03 - Bias-Variance Dilemma

In this exercise session, we generate a synthetic dataset and examine the phenomenon of bias/variance tradeoff in different models. After that, we will analyse some technique to manage this tradeoff on real data. The presented procedures allow to decide which model, among a set of given models, is best suited for the problem analysed.

Bias-Variance Analysis

At first, let us generate a dataset, so that we have a ground-truth about the real model. In this case, let us assume to have input $x \in [0, 5]$ and target:

$$t = t(x) = f(x) + \epsilon = 1 + \frac{1}{2}x + \frac{1}{10}x^2 + \epsilon, \tag{1}$$

where ϵ is a Gaussian random variable with $\mathbb{E}[\epsilon] = 0$ and $Var(\epsilon) = \sigma^2 = 0.7^2$.

In [1]:
```
import numpy as np
n_points = 1000
eps = 0.7

def fun(x):
  return 1 + 1/2 * x + 1/10 * x**2

x = np.random.uniform(low=0, high=5, size=(n_points, 1))
```

In [2]:
```
t = fun(x)
t_noisy = t + eps * np.random.randn(n_points, 1)
```

After that, we consider two different linear regression models:

$$L_1 : y(x) = a + bx \tag{2}$$
$$L_2 : y(x) = a + bx + cx^2 \tag{3}$$

If for the former one we do not need to make use of additional features, for the latter one we need to define the basis functions $\phi_1(x_i) = x_i$ and $\phi_2(x_i) = x_i^2$

In [3]:
```
phi = np.concatenate([x, x**2], axis=1)
```

Once we generated the inputs for both the models, we train them:

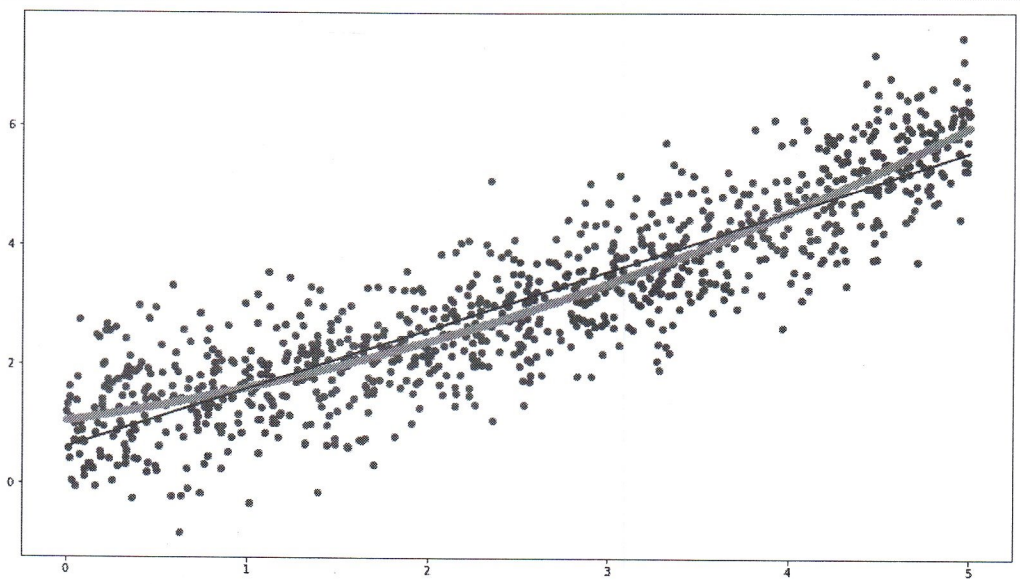
In [4]:
```
from sklearn import linear_model
lin_model = linear_model.LinearRegression()
lin_model.fit(x, t_noisy)
```

Out[4]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [5]:
```
qua_model = linear_model.LinearRegression()
qua_model.fit(phi, t_noisy)
```

Out[5]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [6]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(16,9))
plt.scatter(x, t_noisy, label='true')

plt.plot(x, lin_model.predict(x), label='linear model', color='red')
plt.scatter(x, qua_model.predict(phi), label='quadratic model', color='orange')

plt.show()
```

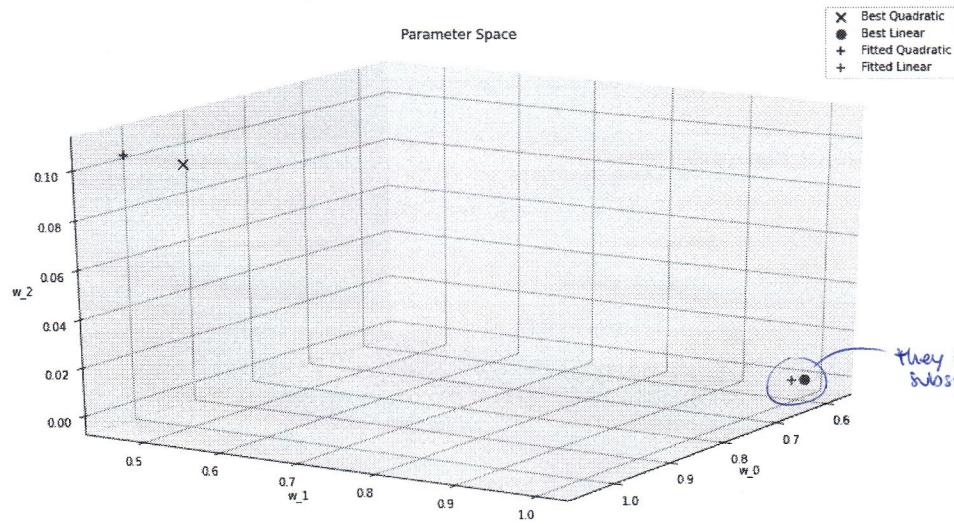


Let us plot in the parameter space the models we estimated and the optimal ones, where the optimal one in the family

L_1 is the model that $min_{a,b} \int_0^5 (f(x) - a - bx)^2 dx$:

In [7]:
```
# Real and best paramseters
real_par = np.array([1, 1/2, 1/10])
best_lin_par = np.array([7/12, 1, 0])

# Fitted parameters
lin_c = np.array([lin_model.intercept_[0], lin_model.coef_[0][0], 0])
qua_c = np.concatenate([qua_model.intercept_, qua_model.coef_.flatten()])
```

In [8]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111, projection='3d')
s = 70

ax.scatter(*real_par, marker='x', color='blue', s=s, label='Best Quadratic')
ax.scatter(*best_lin_par, marker='o', color='red', s=s, label='Best Linear')
ax.scatter(*qua_c, marker='+', color='blue', s=s, label='Fitted Quadratic')
ax.scatter(*lin_c, marker='+', color='red', s=s, label='Fitted Linear')

ax.set_xlabel('w_0')
ax.set_ylabel('w_1')
ax.set_zlabel('w_2')
ax.view_init(elev=20., azim=32)
plt.title('Parameter Space')
plt.legend()
plt.grid()
plt.show()
```



As you can see, the parameters of the family L_1 lies in a 2D subspace, while the approximated quadratic models (i.e., those in L_2) may span over a 3D space. Hence, models from L_2 might coincide with the real model, i.e., real_par, while linear models coming from L_1 will always suffer from a bias which can not be reduced to zero.

