Data-driven Design and Analyses of Structures and Materials (3dasm)

Lecture 9

Miguel A. Bessa | **M.A.Bessa@tudelft.nl** | Associate Professor

OPTION 1. Run this notebook **locally in your computer**:

1. Confirm that you have the 3dasm conda environment (see Lecture 1).
2. Go to the 3dasm_course folder in your computer and pull the last updates of the **repository**:

```
git pull
```

3. Open command window and load jupyter notebook (it will open in your internet browser):

```
conda activate 3dasm
jupyter notebook
```

4. Open notebook of this Lecture.

OPTION 2. Use **Google's Colab** (no installation required, but times out if idle):

1. go to **https://colab.research.google.com**
2. login
3. File > Open notebook
4. click on Github (no need to login or authorize anything)
5. paste the git link: **https://github.com/bessagroup/3dasm_course**
6. click search and then click on the notebook for this Lecture.

Outline for today

- Linear models for regression
    - A practical session on how to train linear regression models

Reading material: This notebook + Chapter 11

Today's lecture is going to be more practical

Since we covered the fundamentals of Bayesian and non-Bayesian machine learning...

- Today we will focus on how to train **linear regression models** using **scikit-learn**
- In a later lecture we will derive the models that we are going to cover today.

As we learned in Lecture 2, let's load the pandas dataframe that is in the "docs" folder of Lecture 9:

In [2]:
```
import pandas as pd
# read csv data provided by someone else (this time I also specify that the first column provides the indices)
car_prob_df = pd.read_csv("docs/data_for_car_prob.csv", index_col=0)
print(car_prob_df)
```

```
            x           y
0    9.516939   29.749036
1   72.398757  642.132203
2   17.950326   36.648484
3    9.440853   18.604106
4   78.791008  769.656168
5   16.961121   57.971010
6   65.410368  559.093313
7   58.671099  463.686613
8   21.550603   92.242676
9   36.866913  197.688573
10  15.728748   56.885233
11  58.511494  388.753795
12  57.419190  399.807488
13  38.459157  213.181519
14   8.841742   20.387384
15  60.733051  516.341724
16  49.256663  307.931956
17  35.895121  181.123049
18  79.195652  750.178284
19  69.156669  553.153541
20  77.634896  746.031880
21   9.254011   20.810698
```

```
22  15.451468   39.872527
23  14.438247   42.118771
24  13.410999   44.775122
25  53.747057  375.013937
26  10.283719   19.438868
27  82.005477  742.336845
28  81.805562  706.620282
29  51.837742  345.212876
30  20.283785   65.303165
31  28.359647  155.185137
32  74.993715  676.628982
33  21.827564   81.150935
34  70.519111  700.520033
35  74.208532  622.453560
36  14.518958   40.927570
37  13.357644   39.770922
38  75.346253  707.973754
39  44.923956  251.300805
40  26.801159  124.098654
41  29.906265  118.100900
42  40.226356  215.082100
43  66.282662  537.845048
44  47.342777  308.558833
45   3.087674    5.947997
46  21.254611  101.295276
47  46.939484  345.778352
48  38.875692  219.095582
49  76.705452  742.720134
```

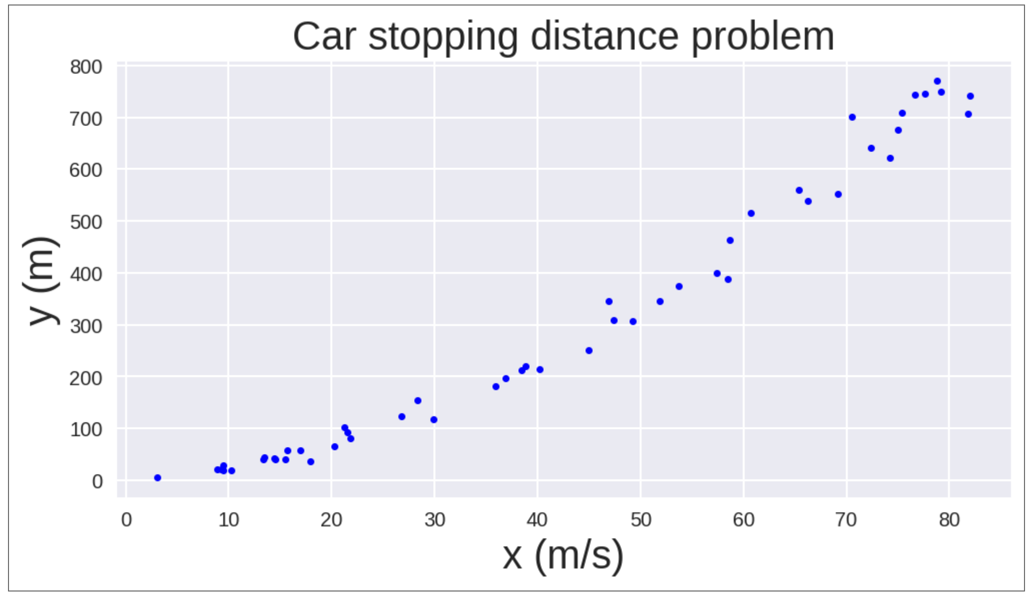
As before, we can separate the data into inputs (features) x and outputs y (targets)

```
In [3]:
Data_x = car_prob_df['x'].values # select the input VALUES from your dataframe into Data_x
Data_y = car_prob_df['y'].values # select the output VALUES from your dataframe inta Data_y
print("Data_x is:\n",Data_x)
print("\nData_y is:\n",Data_y)
```

```
Data_x is:
 [ 9.51693942 72.39875748 17.95032583  9.44085299 78.79100778 16.96112056
 65.4103675  58.67109927 21.55060313 36.86691294 15.72874781 58.51149357
 57.41918959 38.45915667  8.84174221 60.73305107 49.25666345 35.89512052
 79.19565172 69.15666925 77.63489641  9.25401128 15.45146824 14.43824684
 13.41099874 53.74705712 10.28371886 82.00547705 81.80556249 51.8377421
 20.28378484 28.35964692 74.99371524 21.82756352 70.51911096 74.20853195
 14.51895792 13.35764354 75.34625316 44.92395642 26.80115926 29.90626522
 40.22635624 66.28266205 47.34277718  3.08767411 21.25461134 46.93948443
 38.87569199 76.70545196]

Data_y is:
 [ 29.74903647 642.13220315  36.64848446  18.60410602 769.65616843
  57.97101034 559.09331318 463.68661322  92.24267632 197.68857288
  56.88523327 388.75379474 399.80748803 213.18151905  20.38738432
 516.34172363 307.93195589 181.12304936 750.17828361 553.15354059
 746.03187971  20.81069833  39.87252654  42.11877078  44.77512244
 375.01393668  19.43886782 742.33684483 706.62028237 345.21287569
  65.30316533 155.18513747 676.62898211  81.15093549 700.52003305
 622.45356019  40.92757044  39.77092163 707.97375405 251.30080489
 124.09865438 118.10089977 215.08209978 537.84504756 308.55883254
   5.94799685 101.29527607 345.77835213 219.09558165 742.72013356]
```

