01 Generalization

June 6, 2021

Information: *Some common concepts in the process of training and generalization*

Written by: *Zihao Xu*

Last update date: *06.06.2021*

1 Motivation

With the introduction of *linear regression*, now we have a basic understanding of what the **training process** is. As shown in the notebook about *linear regression*, we see many different training results with different models. That naturally leads to several simple ideas:

- How to evaluate the performance of a trained model?
- How to select a best model for a given dataset?
- What if we indeed need a lot of parameters while only relatively small datasets are available?
- …

Here list some common concepts and simple techniques related to the topics above. I would like to write them here because they are very basic and will be commonly used in Further exploration would be written together with specific learning algorithms and models.

2 Capacity, Underfitting and Overfitting

2.1 Training and Generalization

2.1.1 Goal of Training

- Learn the **true relationships** (or **true patterns**) from some data pairs

$$(\mathbf{x}_i, y_i), i = 1, 2, \cdots, N$$

- What we learn needs to **generalize** beyond the training data

2.1.2 Definition of Generalization

- The trained model should perform well on **new, previously unseen** inputs
- The ability to perform well on previously unobserved inputs is called **generalization**

2.1.3 Training error and Generalization error

Typically, when training a machine learning model, we have access to a training set

- In an **optimization** problem, we only cares reducing some error computed on the training set, called **training error**
- In a **machine learning** problem, we would also like to make the expected value of the error on a new input, called **generalization error** to be low as well

Problematically, we can never calculate the generalization error exactly. We typically **estimate** the generalization error of a machine learning model by measuring its performance on a **test set** of examples that were collected separately from the training set. This is called the **test error**.

Take the linear regression as an example, the model is trained by minimizing the training error:

$$\frac{1}{m^{(\text{train})}} \left\| \mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})} \right\|$$

However, what we actually care about is the test error:

$$\frac{1}{m^{(\text{test})}} \left\| \mathbf{X}^{(\text{test})}\mathbf{w} - \mathbf{y}^{(\text{test})} \right\|$$

2.2 Statistical Learning Theory

2.2.1 Motivation

To affect performance on the test set when we can observe only the training set, we would be able to make some progress if we are allowed to make some assumptions about how the training and test set are collected

2.2.2 The i.i.d. assumptions

Data-generating process:

- The training and test data are generated by a probability distribution over datasets called the data-generating process

i.i.d. assumption:

- The examples in each dataset are **independent** from each other
- The training set and test set are **identically distributed**, drawn from the same probability distribution as each other

2.2.3 Guidance on Machine Learning

The **expected training error of a randomly selected model is equal to the expected test error** of that model. Given a probability distribution $p(\mathbf{x}, y)$ which we sample from repeatedly to generate the training set and the test set, the expected training set error is exactly the same as the expected test set error for some fixed parameters $\boldsymbol{\theta}$

In machine learning, we sample the training set, then use it to choose the parameters to reduce training set error, then sampling the test set. Under this process, the expected test error is greater than or equal to the expected value of training error.

Therefore, the factors determining **how well a machine learning algorithm will perform** are its ability to

- Make the training error small

- Make the gap between training and test error small

These two factors correspond to two challenges in machine learning are:

- **Underfitting**: The model is not able to obtain a sufficient low error value on the training set
- **Overfitting**: The gap between the training error and the test error is too large

2.3 Model Capacity

Informally, a model's capacity is its ability to fit a wide variety of functions

- Models with low capacity may struggle to fit the training set
- Models with high capacity can overfit by memorizing properties of the training set that do no serve them well on the test set
- Difficult to compare the complexity among members of substantially different model classes

