TP 2 : underfitting, overfitting

Definition 1

Your model is ***underfitting*** *the training data when the model performs poorly on the training data.*

Your model is ***overfitting*** *your training data when you see that the model performs well on the training data but does not perform well on the evaluation data*

Generalization *is the ability of a model to make correct predictions on new data, which were not used to build it.*

Example 1 *The data set is generated as follows. The sample data* $(x_i, y_i)_{i\in\{1,\cdots,n\}}$ *is a realization of the random variables* (X, Y)

- *The random variable* X *follows a uniform distribution over* $[0, 1]$ *(*$X \sim \mathcal{U}[0, 1]$*)*
- *The noise* $\epsilon \in \mathbb{R}$ *is independent from* X *and* $\epsilon \sim \mathcal{N}(0, 1)$
- $Y = \cos(\frac{3\pi}{2}X) + \epsilon$

The data set is splitted into training data (X_{train}, y_{train}) *and test data* (X_{test}, y_{test}) *(see Figure 1). For any* $d \in \{1, \cdots, 19\}$*, we choose* $\mathcal{F}_d = \{f_\theta : x \to \varphi_d(x).\theta,\ \theta \in \mathbb{R}^{d+1}\}$ *as approximate set where for any* $x \in \mathcal{X} = \mathbb{R}$*,* $\varphi_d(x) = (1, x, x^2, \cdots, x^d) \in \mathbb{R}^{d+1}$*.*

```
import numpy as np
import matplotlib.pyplot as plt
!pip install scikit-learn if you use JupiterHub
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from numpy import linalg as LA
```

```
def true_fun(X):
    return np.cos(1.5 * np.pi * X)

np.random.seed(0)

n_samples = 30
degrees = [1,3,20]
X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

X_train, X_test, y_train, y_test=train_test_split(X,y,test_size=0.20,
    random_state=42)
```

```
plt.figure
X_true = np.linspace(0, 1, 100)
plt.scatter(X_train, y_train, edgecolor="b", s=20, label="Training data")
plt.scatter(X_test, y_test, edgecolor="r", s=20, label="Test data")
plt.plot(X_true, true_fun(X_true),color='orange', label="True function")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-2, 2))
plt.show()
```

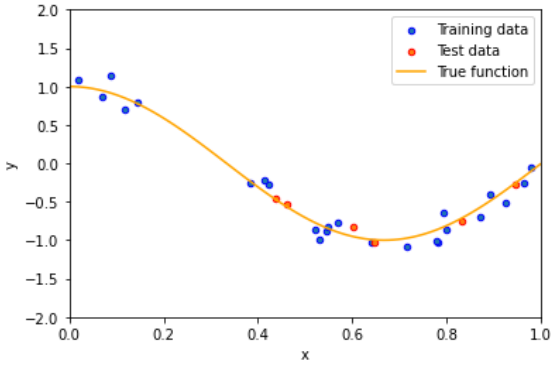


Figure 1: Splitting data

For any degree d, a model $f_{\theta_d^*}$ is created only using (X_{train}, y_{train}). More precisely,

$$\theta_d^* = \underset{\theta}{argmin}\, L(\theta) := \frac{1}{n}\sum_{i=0}^{n-1}(y_{train}[i] - f_\theta(X_{train}[i]))^2$$

where n is the length of X_{train}. Then the training Mean Squared Error (briefly MSE) and the test MSE is computed as follows :

$$Training\ MSE = \frac{1}{n}\sum_{i=0}^{n-1}(y_{train}[i] - f_{\theta_d^*}(X_{train}[i]))^2$$

$$Test\ MSE = \frac{1}{p}\sum_{i=0}^{p-1}(y_{test}[i] - f_{\theta_d^*}(X_{test}[i]))^2$$

where n is the length of X_{train} and p is the length of X_{test}

```
Test_MSE=[]
Training_MSE=[]
for i in range(1,20) :

 polynomial_features = PolynomialFeatures(degree=i, include_bias=False)
 linear_regression = LinearRegression()
 pipeline = Pipeline([("polynomial_features", polynomial_features),
          ("linear_regression", linear_regression), ]
```

```
    )
    model=pipeline.fit(X_train.reshape(-1, 1), y_train.reshape(-1, 1))
    model.predict(X_test.reshape(-1, 1))
    Test_MSE.append(LA.norm(model.predict(X_test.reshape(-1, 1))-y_test.reshape
        (-1,1),2))
    Training_MSE.append(LA.norm(model.predict(X_train.reshape(-1, 1))-y_train.
        reshape(-1,1),2))
```

```
plt.figure
plt.plot(range(1,20),Test_MSE)
plt.plot(range(1,20),Training_MSE)
plt.xlabel("d")
plt.ylabel("MSE")
plt.xticks(range(1,20,3))
plt.legend(loc="best", labels=['Test MSE','Training MSE'])
plt.show()
```

