    - This means that in high dimensions most *neighbors* are not very close and the method becomes less *local*.
- One computational drawback of *k*-NN methods is that all the training data must be stored in order to make predictions.
    - For large training data, may need to sample (or use prototypes)
- The complexity of a *k*-NN model increases as k decreases.
- The least complex model, which is a constant, occurs when $k = n$.
- The most complex model is when $k = 1$.
- The effective degrees of freedom or *edf* for *k*-NN model is n/k.
    - this is a measure of the model complexity. It is approximately the number of parameters that are estimated in the model (to allow comparison with parametric models)

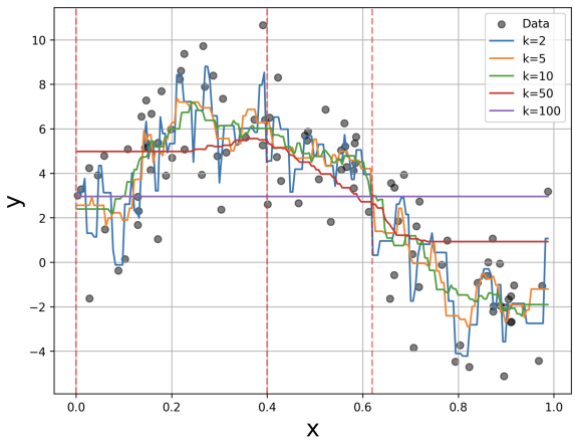
5.2 *k*-NN in action

In Python, the function `KNeighborsRegressor` from `sklearn.neighbors` library will fit a *k*-NN regression model. Here is a $k = 20$ nearest neighbor model.

k-NN k=20



k-NN Models



5.2.1 *k*-NN in action

k	MSE	edf
100	7.5	1
50	4.5	2
10	3.7	10
5	3.3	20
2	3.2	50

6 Predictive Model Comparison (or how to choose the best model)

6.1 Predictive Model Evaluation

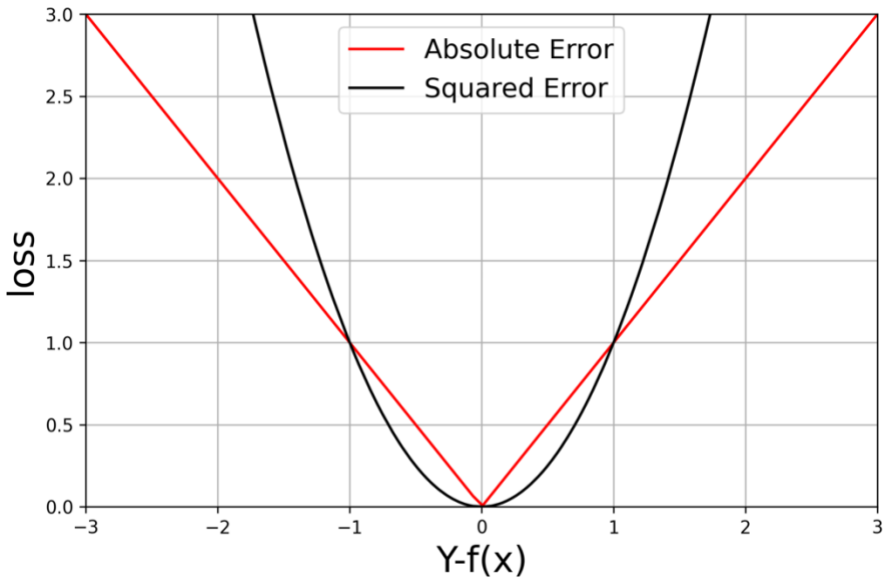
Our goal is prediction, so we should evaluate the models on their *predictive performance*.

- We need to use hold-out data (i.e., data not used to fit the model) to evaluate how well our models do in prediction.
- Call these data test data $D_{\text{test}} = \{(X_j, Y_J)\}_{j=1}^{J}$
    - Note: assume that the test data comes from the same distribution as the training data
    - Or $P_{\text{test}}(X, Y) = P_{\text{train}}(X, Y)$
    - **both** Y and X from same distribution
- Late in the course we will cover ways to do this when we only have training data (e.g., cross-validation)
- but for today, we have an unlimited amount of *test data* at our disposal (since we know how the data were generated)

6.2 Statistical Decision Theory

- In a prediction context, we want a *point estimate* for the value of an unobserved random variable (r.v.) $Y \in \mathbb{R}$ given an input feature $X \in \mathbb{R}$.
- Let $f(X)$ be the prediction of Y given X.
- Define a *loss function* $L(Y, f(\text{X}))$ that indicates how bad it is if we estimate the value Y by $f(X)$.
    - e.g., Y is the number of rainfall events at GIST and X is the day of week from August to October.
    - If we guess $f(X) = 2$, but there are really $Y = 1$, how bad would that be?
- A common loss function is *squared error*:

$$L(Y, f(X)) = (Y - f(X))^2$$



- The best model is the one that minimizes the *expected loss* or Risk or Expected Prediction Error (EPE)

$$Risk = EPE = E[loss]$$

- For *squared error*, the *risk* for using the model *f* is:

$$R(f) = E_{XY}\left[L\left(Y, f(X)\right)\right] = E_{XY}[\left(Y - f(X)\right)^2]$$

where the expectation is taken with respect (w.r.t.) to the *test values* of *X* and *Y*.