There are many factors affecting the model capacity

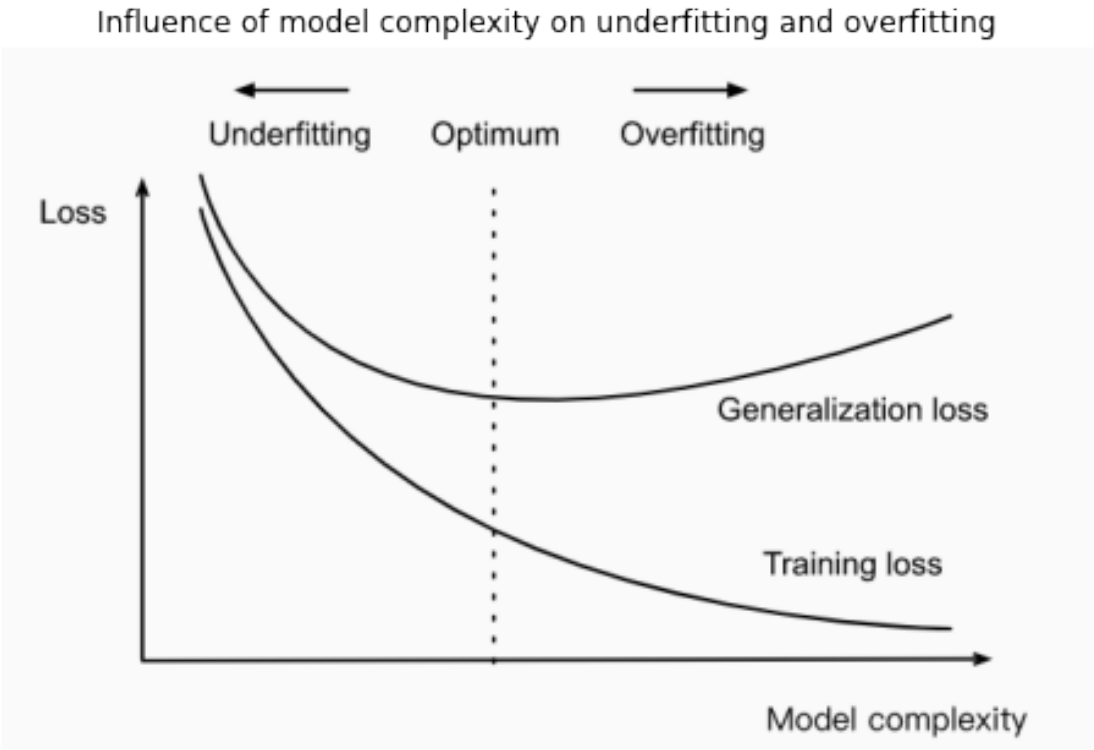
- Number of features and corresponding parameters
- Family of functions the learning algorithm can choose from
- ...

Basic ideas about improving the generalization: **Occam's razor**:

- Among competing hypothesis that explain known observations equally well, we should choose the "simplest" one

Typically, generalization error has a U-shaped curve as a function of model capacity. Here is a simple figure from *Dive into deep learning*

[1]:
```
from PIL import Image
import matplotlib.pyplot as plt

img = Image.open('Loss_vs_capacity.png')
fig, ax = plt.subplots(figsize=(8, 6))
ax.imshow(img)
ax.axis('off')
ax.set_title("Influence of model complexity on underfitting and overfitting")
plt.show()
```



2.4 Dataset Size

Dataset size is another important factor which tend to the influence the generalizability of a model. For a fixed model:

- The fewer samples we have in the training dataset, the more likely we are to encounter overfitting
- As we increase the amount of data, the generalization error typically decreases. (**More data never hurt**)

For a fixed task and data distribution, there is typically a relationship between model complexity and dataset size

- Given more data, we might profitably attempt to fit a more complex model
- Absent sufficient data, simpler models may be more difficult to beat

2.5 Hyper-parameters and Validation Sets

2.5.1 Hyper-parameters

Most machine learning algorithms have hyper-parameters, settings that we can use to control the algorithm's behavior

- The values of hyper-parameters are not adapted by the learning algorithm itself
- For example, the **polynomial regression** has a single hyper-parameter: the degree of the polynomial which acts as a **capacity hyper-parameter**

For most hyper-parameters, it is not appropriate to learn them on the training set

2.5.2 Validation set

Motivation:

- On one hand, it is important that the test examples are not used in any way to make choices about the model, including its hyper-parameters. If we use the test data in the model selection process, there is a risk that we might overfit the test data.
- On the other hand, we cannot rely solely on the training data for choosing hyper-parameters because we cannot estimate the generalization error on the very data that we use to train the model.