If we want to compute the bias and variance of the two chosen models, we should iterate the estimation procedure over multiple datasets.

In [9]:
```
# We write in a function the code we have seen above
def sample_and_fit(n_points):
  x = np.random.uniform(low=0, high=5, size=(n_points, 1))
  t = fun(x)
  t_noisy = t + eps * np.random.randn(n_points, 1)
  lin_model = linear_model.LinearRegression()
  lin_model.fit(x, t_noisy)
  phi = np.concatenate([x, x**2], axis=1)
  qua_model = linear_model.LinearRegression()
  qua_model.fit(phi, t_noisy)
  return lin_model, qua_model

# repeats the code above for n_repetition times, and saves the results
def multiple_sample_and_fit(n_repetitions, n_points):
  lin_coeff = np.zeros((n_repetitions, 3))
  qua_coeff = np.zeros((n_repetitions, 3))

  for i in range(n_repetitions):
    lin_model, qua_model = sample_and_fit(n_points)
    # Store the results
    lin_coeff[i, :] = np.concatenate([lin_model.intercept_, lin_model.coef_.flatten(), np.zeros(1)])
    qua_coeff[i, :] = np.concatenate([qua_model.intercept_, qua_model.coef_.flatten()])
  return lin_coeff, qua_coeff

# plot the models coefficients in a 3D space
def plot_models(lin_coeff, qua_coeff):
  fig = plt.figure(figsize=(16, 9))
  ax = fig.add_subplot(111, projection='3d')

  # (only these 2 lines have changed)
  ax.scatter(qua_coeff[:, 0], qua_coeff[:, 1], qua_coeff[:, 2], marker='.', color='blue', s=s, alpha=0.3)
```

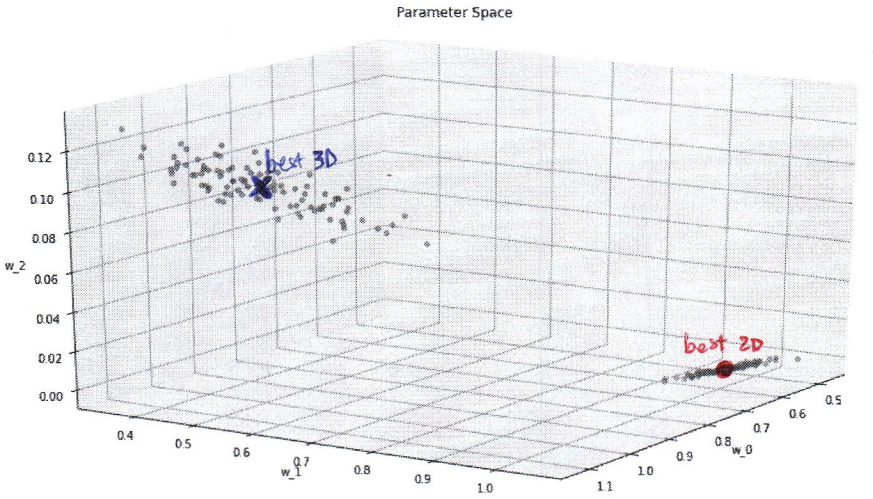
sample some x, compute t_noisy, fit linear and quadratic models

run the above function multiple times and store the coefficients

```
    ax.scatter(lin_coeff[:, 0], lin_coeff[:, 1], lin_coeff[:, 2], marker='.', color='red', s=s, alpha=0.3)

    ax.scatter(*real_par, marker='x', color='blue', s=100)
    ax.scatter(*best_lin_par, marker='o', color='red', s=100)
    ax.set_xlabel('w_0')
    ax.set_ylabel('w_1')
    ax.set_zlabel('w_2')
    ax.view_init(elev=20., azim=32)
    plt.title('Parameter Space')
    plt.grid()
    plt.show()
```

We sample, fit and plot the obtained models:

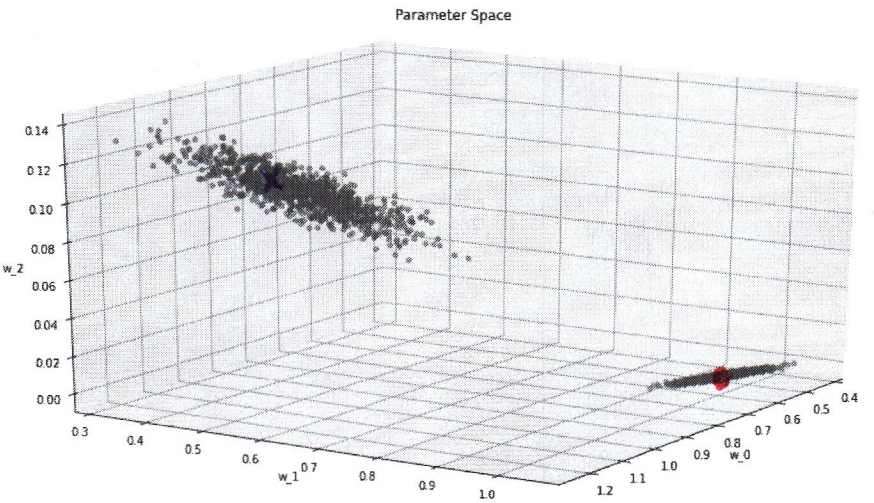
In [10]:
```
n_repetitions = 100
n_points = 1000
lin_coeff, qua_coeff = multiple_sample_and_fit(n_repetitions, n_points)
plot_models(lin_coeff, qua_coeff)
```



It is possible to see how the different trained models (represented as crosses and x symbols) spread around the optimal parameters.