  - Note under squared error loss, the risk is also known as the *mean squared error* (MSE)

- To simplify a bit, let's examine the risk of model *f* at a given fixed input $X = x$. This removes the uncertainty in *X*, so we only have uncertainty coming from *Y*.

$$\begin{aligned} R_x(f) &= E\left[L\left(Y, f(x)\right)|X = x\right] \\ &= E[\left(Y - f(x)\right)^2|X = x] \end{aligned} \qquad \text{for squared error loss}$$

where the expectation is taken with respect to $Y|X = x$.

- The best prediction $f^*(x)$, given $X = x$, is the value that minimizes the risk:

$$\begin{aligned} f^*(x) &= \arg\min_c R_x(c) \\ &= \arg\min_c E[(Y - c)^2|X = x] \end{aligned}$$

Your Turn #6

What is the optimal prediction at $X = x$ under the squared error loss?

- i.e., find $f^*(x)$.

6.2.1 Squared Error Loss Functions

- Conclusion: If quality of prediction is measured by squared error, then the best predictor is the (conditional) expected value $f^*(x) = E[Y|X = x]$.
  - And the minimum Risk/MSE is $R_x(f^*) = \mathrm{V}[Y|X = x]$.

- **Summary**: Under *squared error loss* the Risk (at input *x*) is

$$\begin{aligned} & R_x(f) = \mathrm{E}_Y\big[L\big(Y, f(X)\big)\big|X = x\big] \\ & = \mathrm{E}_Y\left[\big(Y - f(x)\big)^2\middle|X = x\right] \quad \text{using squared error loss} \\ & = Var[Y|X = x] + \big(\mathrm{E}_Y[Y|X = x] - f(x)\big)^2 \\ & = \text{Irreducible Variance} + \text{model squared error} \end{aligned}$$

Note: $\mathrm{Var}[Y|X = x] = \mathrm{E}_Y[Y^2|X = x] - (\mathrm{E}_Y[Y|X = x])^2$

6.2.2 *k*-NN and Polynomial Regression

- The *k*-NN model estimates the conditional expectation by using the data in a *local region* around *x*:

$$\hat{f}_{knn}(x; k) = Ave(y_i|x_i \in \mathrm{N_k}(x))$$

This assumes that the true $f(x)$ can be well approximated by a *locally constant* function.

- Polynomial (linear) regression, on the other hand, assumes that the true $f(x)$ is well approximated by a *globally polynomial* function.

$$\hat{f}_{poly}(x; d) = \beta_0 + \sum_{j=1}^{d} \beta_j x^j$$

6.2.3 Empirical Risk

- The actual Risk/EPE is based on the expected error from *test data* (out-of-sample), or data that was no used to estimate $\hat{f}$.

$$R(f) = \mathrm{E}_{XY}\left[L\left(Y, f(X)\right)\right]$$
$$= E_{XY}\left[\left(Y - f(X)\right)^2\right] \quad for\ squared\ error\ loss$$

where X, Y are from Pr(X, Y) (i.e., test data).

- But is it a bad idea to choose the best model according to empirical risk or training error?

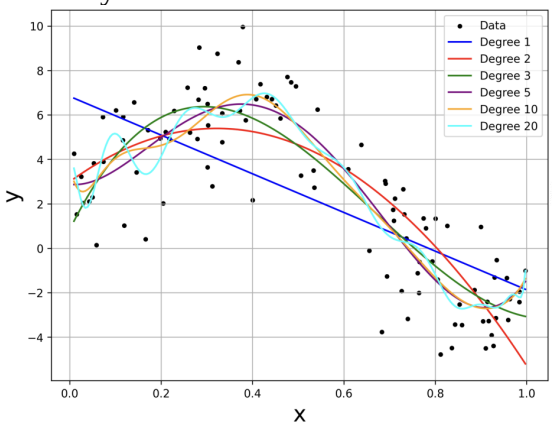
$$\hat{R}_n(f) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f(x_i)\right)$$
$$= \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(x_i)\right)^2 \quad for\ squared\ error\ loss$$

6.3 Choose the best *predictive* model

Your Turn #7

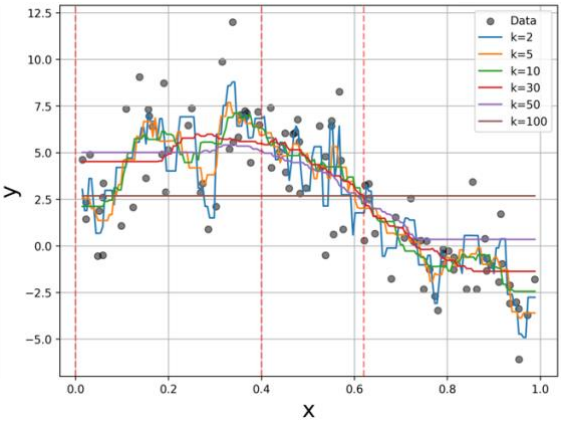
Which model will you choose?
Enter you answer on ***slido***.

Polynomial Models



Degree	MSE	# of params
1	7.5	2
2	4.5	3
3	3.7	4
5	3.3	6
10	3.2	11
20	3.0	21

k-NN Models



k	MSE	edf
100	12.7	1
50	6.6	2
30	4.6	3.3
10	3.9	10
5	3.7	20
2	2.2	50