Solution:

- Split the training data into two disjoint subsets
- The subset of data used to learn the parameters is still typically called the **training set**
- The subset of data used to guide the selection of hyper-parameters is called the **validation set**
- Typically, one uses about 80% of the training data for training and 20% for validation

Remark:

- Since the validation set is used to "train" the hyper-parameters, the validation set error will underestimate the generalization error
- In this notebook, the data to be worked with are actually **training data** and **validation data** with no true test sets. The reported accuracy in each experiment is really the validation accuracy and not a true test set accuracy.

2.5.3 K-Fold Cross-Validation

Motivation:

- When the training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set

Solution:

- Split the original training data into K non-overlapping subsets
- Execute the training and validation process K times, each time training on $K-1$ subsets and validating on a different subset
- The training and validation errors are estimated by averaging over the results from the K experiments

2.6 Estimators, Bias and Variance

Foundational concepts such as parameter estimation, bias and variance are useful to **formally characterize** notions of generalization, underfitting and overfitting

2.6.1 Point Estimation

Let $\{\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)}\}$ be a set of m independent and identically distributed data points. A **point estimator** (or **statistic**) is any function of the data

$$\hat{\boldsymbol{\theta}}_m = g(\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)})$$

- The definition does not require that g return a value that is close to the true $\boldsymbol{\theta}$ or even that the range of g be the same as the set of allowable values of $\boldsymbol{\theta}$
- A good estimator is a function whose output is close to the true underlying $\boldsymbol{\theta}$ that generated from the training data

Assume that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a function of the data. Since the **data is drawn from a random process**, any function of the data is random. Therefore, $\hat{\boldsymbol{\theta}}$ is a **random variable**.

2.6.2 Bias

The **bias** of an estimator is defined as

$$\text{bias}\left(\hat{\boldsymbol{\theta}}_m\right) = \mathbb{E}\left(\hat{\boldsymbol{\theta}}_m\right) - \boldsymbol{\theta}$$

where the expectation is over the data (seen as samples from a random variable) and $\boldsymbol{\theta}$ is the true underlying value of $\boldsymbol{\theta}$ used to define the data-generating distribution.

- An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **unbiased** if $\text{bias}\left(\hat{\boldsymbol{\theta}}_m\right) = \mathbf{0}$, which implies that $\mathbb{E}\left(\hat{\boldsymbol{\theta}}_m\right) = \boldsymbol{\theta}$
- An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **asymptotically unbiased** if $\lim_{m\to\infty}\text{bias}\left(\hat{\boldsymbol{\theta}}_m\right) = \mathbf{0}$, which implies that $\lim_{m\to\infty}\mathbb{E}\left(\hat{\boldsymbol{\theta}}_m\right) = \boldsymbol{\theta}$

While unbiased estimators are clearly desirable, they are not always the **best** estimators.

2.6.3 Variance

The **variance** of an estimator is just the variance

$$\mathbb{V}\left(\hat{\boldsymbol{\theta}}_m\right)$$

where the random variable is the training set.

The variance of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently resample the dataset from the underlying data-generating process. Just as we might like an estimator to exhibit low bias, we would also like it to have relatively low variance.

We often estimate the generalization error by computing the sample mean of the error on the test set. The number of examples in the test set determines the accuracy of the estimate. Taking advantage of the central limit theorem, we can use the standard error to compute the probability that the true expectation falls in any chosen interval.

2.6.4 Trading off Bias and Variance

Motivation

- Low variance algorithms tend to be **less complex**, with simple or rigid underlying structure
- Low bias algorithms tend to be **more complex**, with flexible underlying structure
- Algorithms that are not complex enough produce **underfit** models that can't learn the signal from the data
- Algorithms that are too complex produce **overfit** models that memorize the noise instead of the signal

Performance

- **High bias**, low variance algorithms train models that are consistent, but inaccurate on *average*
- **High variance**, low bias algorithms train models that are accurate on average, but inconsistent