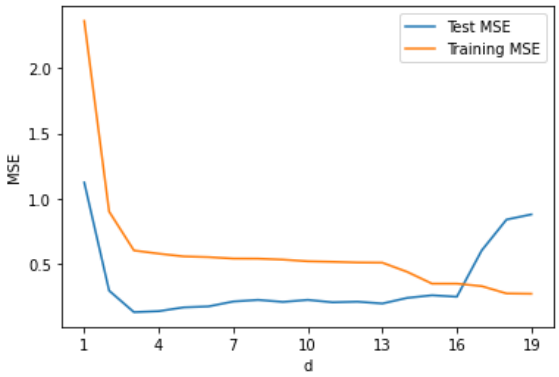


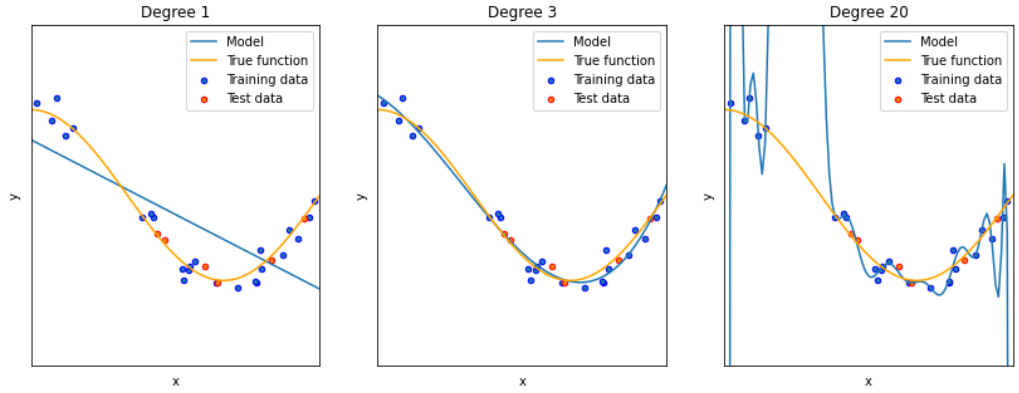
Figure 2: MSE (Mean Squared Error) with respect to the size of $\mathcal{F}_d$

In Figure 2, the training MSE and the test MSE is plotted with respect to the degree d. Since $\mathcal{F}_1 \subset \cdots \subset \mathcal{F}_{19}$*, the 'size' of* $\mathcal{F}_d$ *inscreases with respect to the degree d.*

```
plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i], include_bias=
    False)
    linear_regression = LinearRegression()
    pipeline = Pipeline(
        [
            ("polynomial_features", polynomial_features),
            ("linear_regression", linear_regression),
        ]
    )
    model=pipeline.fit(X_train.reshape(-1, 1), y_train.reshape(-1, 1))
    model.predict(X_test.reshape(-1, 1))

  # Evaluate the models using crossvalidation
```



!h

Figure 3: Polynomial model using `sklearn`

```
    X_true = np.linspace(0, 1, 100)
    plt.plot(X_true, pipeline.predict(X_true[:, np.newaxis]), label="Model")
    plt.plot(X_true, true_fun(X_true), color='orange', label="True function")
    plt.scatter(X_train, y_train, edgecolor="b", s=20, label="Training data")
    plt.scatter(X_test, y_test, edgecolor="r", s=20, label="Test data")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title(
        "Degree {}".format(
            degrees[i]
        )
    )
plt.show()
```

- When $d = 1$ the model is **underfitting**; the model has "not learned enough" from the training data, resulting in low generalization and unreliable predictions. More precisely, it is not flexible enough to take into account the structure of the data set (see on the left side of Figure 3)
- When d is large enough ($d \geq 16$), the model is **overfitting**; the model learning "too much" from the training data set. More precisely, a model learns the details and noises in the training data that negatively impacts the performance of the model on new data (see on the right side of Figure 3).
- Ideally, you want to select a model between underfitting and overfitting. For example $d = 3$ (see Figure 3).