If we increase the number of repetitions, we just obtain more point in the cloud:

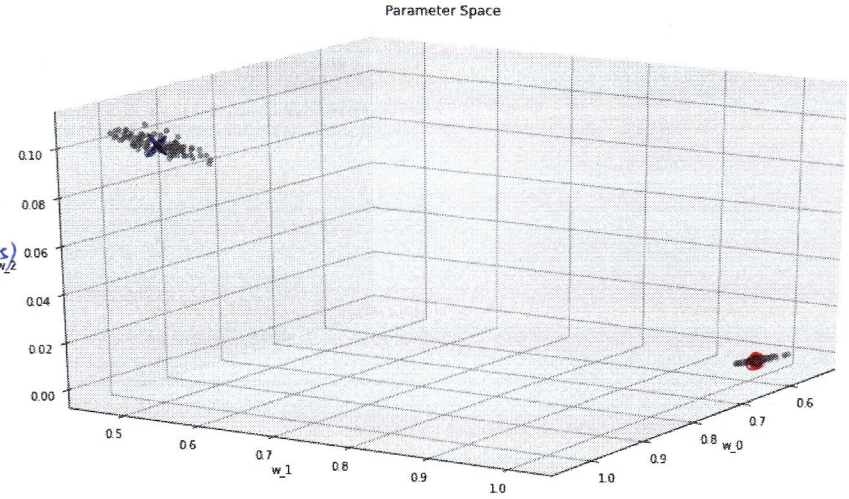
In [11]:
```
n_repetitions = 1000    (↑)
n_points = 1000
lin_coeff, qua_coeff = multiple_sample_and_fit(n_repetitions, n_points)
plot_models(lin_coeff, qua_coeff)
```



just more points, showing changes in terms of variability

On contrary, the more the points we consider for estimation ($N \to \infty$), the more the resulting parameter vectors are close to the real and the best model in the hypothesis space, depending on the hypothesis space we choose.

In [12]:
```
n_repetitions = 100
n_points = 10000    (↑)
lin_coeff, qua_coeff = multiple_sample_and_fit(n_repetitions, n_points)
plot_models(lin_coeff, qua_coeff)
```



the clouds are now less dispersed (more dense around the optimal parameters)

Remember that even if we the estimated model coincides with the best model (e.g., by considering an infinite number of samples), we are not able to reduce the error on newly seen points to zero due to the irreducible error σ^2.

Bias-Variance Decomposition

At last, we extract a *new point* and evaluate its bias and variance, by relying on the **bias-variance decomposition**. By considering the expected square error on an **unseen sample** x, we obtain:

noise (irreducible) — variance of the model (= variance of predictions) — bias²

$$\mathbb{E}_D[(t(x) - y(x))^2] = \sigma^2 + Var_D[y(x)] + \mathbb{E}_D[f(x) - y(x)]^2, \quad (4)$$

expectation taken w.r.t. all the dataset that we can obtain

where we recall that f(x) is the real function generating the samples and the expectation is performed over the different realizations of the dataset.

In [13]:
```
n_points = 1
x_new = np.random.uniform(low=0, high=5, size=(n_points, 1))

# Compute target and add noise
t_new = fun(x_new)
t_new_noisy = t_new + eps * np.random.randn(n_points, 1)

# Compute features values
x_enh_new = np.array([1, x_new, 0])
phi_enh_new = np.array([1, x_new, x_new**2])

# Predict target using the two models
y_pred_lin = lin_coeff @ x_enh_new
y_pred_qua = qua_coeff @ phi_enh_new
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:9: VisibleDeprecationWarning: Creating an nd
array from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differen
t lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creatin
g the ndarray
  if __name__ == '__main__':
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differe
nt lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creati
ng the ndarray
  # Remove the CWD from sys.path while we load stuff.
```

$$\mathbb{E}[(t(x) - y(x))^2] \quad (5)$$

In [14]:
```
# we are taking the mean w.r.t. the different datatsets
error_lin = np.mean((t_new_noisy - y_pred_lin)**2) # square is inside here
```

$$\mathbb{E}[f(x) - y(x)]^2 \quad (6)$$

In [15]:
```
bias_lin = np.mean(t_new - y_pred_lin)**2 # square is outside here
```

$$Var[y(x)] \quad (7)$$

In [16]:
```
variance_lin = np.var(y_pred_lin)
```

$$\sigma^2 = \mathbb{E}[(t(x) - y(x))^2] - Var[y(x)] - \mathbb{E}[f(x) - y(x)]^2, \quad (8)$$

In [17]:
```
var_t_lin = error_lin - variance_lin - bias_lin

# we repeat the same formulas for the quadratic case
error_qua = np.mean((t_new_noisy - y_pred_qua)**2)
bias_qua = np.mean(t_new - y_pred_qua)**2
variance_qua = np.var(y_pred_qua)
var_t_qua = error_qua - variance_qua - bias_qua

print('---Single Point---')
```

```
print('Linear error: {}'.format(error_lin[0][0]))
print('Linear bias: {}'.format(bias_lin[0][0]))
print('Linear variance: {}'.format(variance_lin[0][0]))
print('Linear sigma: {} (unreliable)'.format(var_t_lin[0][0]))

print('---Single Point---')
print('Quadratic error: {}'.format(error_qua[0][0]))
print('Quadratic bias: {}'.format(bias_qua[0][0]))
print('Quadratic variance: {}'.format(variance_qua[0][0]))
print('Quadratic sigma: {} (unreliable)'.format(var_t_qua[0][0]))
```

```
---Single Point---
Linear error: 0.16829617648712575
Linear bias: 0.018256051122378655
Linear variance: 5.8549036527774245e-05
Linear sigma: 0.14998157632821932 (unreliable)
---Single Point---
Quadratic error: 0.07514463485348875
Quadratic bias: 1.1827490508743934e-06
Quadratic variance: 8.764588768967725e-05
Quadratic sigma: 0.07505580621674819 (unreliable)
```

In this case, the estimation of the variance σ^2 does not make sense since has been performed on a single sample.