Total Error

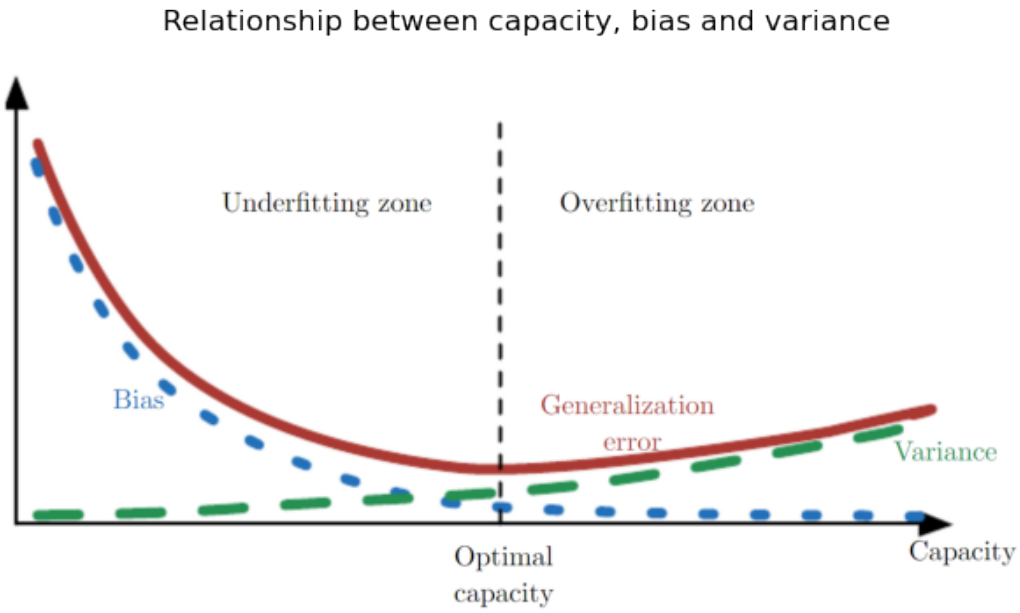
- To get good predictions, find a balance of Bias and Variance that minimizes **total error**

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

  where the irreducible error is "noise" that can't be reduced by algorithms

Here is a figure showing the relationship between model capacity, bias and variance, from the *MIT Deep Learning Book*.

[2]:
```
img = Image.open('Trading_off_Bias_and_Variance.png')
fig, ax = plt.subplots(figsize=(8, 6),dpi=100)
ax.imshow(img)
ax.axis('off')
ax.set_title("Relationship between capacity, bias and variance")
plt.show()
```



2.6.5 Consistency

The behavior of an estimator as the amount of training data grows is also concerned. In particular, we usually wish that, as the number of data points m in our dataset increases, out point estimates converge to the true value of the corresponding parameters. More formally, we would like that

$$\text{plim}_{m\to\infty}\hat{\theta}_m = \theta$$

The symbol plim indicates convergence in probability, meaning that for any $\epsilon > 0$

$$P\left(\left|\hat{\theta} - \theta\right| > \epsilon\right) \to 0 \text{ as } m \to \infty$$

This property is known as **consistency**

Consistency ensures that the bias induced by the estimator diminishes as the number of data examples grows. However, the reverse is not true - asymptotic unbiasedness does not imply consistency

3 Example: Polynomial Regression

Here is a simple example showing the concepts of overfitting and underfitting. Here directly use the **Scikit-Learn** package for convenience.

3.1 Generate Dataset

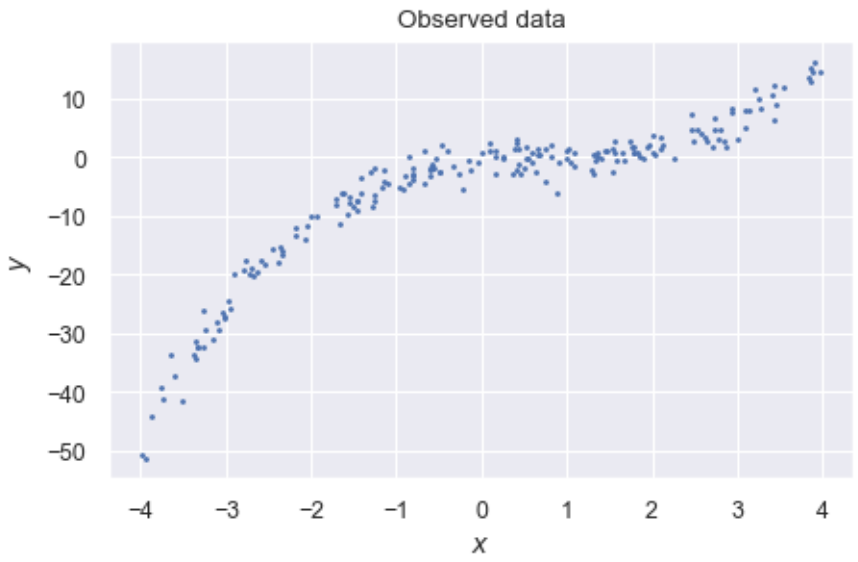
Try to fit a linear regression model to the data generated from