And we can plot the data:

```
In [4]:
fig_car_data, ax_car_data = plt.subplots() # create a plot
ax_car_data.plot(Data_x, Data_y, 'b.')
ax_car_data.set_xlabel("x (m/s)", fontsize=20) # create x-axis label with font size 20
ax_car_data.set_ylabel("y (m)", fontsize=20) # create y-axis label with font size 20
ax_car_data.set_title("Car stopping distance problem", fontsize=20); # create title with font size 20
```



Supervised learning: regression models

As we have been discussing, when we do regression via supervised learning we want to:

- create a machine learning model
- train it on known data (known inputs x and outputs y)
- predict for new (unseen) data points, i.e. predict y^* for a new value of x.

Today we will talk about the simplest models: **linear regression**.

Linear regression models

Linear regression models encompass a class of machine learning methods that is larger than you might think...

As we will see, despite being called "linear" these models can do more than fitting a simple "line" to our data.

For now, let's consider 1d datasets, i.e. where we have one input x and one output y.

Simplest 1d linear regression model: fitting a line to your data

1. Observation distribution:

Usually, assumed as a Gaussian distribution,

$$p(y|x,\mathbf{z}) = \mathcal{N}(y|\mu_{y|z} = \mathbf{w}^T\boldsymbol{\phi}(x), \sigma^2_{y|z} = \sigma^2)$$

where $\mathbf{z} = (\mathbf{w}, \sigma)$ are all the hidden rv's of the model, i.e. the model parameters.

- the vector $\mathbf{w} = [w_0, w_1]^T$ includes the **bias** term w_0 and the **weight** w_1.
- the vector $\boldsymbol{\phi}(x) = [1, x]^T$ includes the **basis functions**.

1. A chosen Prior distribution on each hidden rv of $\mathbf{z}$:

Usually, the prior on w_0 and σ is the Uniform distribution.
However, the prior on the weight w_1 is often chosen as something else (but it can also be Uniform).

Does this model remind you of something we did?

- Car stopping distance problem when we knew one of the rv's and fixed x!

Note that,

$$p(y|x, \mathbf{z}) = \mathcal{N}(y|\mathbf{w}^T \boldsymbol{\phi}(x), \sigma^2) \tag{1}$$
$$= \mathcal{N}(y|w_0 + w_1 x, \sigma^2) \tag{2}$$

where we previously called $w_0 \equiv b$, $w_1 \equiv z$ and $\sigma^2 \equiv \sigma^2_{y|z}$.

Therefore, the only difference is that we now start to consider more than one rv, so we group them into vector $\mathbf{z} = (\mathbf{w}, \sigma)$.
About notation:

- We can also write $\mathbf{z}^T = [\mathbf{w}^T, \sigma]$.

1d linear regression models with different basis functions: fitting a polynomial to your data

However, we know that usually a straight line does not provide a good fit to most data sets.

A very important realization:

- The basis functions vector $\boldsymbol{\phi}(x)$ does not need to be a *linear transformation*.

- It could be a polynomial or any other **nonlinear** transformation of the feature (input) x.

As long as the parameters of the basis functions vector $\boldsymbol{\phi}(x)$ are **fixed**, the model remains **linear in the parameters**, even if is not linear in the input (feature). That's why we still call this a **linear model**.

Here's how our **linear** regression model looks like for a polynomial basis functions vector $\boldsymbol{\phi}(x)$.

1. Observation distribution:

$$p(y|x,\mathbf{z}) = \mathcal{N}(y|\mu_{y|z} = \mathbf{w}^T\boldsymbol{\phi}(x), \sigma^2_{y|z} = \sigma^2)$$

where $\mathbf{z} = (\mathbf{w}, \sigma)$ are all the hidden rv's of the model, i.e. the model parameters.