Similarly, we can compute the bias and variance of the two models by integrating **over the entire input space** $[0, 5]$ and have a more stable estimate of the bias, variance, and irreducible error:

In [18]:
```
n_points = 101

x_new = np.linspace(0, 5, n_points)
t_new = fun(x_new)
t_new_noisy = t_new + eps * np.random.randn(n_points, 1).flatten()

x_enh_new = np.stack([np.ones(n_points), x_new, np.zeros(n_points)])
phi_enh_new = np.stack([np.ones(n_points), x_new, x_new**2])

y_pred_lin = lin_coeff @ x_enh_new
y_pred_qua = qua_coeff @ phi_enh_new
```

In [19]:
```
y_pred_lin.shape
```

Out[19]:
```
(100, 101)
```

In [20]:
```
# replicate the new points for each model
error_lin = np.mean((np.tile(t_new_noisy, (n_repetitions,1)) - y_pred_lin)**2)
bias_lin = np.mean(np.mean(np.tile(t_new, (n_repetitions,1)) - y_pred_lin, axis=0)**2) # I have to take
variance_lin = np.mean(np.var(y_pred_lin, axis=0)) # same here
var_t_lin = error_lin - variance_lin - bias_lin

error_qua = np.mean((np.tile(t_new_noisy, (n_repetitions,1)) - y_pred_qua)**2)
bias_qua = np.mean(np.mean(np.tile(t_new, (n_repetitions,1)) - y_pred_qua, axis=0)**2)
variance_qua = np.mean(np.var(y_pred_qua, axis=0))
var_t_qua = error_qua - variance_qua - bias_qua

print('---All dataset---')
print('Linear error: {}'.format(error_lin))
print('Linear bias: {}'.format(bias_lin))
print('Linear variance: {}'.format(variance_lin))
print('Linear sigma: {}'.format(var_t_lin))

print('---All dataset--')
print('Quadratic error: {}'.format(error_qua))
print('Quadratic bias: {}'.format(bias_qua))
print('Quadratic variance: {}'.format(variance_qua))
print('Quadratic sigma: {}'.format(var_t_qua))
```

```
---All dataset---
Linear error: 0.510381557725224
Linear bias: 0.03614256627373918
Linear variance: 0.00010170007610452258
Linear sigma: 0.47413729137538024
---All dataset--
Quadratic error: 0.4922498188022058
Quadratic bias: 1.924361701616088e-06
Quadratic variance: 0.00013919715198934196
Quadratic sigma: 0.49210869728851486
```

cool! we expected this because we created data with $\sigma = 0.7$ (here it's σ^2)

With this procedure we are able to check that the L_2 is generally able to reduce bias and variance sum, but it is not able to get a null irreducible error.

Remark 1. In real situations we do not know the real model generating data. Therefore, the analysis we conducted before is not a viable option. Nonetheless, the phenomena we analysed are still valid, since in real cases we will have a single point in the parameter space, which will be distributed according to the distribution induced by the model we selected

Dealing with the Bias-Variance Tradeoff

In previous analysis, we made the assumption to know the real model, which is not an assumption we can do in real scenarios. To deal with the bias/variance dilemma in *real scenarios* we use a set of techniques able to take into account the bias/variance tradeoff without having the necessity to know the *real* model.

At first, we want to generate a new dataset:

In [21]:
```
n_tot = 40
```

```
eps = 2

def fun(x):
  return (0.5 - x) * (5 - x) * (x -3)

x = np.random.uniform(low=0, high=5, size=(n_tot, 1))
y = fun(x) + eps * np.random.randn(n_tot, 1)
```

We divide the dataset into:

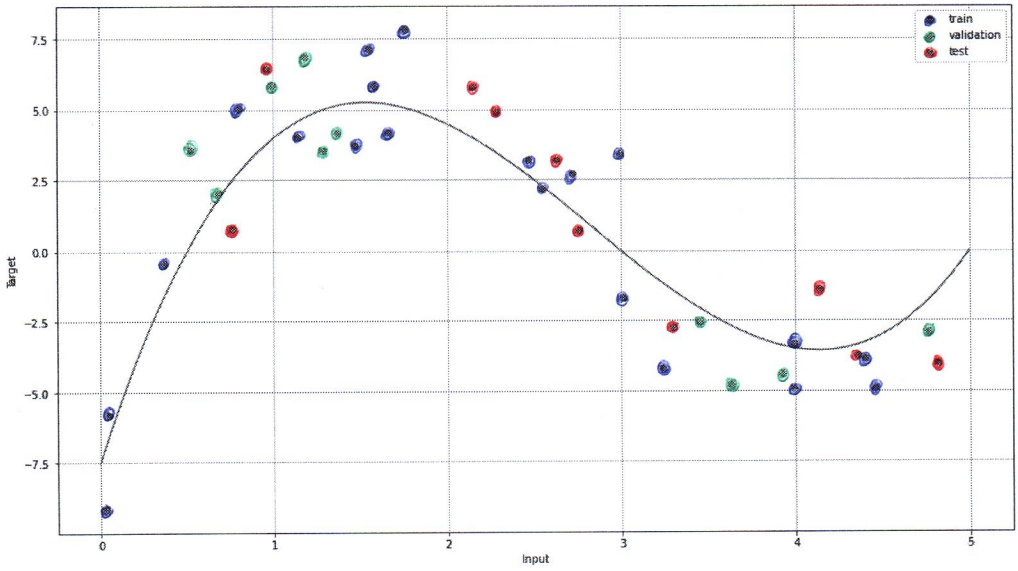
- Training set X_{train}, i.e., the data we will use to learn the model parameters;
- Validation set X_{vali}, i.e., the data we will use to select the model;
- Test set X_{test}, i.e., the data we will use to test the performance of our model.

Usually, we use a split proportional to 50%/25%/25% for the three sets.

In [22]:
```
n_train = 20
n_vali = 10
n_test = 10

x_train, y_train = x[:n_train], y[:n_train]
x_vali, y_vali = x[n_train:n_train + n_vali], y[n_train:n_train + n_vali]
x_test, y_test = x[n_train + n_vali:], y[n_train + n_vali:]
```

normally we should perform a shuffling, here we don't need it because we know that data are generated random in all ways

In [23]:
```
plt.figure(figsize=(16,9))
plt.scatter(x_train, y_train, label='train')
plt.scatter(x_vali, y_vali, label='validation')
plt.scatter(x_test, y_test, label='test')

plt.plot(np.linspace(0,5, 100), fun(np.linspace(0,5,100)))
plt.grid()
plt.legend()
plt.xlabel('Input')
plt.ylabel('Target')
plt.show()
```



Here we are not required to shuffle the data since we generate the input at random, but, in general, it is a good practice to randomly rearrange the data before splitting, since some ordering might be induced by those who provided you the data (e.g., enforced by some query on a database). Here, we use:

- Hypothesis space: $y_n = f(x_n, w) = \sum_{k=0}^{o} x_n^k w_k$;
- Loss measure: $RSS(w) = \sum_n (y_n - t_n)^2$
- Optimization method: Least Square (LS) method;

and we want to select among polynomial models with order $o \in \{0, \ldots, 9\}$. If we compute the training error on different models we have:

In [24]:
```
# We employ this useful function from scikit-learn to produce polynomial features

from sklearn.preprocessing import PolynomialFeatures

degree = 3
poly = PolynomialFeatures(degree=degree)
x_train_tr = poly.fit_transform(x_train)
print(x_train_tr)
```

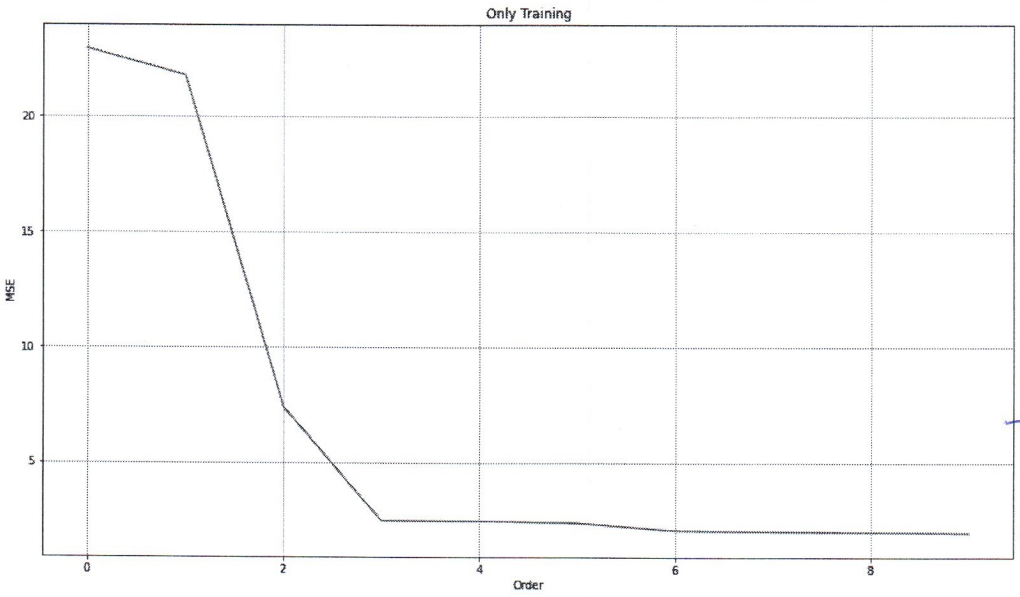
```
[[1.00000000e+00 1.76251963e+00 3.10647544e+00 5.47522393e+00]
 [1.00000000e+00 1.47116833e+00 2.16433625e+00 3.18410295e+00]
 [1.00000000e+00 2.99021936e+00 8.94141182e+00 2.67367827e+01]
 [1.00000000e+00 2.72103678e+00 7.40404114e+00 2.01466683e+01]
 [1.00000000e+00 2.55151932e+00 6.51025086e+00 1.66110309e+01]
 [1.00000000e+00 3.98921246e+00 1.59138161e+01 6.34835935e+01]
 [1.00000000e+00 2.46906319e+00 6.09627306e+00 1.50520834e+01]
 [1.00000000e+00 1.66058141e+00 2.75753061e+00 4.57910406e+00]
 [1.00000000e+00 1.55165909e+00 2.40764593e+00 3.73584569e+00]
 [1.00000000e+00 3.68169014e-01 1.35548423e-01 4.99047291e-02]
 [1.00000000e+00 3.99063723e+00 1.59251855e+01 6.35516382e+01]]
```

1 x x^2 x^3

```
[1.00000000e+00 7.98226371e-01 6.37165340e-01 5.08602177e-01]
[1.00000000e+00 4.64483795e-02 2.15745196e-03 1.00210147e-04]
[1.00000000e+00 1.58293370e+00 2.50567909e+00 3.96632387e+00]
[1.00000000e+00 3.24156959e+00 1.05077734e+01 3.40616789e+01]
[1.00000000e+00 4.45488621e+00 1.98460111e+01 8.84117212e+01]
[1.00000000e+00 3.00905320e+00 9.05440117e+00 2.72451748e+01]
[1.00000000e+00 1.13922292e+00 1.29782887e+00 1.47851640e+00]
[1.00000000e+00 4.39895792e+00 1.93508307e+01 8.51234901e+01]
[1.00000000e+00 2.78675587e-02 7.76600826e-04 2.16419691e-05]]
```

In [25]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error

models = [] # models are stored here
models_mse = [] # mse are stored here

for degree in range(0, 10):
  # fit a linear model for each of the selected polynomial orders
  lin_model = linear_model.LinearRegression()
  poly = PolynomialFeatures(degree=degree)
  x_train_tr = poly.fit_transform(x_train)
  lin_model.fit(x_train_tr, y_train)

  # measure model performance
  y_train_pred = lin_model.predict(x_train_tr)
  mse = mean_squared_error(y_train, y_train_pred)

  # store
  models.append(lin_model)
  models_mse.append(mse)
```

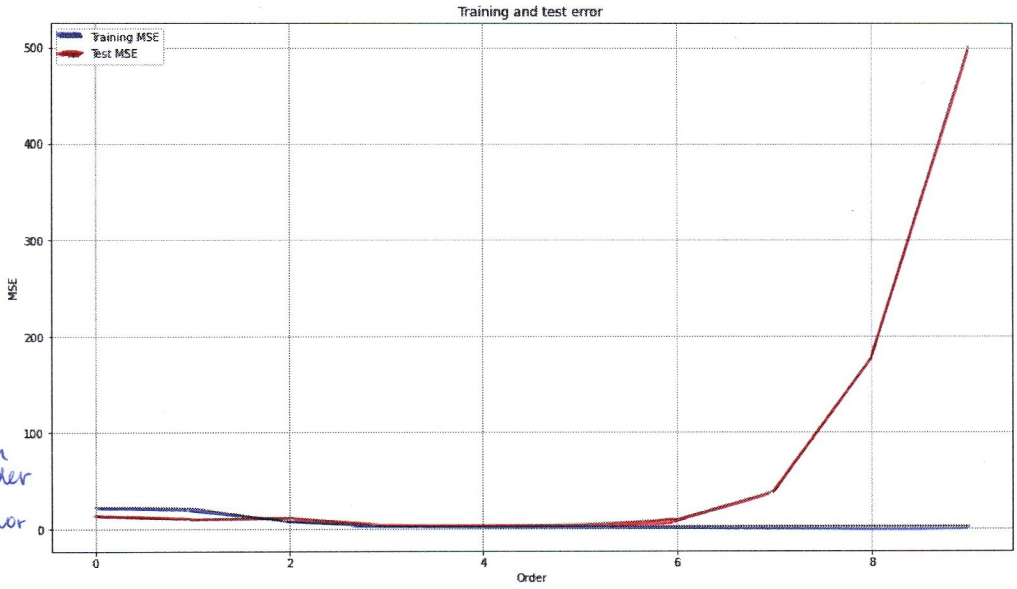
In [26]:
```
plt.figure(figsize=(16,9))
plt.plot(models_mse)
plt.title('Only Training')
plt.grid()
plt.xlabel('Order')
plt.ylabel('MSE')
plt.show()
```



increasing the complexity of the model leads to a decrease of the training error (what about overfitting?)

We can see that this model does not equally perform on a test set: with:

In [27]:
```
models_mse_test = []
for degree in range(0, 10):
  # load the fitted model
  lin_model = models[degree]