$$y_i = -0.5 + 1.5x_i - 2\frac{x_i^2}{2!} + 2.5\frac{x_i^3}{3!} + 2\epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0, 1)$ and we sample $x_i \sim U([-4, 4])$

First generate the dataset and visualize it

[3]:
```
import seaborn as sns
import numpy as np
# Ensure reproducibility
np.random.seed(123)
# Plot setting
sns.set()
sns.set_context('paper')

# Size of training set and validation set
num_train, num_test = 160, 40
# Number of total observations
num_obs = num_train + num_test
# Sample x
x = (8 * np.random.rand(num_obs) - 4.0).reshape(-1, 1)
# True parameters
theta_true = np.array([-0.5, 1.5, -2, 2.5]).reshape(-1, 1)
# Get the polynomial featurtes
y = - 0.5 + 1.5 * x - 2 * x ** 2 / 2 + 2.5 * x ** 3 / 6\
+ 2 * np.random.randn(num_obs).reshape(-1,1)
# Visualize the dataset
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x, y, '.', markersize=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Observed data')
plt.show()
```

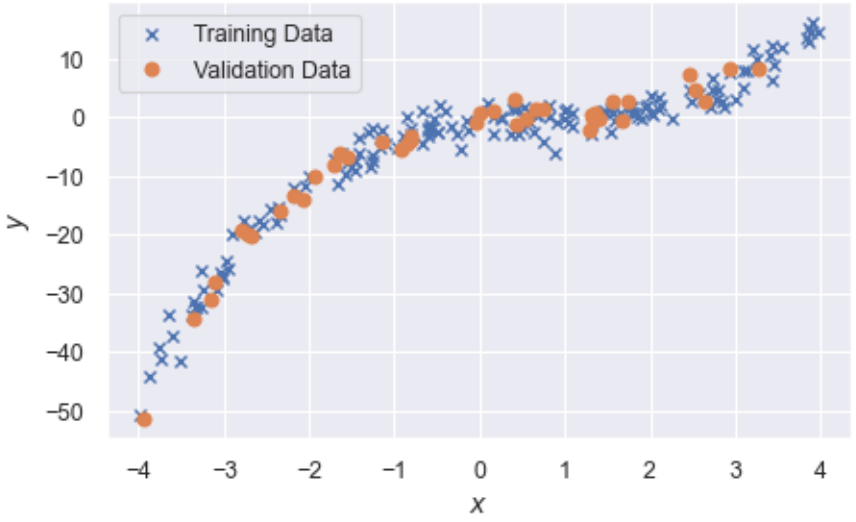


3.2 Split training set and validation set

The naive way is to simply divide the observations into two parts

[4]:
```
# Get x and y zipped
data = np.concatenate((x, y), axis=1)
# Randomly permute rows
data_permuted = np.random.permutation(data)
# Split in a training set by picking the first num_train rows
data_train = data_permuted[:num_train]
# Split in a validation set
data_val = data_permuted[num_train:]
# Get x_train and y_train
x_train = data_train[:, 0].reshape(-1, 1)
y_train = data_train[:, 1].reshape(-1, 1)
# Get x_val and y_val
x_val = data_val[:, 0].reshape(-1, 1)
y_val = data_val[:, 1].reshape(-1, 1)
# Sanity check
print('Shape of train set:\t\t', x_train.shape)
print('Shape of validation set:\t', x_val.shape)
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x_train, y_train, 'x', label='Training Data')
ax.plot(x_val, y_val, 'o', label='Validation Data')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best')
plt.show()
```