- the vector $\mathbf{w} = [w_0, w_1, w_2 \ldots, w_d]^T$ includes the **bias** term w_0 and the remaining **weights** w_i with $i = 1, \ldots, d$.
- the vector $\boldsymbol{\phi}(x) = [1, x, x^2, \ldots, x^d]^T$ includes the **basis functions**, which now correspond to a polynomial of degree d

  .

1. A chosen Prior distribution for each hidden rv of $\mathbf{z}$, as mentioned previously.

Linear regression models from a Bayesian perspective

The choice of likelihood and prior determines what is the linear regression model that you are choosing!

Likelihood	Prior (on the weights)	Posterior	Name of the model	Book section
Gaussian	Uniform	Point estimate	Least Squares regression	11.2.2
Gaussian	Gaussian	Point estimate	Ridge regression	11.3
Gaussian	Laplace	Point estimate	Lasso regression	11.4
Student-t	Uniform	Point estimate	Robust regression	11.6.1
Laplace	Uniform	Point estimate	Robust regression	11.6.2
Gaussian	Gaussian	Gaussian	Bayesian linear regression	11.7

We will derive some of these models in a later class. Today, we focus on the practical aspects!

Training (fitting) a linear model with Scikit-learn

Let's see how to use **scikit-learn** to train linear regression models.

- **scikit-learn** is a well-documented and user-friendly library that is great for introducing machine learning.

- You should really read the documentation.
    - It includes many useful examples.
    - It provides a short introduction to common machine learning algorithms.

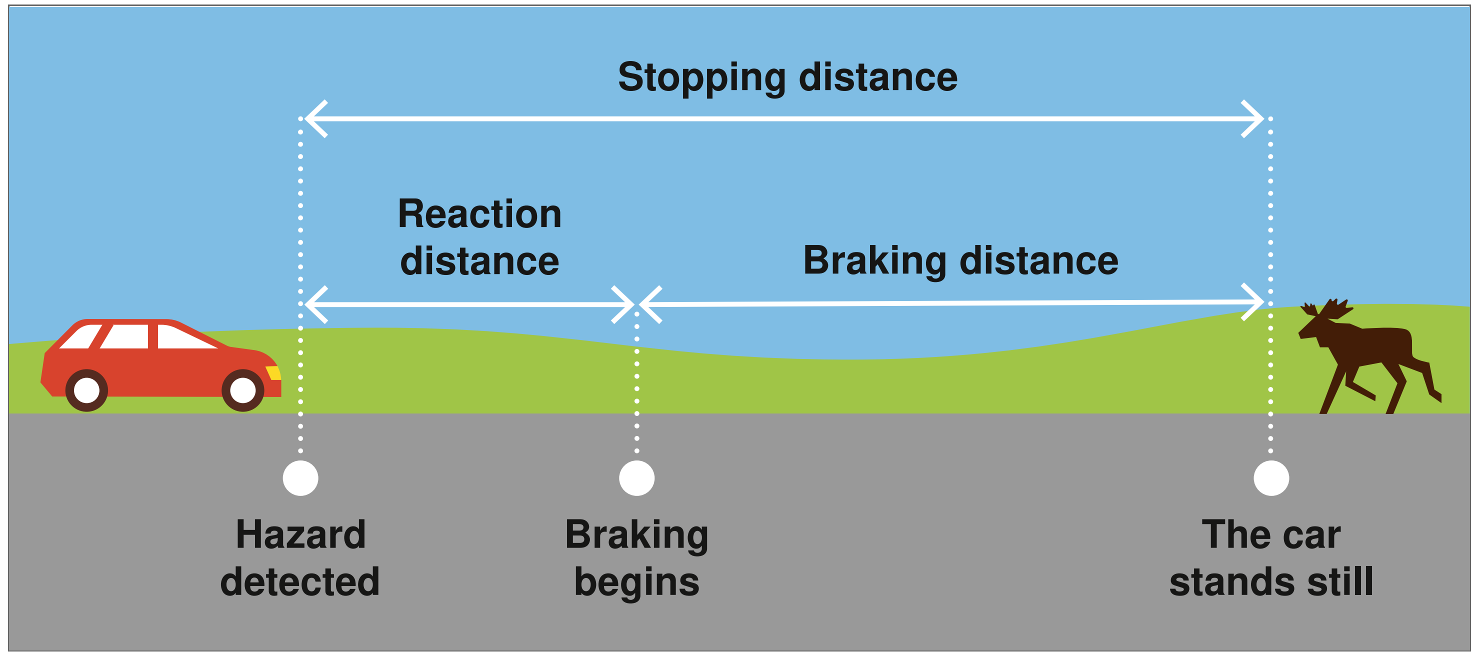
Example 1: training a linear model for the car stopping distance problem

Let's start by importing from scikit-learn the simplest linear regression model:

- Least Squares Regression.

And let's consider the simplest basis function:

- A line (polynomial of degree 1).



In [5]:

```python
from sklearn.linear_model import LinearRegression # For Least Squares Regression
from sklearn.preprocessing import PolynomialFeatures # For Polynomial basis functions
from sklearn.pipeline import make_pipeline # to link different objects
```

Now let's define the model (Least Squares Regression + polynomial of degree 1)

In [6]:

```
              # We start by defining the model (polynomial basis + Least Squares Regression)
degree = 1 # degree of polynomial we want to fit
poly_model = make_pipeline(PolynomialFeatures(degree),LinearRegression())
```

Then we train the model for our data (input Data_x and output Data_y that were loaded with pandas)

In [7]:
```
# Uncomment line below (this is just for students to understand: don't panic when encountering an error!)
#poly_model.fit(Data_x,Data_y) # but it gives an ERROR!
```

This gives an error! Fortunately, scikit-learn tells us what happened...