  # compute polynomial features for the test
  poly = PolynomialFeatures(degree=degree)
  x_test_tr = poly.fit_transform(x_test)

  # measure model performance on TEST
  y_test_pred = lin_model.predict(x_test_tr)
  mse = mean_squared_error(y_test, y_test_pred)
  models_mse_test.append(mse)

plt.figure(figsize=(16,9))
plt.plot(models_mse, label='Training MSE')
plt.plot(models_mse_test, label='Test MSE')
plt.title('Training and test error')
plt.grid()
plt.xlabel('Order')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

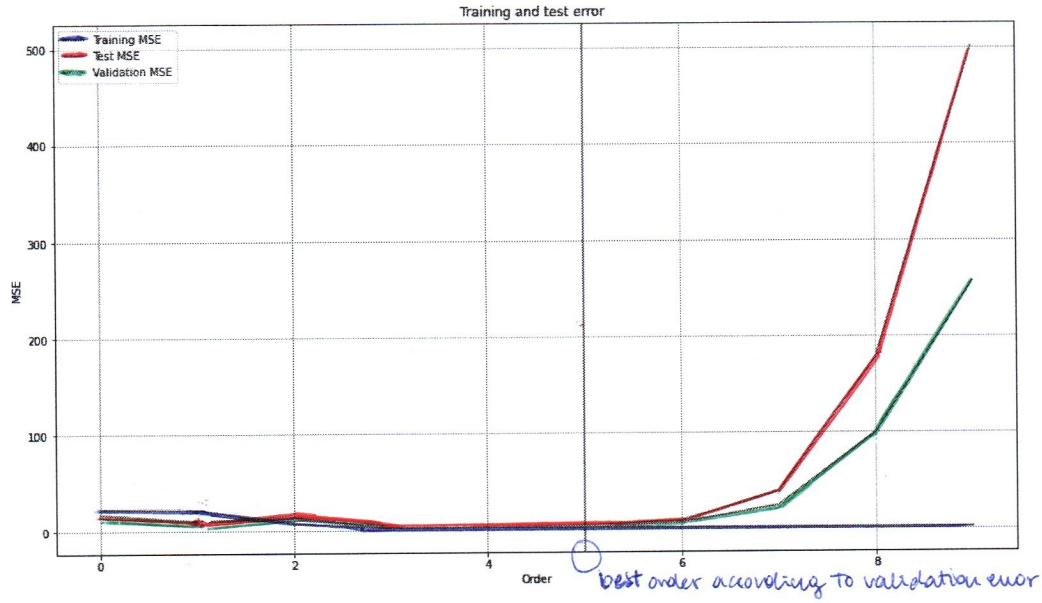


the increment of the test error is due to the overfitting effect

What can we do?
We can use the validation set to select the best order (we select the models for which the validation error is minimum)

To deal with this problem we will use a validation set and tune the order o of the polynomial on that set:

In [28]:
```
models_mse_vali = []
for degree in range(0, 10):
  # load the fitted model
  lin_model = models[degree]

  # compute polynomial features for the test
  poly = PolynomialFeatures(degree=degree)
  x_vali_tr = poly.fit_transform(x_vali)
  y_vali_pred = lin_model.predict(x_vali_tr)

  # measure model performance on TEST
  mse = mean_squared_error(y_vali, y_vali_pred)
  models_mse_vali.append(mse)

plt.figure(figsize=(16,9))
plt.plot(models_mse, label='Training MSE')
plt.plot(models_mse_test, label='Test MSE')
plt.plot(models_mse_vali, label='Validation MSE')

best_model = np.argmin(models_mse_vali)
plt.axvline(best_model)
plt.title('Training and test error')
plt.grid()
plt.xlabel('Order')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



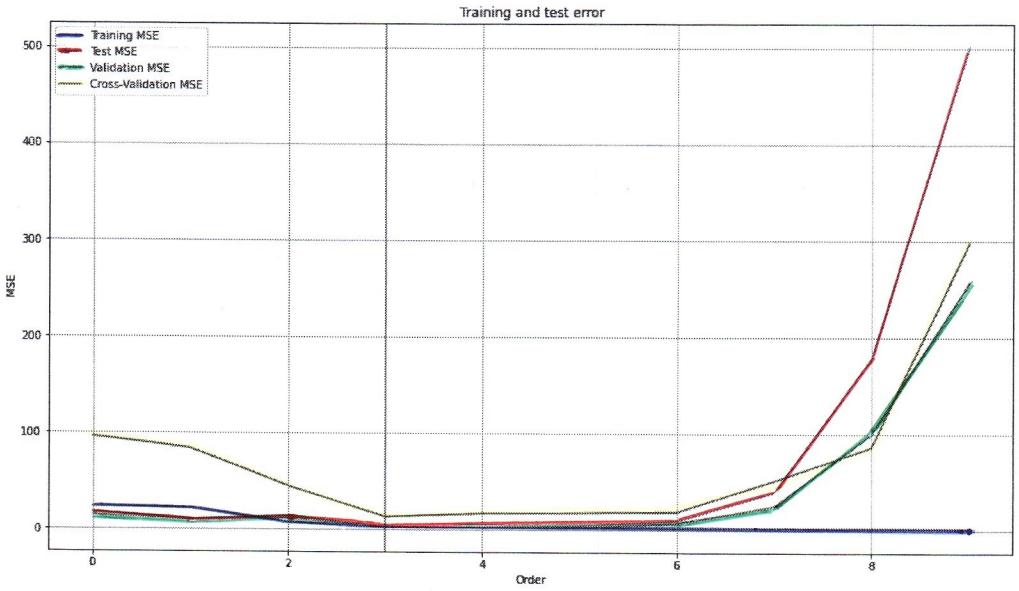
However, the results of this procedure are strongly dependent on the validation set we used. Moreover, we are not using some of the samples during the training, which could improve the model accuracy.

To better exploit the available data, we could resort to the use of **cross-validation**. This way, we have to join the Training and Validation set and divide it in k equally long partitions and use sequentially $k-1$ of them as training set and the remaining one as validation set. In the end, we average the results.

In [29]:
```
k_fold = 4
n_cross = n_train + n_vali
x_cross, y_cross = x[:n_cross], y[:n_cross]
```