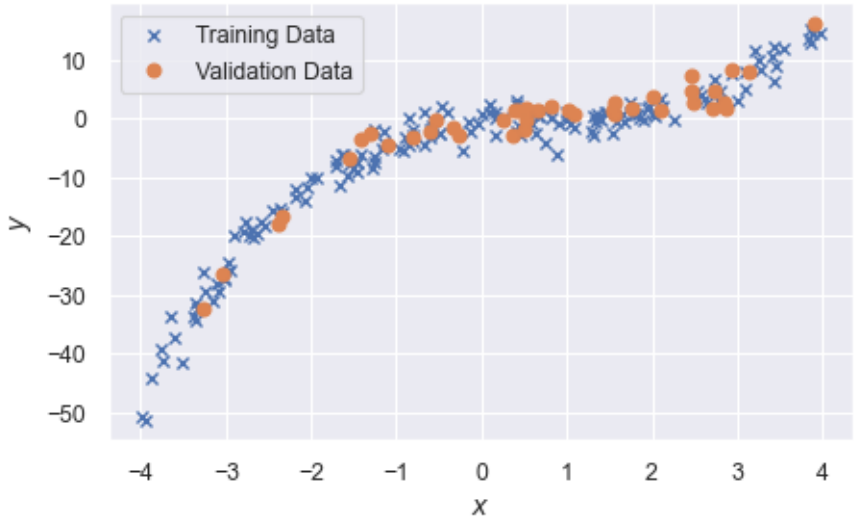
```
Shape of train set:              (160, 1)
Shape of validation set:         (40, 1)
```



We can also use the **train_test_split** utility in **Scikit-Learn** to separate the training set and validation set easily

[5]:
```
from sklearn.model_selection import train_test_split

x_train, x_val, y_train, y_val = \
train_test_split(x,y,test_size=0.2,random_state=123)
# Sanity check
print('Shape of train set:\t\t', x_train.shape)
print('Shape of validation set:\t', x_val.shape)
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x_train, y_train, 'x', label='Training Data')
ax.plot(x_val, y_val, 'o', label='Validation Data')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best')
plt.show()
```

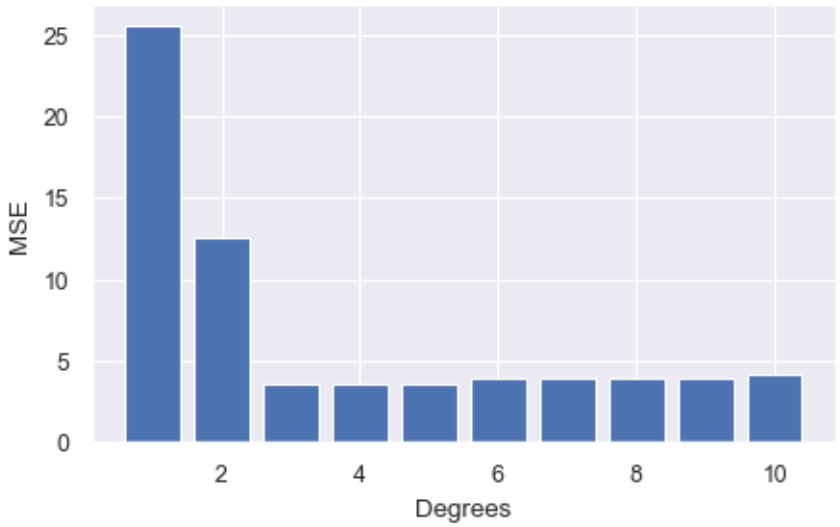
```
Shape of train set:              (160, 1)
Shape of validation set:         (40, 1)
```