Scikit-learn expects the inputs to be formatted as a 2D array (matrix), instead of a 1D array (vector). This happens because usually we fit machine learning models for multidimensional inputs.

In [8]:

```
# Reshape the input vector into a 2D array:
Data_X = np.reshape(Data_x, (-1, 1)) # we use capital letters for matrices and lower case for vectors
```

In [9]:

```
print(Data_X)
```

```
[[ 9.51693942]
 [72.39875748]
 [17.95032583]
 [ 9.44085299]
 [78.79100778]
 [16.96112056]
 [65.4103675 ]
 [58.67109927]
 [21.55060313]
 [36.86691294]
 [15.72874781]
 [58.51149357]
 [57.41918959]
 [38.45915667]
 [ 8.84174221]
 [60.73305107]
 [49.25666345]
 [35.89512052]
 [79.19565172]
 [69.15666925]
 [77.63489641]
 [ 9.25401128]
 [15.45146824]
 [14.43824684]
 [13.41099874]
 [53.74705712]
 [10.28371886]
 [82.00547705]
 [81.80556249]
 [51.8377421 ]
 [20.28378484]
 [28.35964692]
 [74.99371524]
 [21.82756352]
 [70.51911096]
 [74.20853195]
 [14.51895792]
 [13.35764354]
 [75.34625316]
 [44.92395642]
 [26.80115926]
 [29.90626522]
 [40.22635624]
 [66.28266205]
 [47.34277718]
 [ 3.08767411]
 [21.25461134]
 [46.93948443]
 [38.87569199]
 [76.70545196]]
```

After reshaping the input as a 2D array, we see that we can fit the model!

```
In [10]: poly_model.fit(Data_X,Data_y) # now we were able to train (fit) our linear model to the data!
```

```
Out[10]:

Pipeline(steps=[('polynomialfeatures', PolynomialFeatures(degree=1)),
                ('linearregression', LinearRegression())])
```

That's it! Here's your first ML model: fitting a straight line 😆

Now that we have a model, we can predict the output y^* for any new input point x^*.
In particular, we can predict the output for each of the input points x that we used for training the model (i.e. at Data_X).

```
In [11]:
y_pred = poly_model.predict(Data_X) # In scikit-learn, predicting from a model is a one-liner
```

Done! These are the predictions for all your training points.

But we can also predict the output y^* for other points.
This enables us to visualize the model by predicting the output for a uniformly spaced set of points.

```
In [12]:
               # Now create linearly spaced points for plotting our linear model
x_plot = np.linspace(0, 90, 200) # 200 points uniformly spaced
y_plot = poly_model.predict(np.reshape(x_plot, (-1, 1))) # prediction of those points (note the reshape again)
```

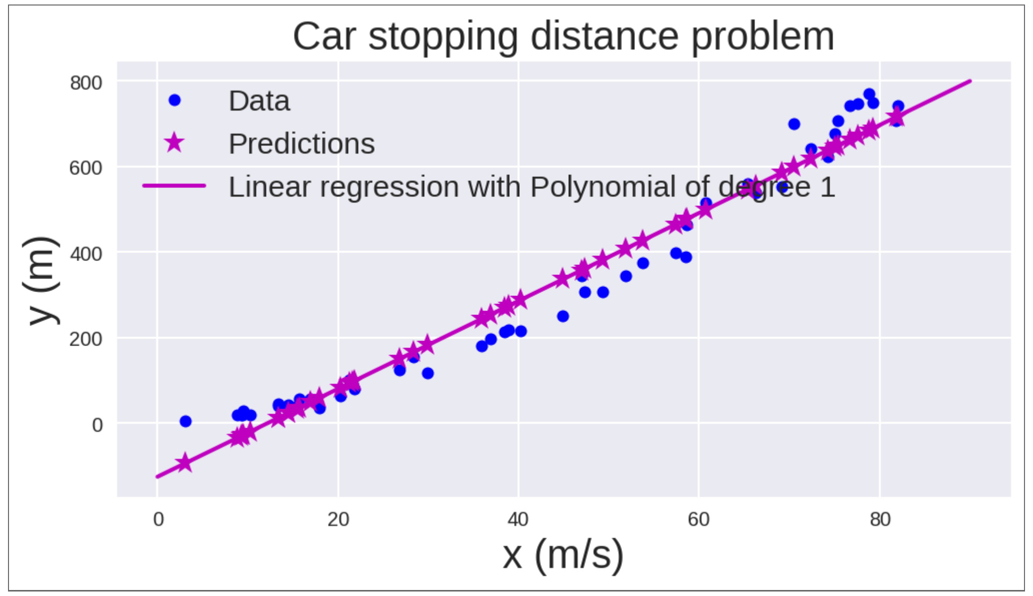
Finally, we can just plot the data, the predictions for each input training point, and the linear model.

In [13]:

```
               # The usual plotting style for this model and the data.
fig_poly, ax_poly = plt.subplots() # create a plot
ax_poly.plot(Data_x, Data_y, 'b.', markersize=12,label="Data") # Markers locating data points)
ax_poly.plot(Data_x, y_pred, 'm*', markersize=12,label="Predictions") # Markers locating prediction points)
legend_str = "Linear regression with Polynomial of degree " + str(degree)
ax_poly.plot(x_plot, y_plot, 'm-', linewidth=2,label=legend_str) # polynomial interpolation plotted
ax_poly.set_xlabel("x (m/s)", fontsize=20) # create x-axis label with font size 20
ax_poly.set_ylabel("y (m)", fontsize=20) # create y-axis label with font size 20
ax_poly.set_title("Car stopping distance problem", fontsize=20) # create title with font size 20
ax_poly.legend(loc='upper left', fontsize=15) # replot legend
plt.close(fig_poly) # do not plot the figure now. We will show it in a later cell
```

In [14]:
```
fig_poly # just call the figure that we created in the previous cell.
```

Out[14]:



Now we can see all the things that we have done in a single plot!

Exercise 1

1. Put it all together and create a linear model using a polynomial of degree 2 for the same data.
1. Compare your plot with the one obtained for the straight line (polynomial of degree 1).
1. Play a bit with your code by changing the degree of the polynomial. What happens?

In [15]:

```
# Write your code for Exercise 1:

# until here.
```

If you keep increasing the polynomial degree, you see that the prediction gets worse!
This is called **overfitting**.

- Overfitting is a natural consequence of having a model that is more complex than it should be!

- As we know, the mean of the data that originates from the car stopping distance problem is generated with a quadratic model: $y = z_1 x + z_2 x^2$
    - Therefore, we don't need additional complexity to describe the *mean* of the data!

Also, the fit is better **within the domain** that we used for training ($x \in [3, 83]$) than away from it.
In other words, we **interpolate** better than we **extrapolate**.

- Nevertheless, overfitting is also an issue when we are interpolating...
    - See this by making the degree very high, e.g. 30, and plot for $x \in [5, 80]$.

It might be surprising, but this is an important issue in ML.
It is very common to use models that are complicated (have many parameters) and that perform poorly even when interpolating, but especially when extrapolating!

Example 2: linear model for a noiseless problem

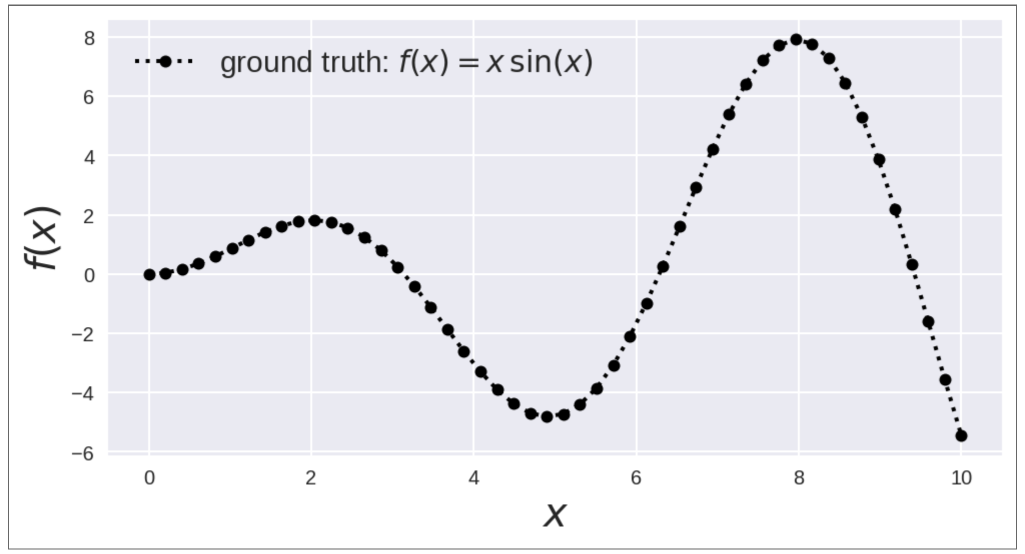
Consider now a problem not governed by a polynomial law and without uncertainty.
Let's consider the function $x\sin(x)$ in the domain $x \in [0, 10]$. We did this in Lecture 2!

In [16]:

```
               # 1. Define the function f(x) = x sin(x)
def f(x):
    return x * np.sin(x)
# 2. Create a vector of 50 points that are uniformly spaced between 0 and 10
n_data = 50 # number of points for plotting the function
x_data = np.linspace(0, 10, n_data) # uniformly spaced points
# 3. Compute the output vector:
y_data = f(x_data)
# 4. Plot the function and the data
fig1, ax1 = plt.subplots() # This opens a new figure
# Plot points and interpolate them:
ax1.plot(x_data, y_data, 'ko:', markersize=6, linewidth=2,label=u'ground truth: $f(x) = x\,\sin(x)$')
ax1.set_xlabel('$x$', fontsize=20) # label of the x axis
ax1.set_ylabel('$f(x)$', fontsize=20) # label of the y axis
ax1.legend(loc='upper left', fontsize=15) # plot legend in the upper left corner
```

Out[16]:

```
<matplotlib.legend.Legend at 0x7f12de025640>
```



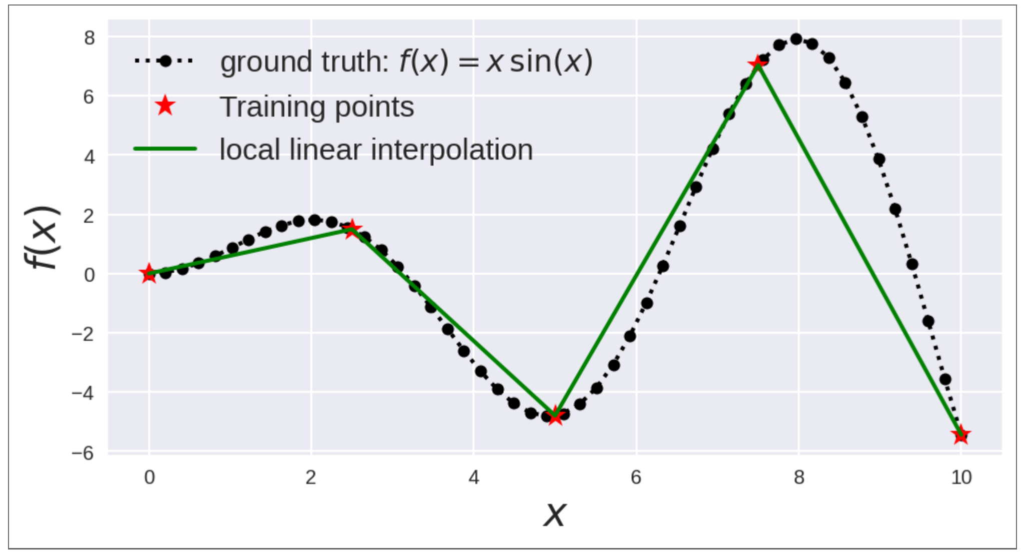
With lots of data, even linear interpolation between the points can approximate the function well.

However, what if we use just a few points from our dataset x_data?

In [17]:

```
n_train = 5 # points to train the algorithm
x_train = np.linspace(0, 10, n_train) # 5 points uniformly distributed
y_train = f(x_train)
ax1.plot(x_train, y_train, 'r*', markersize=12,label="Training points") # Markers locating training points
ax1.plot(x_train, y_train, 'g-', linewidth=2, label=u'local linear interpolation') # linear interpolation
ax1.legend(loc='upper left', fontsize=15) # replot legend
fig1 # replot fig1 now overlaying the plot in the previous cell
```

Out[17]:



This is called local interpolation because each line only depends on the two points it is connecting (not on the other points).

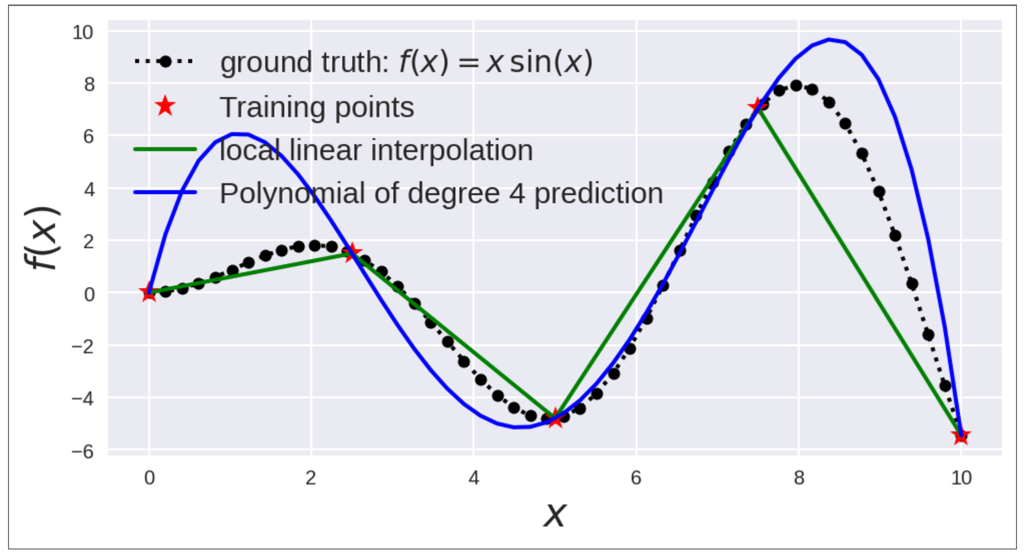
- This is not an ML model! It's different from the linear model we created for the car stopping distance problem.

So, let's train a linear ML model using a polynomial of degree 4 as basis function for this example.

In [18]:

```
degree = 4 # degree of polynomial

poly_model = make_pipeline(PolynomialFeatures(degree), LinearRegression()) # model

X_train = np.reshape(x_train, (-1, 1)) # convert input vector into 2d array

poly_model.fit(X_train,y_train) # fit the polynomial to our 5 training points
#y_pred = poly_model.predict(X_train) # prediction of our polynomial

X_data = np.reshape(x_data, (-1, 1)) # Don't forget to convert to a 2d array
y_pred = poly_model.predict(X_data) # prediction our model for all 50 data points

# Plot x_data and prediction as a blue line:
ax1.plot(x_data, y_pred, 'b-', linewidth=2, label="Polynomial of degree %d prediction" % degree)

# Replot figure and legend:
ax1.legend(loc='upper left', fontsize=15)
fig1
```

Out[18]:



Our polynomial (blue) is clearly different to the function that we want to "learn", i.e. $x\sin(x)$.

How do we evaluate the quality of our approximation?

- By evaluating the error of our polynomial model in the points that we didn't use in the fit.

Two common metrics are R^2 and MSE (you will have to search for them and explain them!)

In [19]:

```
                # Import error metrics:
from sklearn.metrics import mean_squared_error, r2_score

# Compute MSE and R2 for the polynomial model we fitted
mse_value = mean_squared_error(y_data, y_pred)
r2_value = r2_score(y_data, y_pred)

print('MSE for polynomial = ', mse_value)
print('R2 score for polynomial = ', r2_value)
```

```
MSE for polynomial =  5.826961574280784
R2 score for polynomial =  0.578691860751741
```

As expected, these predictions are not great because:

- We want MSE to be as low as possible
- The closer R^2 is to 1.0 the better

You will dive deeper into this when solving the **Midterm Project**.

Example 3: linear model for noisy datasets

Let's consider one last case where we perturb the function of Example 2 $f(x) = x\sin(x)$ with noise.

- This is similar to what happens in the car stopping distance problem.
- Here, the difference is that at every point the noise is random (not as predictable as before)

Let's "fabricate" such dataset.

In [20]:

```
seed = 1987 # set a random seed so that everyone gets the same result
np.random.seed(seed)

# Let's perturb every y_data point with Gaussian noise
random_std = 0.5 + 1.0 * np.random.random(y_data.shape)

# Then, take the random value for STD from 0.5 to 1.5 for each
# data point and create noise following a Gaussian distribution with
# that STD at that point:
noise = np.random.normal(0, random_std)

# The perturbed data becomes:
y_noisy_data = y_data + noise
```

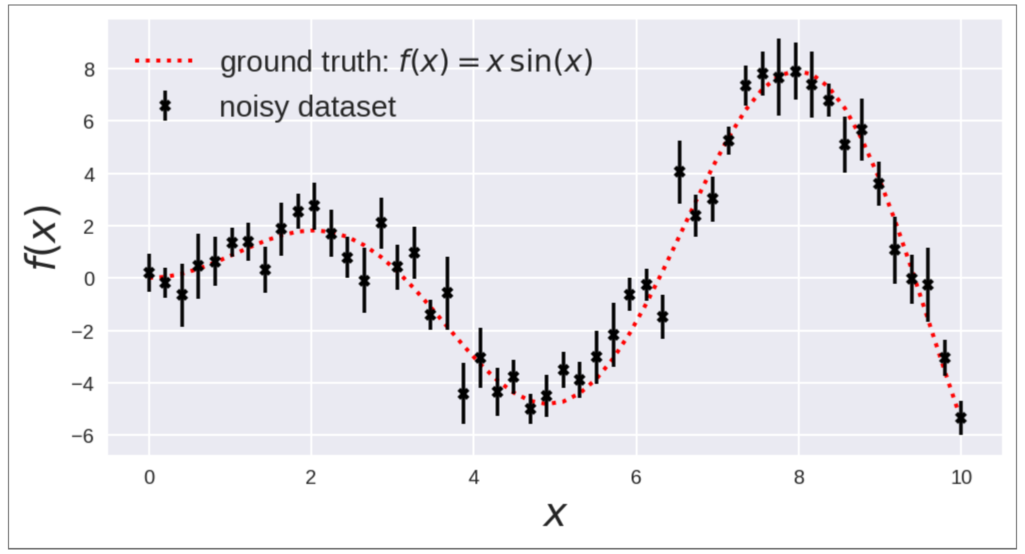
For comparison, we plot the noisy data with the noiseless function that we would like to discover $x\,\sin(x)$:

In [21]:

```
fig2, ax2 = plt.subplots() # This opens a new figure

# Plot the noiseless function ("the ground thruth")
ax2.plot(x_data, y_data, 'r:', linewidth=2,
         label=u'ground truth: $f(x) = x\,\sin(x)$')

# Plot the noisy dataset that we are given:
plt.errorbar(x_data, y_noisy_data, random_std, fmt='kX',
             markersize=6, label=u'noisy dataset')

ax2.set_xlabel('$x$', fontsize=20) # label of the x axis
ax2.set_ylabel('$f(x)$', fontsize=20) # label of the y axis
ax2.legend(loc='upper left', fontsize=15) # plot legend in the upper left corner
```

Out[21]:

```
<matplotlib.legend.Legend at 0x7f12dde71b20>
```



Note a couple of things:

- The black "x" marks the actual measured value.
- The black bars indicate the noise in each data point (each data point has a different noise value). Formally, we call this aleatoric uncertainty.
    - In the plot, I centered the error bars around the measured value. Note that the bars do not have the same length (different standard deviation).

A note on data preprocessing

Usually, when we are given a dataset we need to find a way to **train** and **test** our model using that data.
However, to test our model we have to use data that we have not used in training, otherwise we would be cheating!
This is done by splitting the dataset (in this case x_data) into two sets:

1. **Training** set (for example: 75% of the dataset)

1. **Test** set with the remaining points of the dataset

Scikit-learn has a very easy way of doing this:

In [22]:

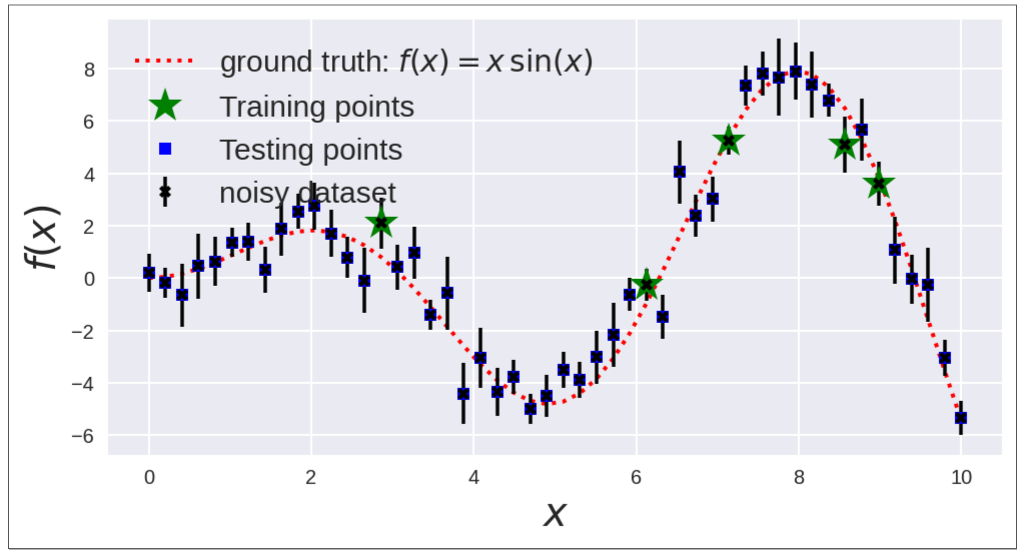
```
from sklearn.model_selection import train_test_split

X_data = np.reshape(x_data,(-1,1)) # a 2D array that scikit-learn likes
# Let's split the data points into 10% for the training set (5 points)
# and the rest for the test set:
X_train, X_test, y_train, y_test = train_test_split(X_data,
                                    y_noisy_data, test_size=0.90,
                                    random_state=seed)
```

Note that the train_test_split module of scikit-learn picks points pseudo-randomly according to the random_state seed value. Let's visualize the training and testing sets:

In [23]:

```
x_train = X_train[:] # converting back to 1D array for plotting
x_test = X_test[:] # converting back to 1D array for plotting
# Plot the noisy training dataset:
ax2.plot(x_train, y_train, 'g*', markersize=18, label="Training points") # Markers locating training points
ax2.plot(x_test, y_test, 'bs', markersize=6,label="Testing points") # Markers locating training points
ax2.set_xlabel('$x$', fontsize=20) # label of the x axis
ax2.set_ylabel('$f(x)$', fontsize=20) # label of the y axis
ax2.legend(loc='upper left', fontsize=15) # plot legend in the upper left corner
fig2
```

Out[23]:



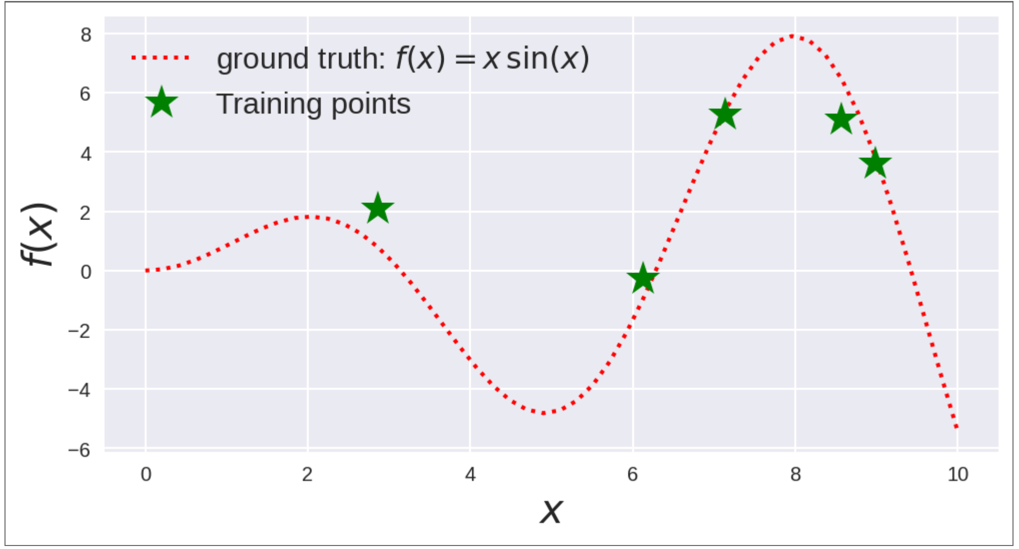
Let's create a new figure with less clutter by just plotting the ground truth function and the training points.

In [24]:

```
fig3, ax3 = plt.subplots() # This opens a new figure

# Plot the noiseless function ("the ground thruth")
ax3.plot(x_data, y_data, 'r:', linewidth=2, label=u'ground truth: $f(x) = x\,\sin(x)$')
ax3.plot(x_train, y_train, 'g*', markersize=18, label="Training points") # Markers locating training points
ax3.set_xlabel('$x$', fontsize=20) # label of the x axis
ax3.set_ylabel('$f(x)$', fontsize=20) # label of the y axis
ax3.legend(loc='upper left', fontsize=15) # plot legend in the upper left corner
```

Out[24]:

```
<matplotlib.legend.Legend at 0x7f12ddea9100>
```



Exercise 2

Fit a polynomial of degree 4 to this training data and calculate the R^2 and MSE metrics for the testing data.

In [25]:

```
# Write your code for Exercise 2:

# until here.
```

Well done...

Yet, this does not seem like a great result, does it?

The R^2 value is so bad that it is even negative!

- What explains this result?
- Can we do something to fix this while still using polynomials?
- If we used more points would that help?
- What if we increased the degree of the polynomial?

You will explore these and other things in Part 1 of the Midterm Project…

Have fun!