```
cross_indices = list(range(len(x_cross)))
MSE_cross = np.zeros(10)
for degree in range(10):
  # divide data
  for k in range(k_fold):
    start_vali = int(np.round(n_cross * k / k_fold)) # start of the validation fold
    end_vali = int(np.round(n_cross * (k+1) / k_fold)) # end of the validation fold

    vali_indices = list(range(int(start_vali), int(end_vali)))
    train_indices = [i for i in cross_indices if i not in vali_indices] # train on all the other indices

    x_fold_train = x_cross[train_indices]
    y_fold_train = y_cross[train_indices]

    x_fold_vali = x_cross[vali_indices]
    y_fold_vali = y_cross[vali_indices]

    # train a model with polynomial feature
    lin_model = linear_model.LinearRegression()
    poly = PolynomialFeatures(degree=degree)
    x_train_tr = poly.fit_transform(x_fold_train)
    x_vali_tr = poly.fit_transform(x_fold_vali)
    lin_model.fit(x_train_tr, y_fold_train)

    # measure cross_validation error
    y_vali_pred = lin_model.predict(x_vali_tr)
    MSE = mean_squared_error(y_fold_vali, y_vali_pred)
    MSE_cross[degree] += MSE
```

In [30]:
```
plt.figure(figsize=(16,9))
plt.plot(models_mse, label='Training MSE')
plt.plot(models_mse_test, label='Test MSE')
plt.plot(models_mse_vali, label='Validation MSE')
plt.plot(MSE_cross, label='Cross-Validation MSE')

best_model = np.argmin(MSE_cross)
plt.axvline(best_model)
plt.title('Training and test error')
plt.grid()
plt.xlabel('Order')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



The value of the error in the optimal point may be greater than the actual error we have on a test set. If we want to perform the Leave One Out (LOO), we just have to modify the number of folds we consider in the cross-validation procedure, i.e., $\mathrm{k\_fold} = 40$. (because we need points for the test)

This kind of procedure is less biased than cross-validation, but requires more computational time.

Homeworks

Complexity-Adjusted Model Evaluation

Compute the C_p, AIC, BIC and adjusted-$R2$ indexes for the dataset generated in the last part of the lecture (training + validation).

Implementing Adaboost

Implement the AdaBoost meta-algorithm for linear regression and by using a weighted re-sampling technique to create each model on the data we considered in the last part of the lecture.

Bootstrapping from generated data: Naive Bayes

Let's go back to the previous exercise session. Using Naive Bayes we know hot to generate new sample, thus we can arbitrarily enlarging our training set.

What happens if we train a new model with also the generated data?

Try to confirm your intuitions with this experiment:

- divide the dataset in train and test
- train a Naive Bayes model with the training part
- generate new samples with the generative model
- train another Naive Bayes model with the enlarged training set
- test both models on the training set
- compute bias, variance and irreducibile error, using the formula when needed

4 Bias-Variance Dilemma

Exercise 4.1

While you fit a Linear Model to your data set. You are thinking about changing the Linear Model to a Quadratic one (i.e., a Linear Model with quadratic features $\phi(x) = [1, x, x^2]$). Which of the following is most likely true:

1. Using the Quadratic Model will decrease your Irreducible Error;
2. Using the Quadratic Model will decrease the Bias of your model;
3. Using the Quadratic Model will decrease the Variance of your model;
4. Using the Quadratic Model will decrease your Reducible Error.

Provide motivations to your answers.

Exercise 4.2

Which of the following is/are the benefits of the sparsity imposed by the Lasso?

1. Sparse models are generally more easy to interpret;
2. The Lasso does variable selection by default;
3. Using the Lasso penalty helps to decrease the bias of the fits;
4. Using the Lasso penalty helps to decrease the variance of the fits.

Provide motivation for your answer.

Exercise 4.3

We estimate the regression coefficients in a linear regression model by minimizing ridge regression for a particular value of λ. For each of the following, describe the behaviour of the following elements as we increase λ from 0 (e.g., remains constant, increases, decreases, increase and then decrease):

1. The training RSS;

2. The test RSS;
3. The variance;
4. The squared bias;
5. The irreducible error.

Exercise 4.4

Suppose that Figure 4.1 is showing the Test error of K-NN obtained by using different values for K.

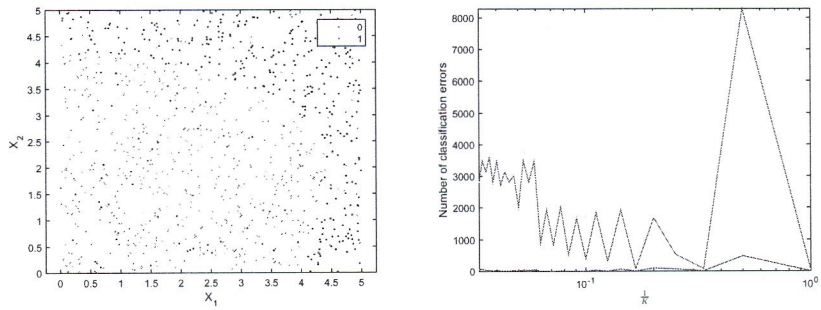


Figure 4.1: Dataset and corresponding error for different K in the K-NN classifier.

Which of the following is most likely true of what would happen to the Test Error curve as we move $\frac{1}{K}$ further above 1?

1. The Test Errors will increase;
2. The Test Errors will decrease;
3. Not enough information is given to decide;
4. It does not make sense to have $\frac{1}{K} > 1$;

Exercise 4.5

Comment on advantages and drawbacks of the following choices:

1. Increase the model complexity and fix number of samples;
2. Increase the number of the samples and fix model complexity.

Exercise 4.6

Assume to have two different linear models working on the same dataset of $N = 100$ samples.

- The first model has $k_1 = 2$ input, considers linear features and has a residual sum of squares of $RSS_1 = 0.5$ on a validation set;
- The second model has $k_2 = 8$ input, considers only quadratic features and has a residual sum of squares of $RSS_2 = 0.3$ on a validation set;

Which one would you choose? Why? Recall that the F-test for statistics for distinguish between linear models is:

$$\hat{F} = \frac{N - p_2}{p_2 - p_1} \frac{RSS_1 - RSS_2}{RSS_2} \sim F(p_2 - p_1, N - p_2),$$

where p_1 and p_2 are the two parameters of the two models and $F(a, b)$ is the Fisher distribution with a and b degrees of freedom.

Exercise 4.7

Which techniques would you consider for model selection in the case we have:

1. A small dataset and a set of simple models;
2. A small dataset and a set of complex models;
3. A large dataset and a set of simple models;
4. A large dataset and a trainer with parallel computing abilities.

Justify you choices.

Exercise 4.8

Suppose you have a dataset and you decided to use all the samples to train your model, including the selection of the parameters of your model and the features you want to consider. What are the problems and issues arising if you use this methodology?

Which procedure a ML scientist should follow?

Answers

Answer of exercise 4.1

1. NO Changing the model does not influence the Irreducible Error.
2. YES We are considering a larger hypothesis space thus it is most likely that the Bias will decrease.
3. NO A more complex model is likely to increase the variance w.r.t. a simpler one.
4. MAYBE If the model is able to reduce the bias and, at the same time, the variance is not increasing too much, we are providing a more accurate model.

Answer of exercise 4.2

1. YES since they provide a clear distinction between those input which are more meaningful (with non–zero parameters) and those which are less relevant (zero parameters);
2. YES since it only includes in the model those input which are meaningful for the model;
3. NO since we are reducing the number of parameters considered in the model, thus the hypothesis space is reduced. This will likely increases the bias of the obtained model;
4. YES since the regularization action helps in avoid overfitting and using unnecessary complex models.

Answer of exercise 4.3

1. INCREASES: by increasing λ, we are forced to use simpler models. This means that training RSS will steadily increase because we are less able to fit the training data exactly.
2. DECRESASES AND THEN INCREASES: at first, the test RSS improves (decreases), because we are less likely to overfit our training data. Eventually, we will start fitting models that are too simple to capture the true effects and the test RSS will start to increase.
3. DECREASES: increasing λ will imply that we are fitting simpler models, which reduces the variance of the fits.

4. INCREASES: by increasing λ we fit simpler models, which likely have larger squared bias.

5. REMAINS CONSTANT: increasing will have no effect on irreducible error, since it has nothing to do with the model we fit.

Answer of exercise 4.4

Clearly the K-NN classifier can only contemplate integer values for the parameter K, thus it does not make sense to decrease the K below 1. The only true statement is the fourth one.

Answer of exercise 4.5

1. Increase the model complexity:

   - Advantages: the Hypothesis space we are considering is larger, thus it may happen that the bias becomes smaller than before.

   - Drawbacks: a more complex model will likely have an increased variance.

2. Increase the number of samples:

   - Advantages: we decrease the variance of the model we are considering.

   - Drawbacks: we need to obtain these samples (which may be expensive) and the training phase might become more and more time consuming (as well as the test one if you are considering non-parametric methods).

Answer of exercise 4.6

The first model has $p_1 = k_1 + 1 = 3$ parameters and the second one $p_2 = k_2 + \frac{k_2(k_2-1)}{2} + 1 = 8 + 28 + 1 = 37$ parameters. To evaluate if two regressive models, under the assumption that the noise is i.i.d. zero mean Gaussian random variable, we can use an F-test. In the specific, we know that the test statistic is:

$$\hat{F} = \frac{N - p_2}{p_2 - p_1} \frac{RSS_1 - RSS_2}{RSS_2} \sim F(p_2 - p_1, N - p_2),$$

is distributed as an F distribution, with $(p_2 - p_1, N - p_2)$ degrees of freedom. Thus, we can evaluate the p-value of the statistic $\hat{F}$ computed on the two aforementioned models:

$$\hat{F} = \frac{100 - 37}{37 - 3} \frac{0.5 - 0.3}{0.3} = 1.2353,$$

whose p-value is 0.2311. Thus there is no statistical evidence at confidence level lower than $\alpha = 0.2311$ that the second model is better than the first one.

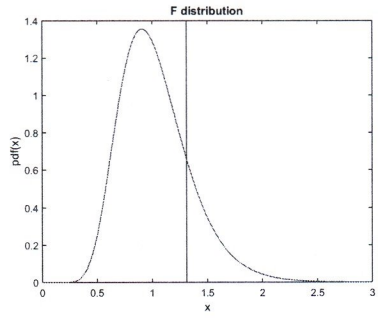


Figure 4.2: Representation of the F-statistic $\hat{F}$ in the case analysed.

Answer of exercise 4.7

1. LOO On a small dataset and for simple models the LOO procedure is not too computationally complex, thus it provides a good approximation (almost unbiased) estimation of the test error.

Complex models and small dataset? Using CV we would reduce the training set and since the dataset is small we cannot afford it (the models are complex, they need data!)

2. AIC (Adjustment techniques) In this case the training on a smaller dataset may lead to overfitting and thus does not provide any information on the model providing good performance on a newly seen samples. If the model is complex, it might be the case that the LOO procedure is still not feasible.

3. CV You should be able to provide a stable estimates to select your model, but at the same time you can not perform LOO for computational complexity reasons.

4. LOO In this case we are able to perform multiple training at the same time, thus the time required for LOO can be reduced by k times, where k is the number of parallel process you can run at the same time.

Answer of exercise 4.8

If we are not considering a validation set to select the model we will most likely select the most complex model among the ones considered, which could lead to overfitting the training set. We could avoid to use a validation set with early stopping techniques (if we are using a gradient descend technique) or by considering an adjustment technique.

Moreover, we do not have any clue about the error we are likely to have in the case of a newly seen data. Thus, we are not able to provide any performance on the goodness of the prediction our model is going to provide.

A proper ML procedure would consider the split of your data into 3 sets (training, validation, test), where the training set is used to estimate the model, the validation set to

select the model and the test set to evaluate the error on unforeseen data. Equivalently, if we use crossvalidation, LOO or adjustment techniques, we should at least save some data to test the performance of the considered method.