3.3 Train and validation

Let's find the best polynomial degree by training and validation

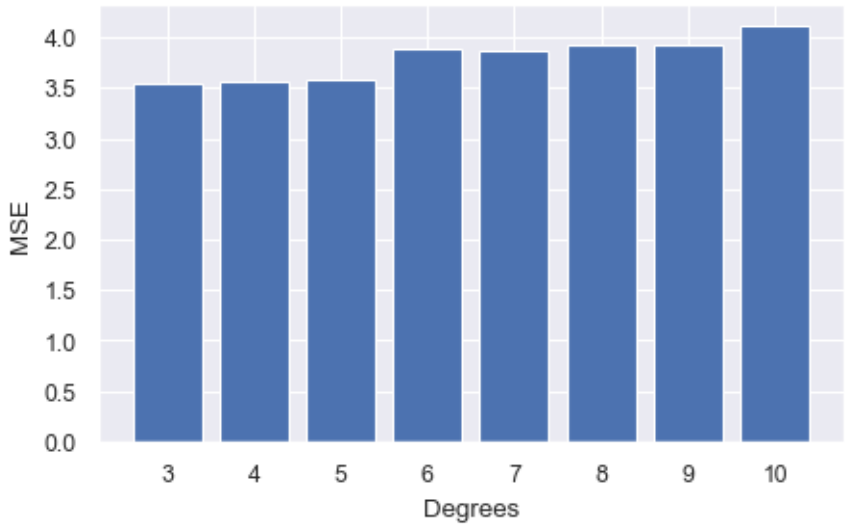
[6]:
```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures

# Try with different degree
degrees = np.arange(1, 11)
MSE = []
Estimator = []
for index, degree in enumerate(degrees):
    # Assign the regression model and preprocessing method
    estimator = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    # Fit with the created model
    estimator.fit(x_train, y_train)
    # See the performance on evaluation points
    y_pred = estimator.predict(x_val).reshape(-1, 1)
    # Calculate the validation MSE
    loss = np.sum((y_pred - y_val)**2) / y_pred.shape[0]
    Estimator.append(estimator)
    MSE.append(loss)
# Plot out the MSE for different degrees
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.bar(range(1, len(MSE) + 1), MSE)
ax.set_xlabel('Degrees')
ax.set_ylabel('MSE')
plt.show()
```



It seems that the MSEs for degree 1 and 2 are so huge that we should not plot them together wit other degrees

[7]:
```
# Get the minimum index
print("Degree %d gives the minimized validation mean square error."%(np.
 ↪argmin(MSE)+1))
# Plot out the MSE for different degrees
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.bar(range(3, len(MSE) + 1), MSE[2:])
ax.set_xlabel('Degrees')
ax.set_ylabel('MSE')
plt.show()
```

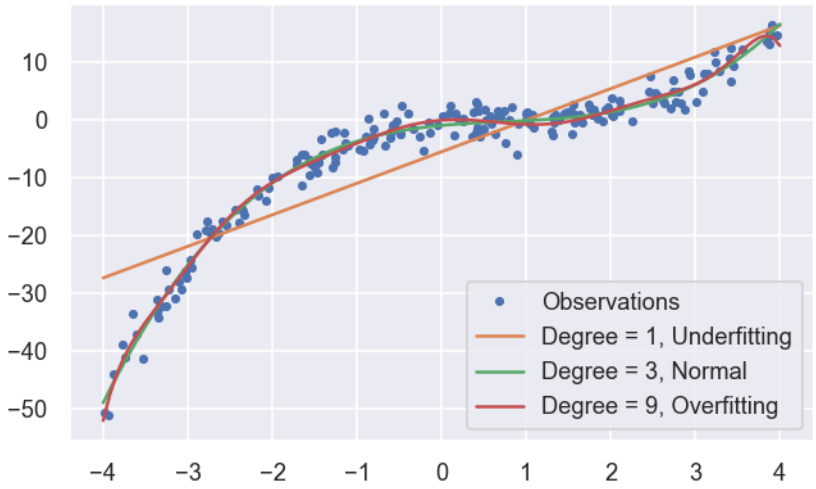
```
Degree 3 gives the minimized validation mean square error.
```



3.4 Visualization - Underfitting, Normal fitting and Overfitting

Here shows the visualization of three types of fitting

[8]:
```
fig, ax = plt.subplots(figsize=(5,3),dpi=150)
xe = np.linspace(-4,4,100).reshape(-1,1)
ye_1 = Estimator[0].predict(xe)
ye_3 = Estimator[2].predict(xe)
ye_10 = Estimator[9].predict(xe)
ax.plot(x,y,'.',label='Observations')
ax.plot(xe,ye_1,label='Degree = 1, Underfitting')
ax.plot(xe,ye_3,label='Degree = 3, Normal')
ax.plot(xe,ye_10,label='Degree = 9, Overfitting')
plt.legend(loc='best')
plt.show()
```


10
0
−10
−20
−30
−40
−50
−4
−3
−2
−1
0
1
2
3
4
Observations
Degree = 1, Underfitting
Degree = 3, Normal
Degree = 9, Overfitting

[ ]: