Homework 4

References

- Lectures 13-16 (inclusive).

Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

Student details

- **First Name:** Kyle
- **Last Name:** Illenden
- **Email:** killende@purdue.edu

Problem 1 - Estimating the mechanical properties of a plastic material from molecular dynamics simulations

First, make sure that this dataset is visible from this Jupyter notebook. You may achieve this by either:

- Downloading the data file and then manually upload it on Google Colab. The easiest way is to click on the folder icon on the left of the browser window and click on the upload button (or drag and drop the file). Some other options are here.
- Downloading the file to the working directory of this notebook with this code:

In [ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecture
download(url)
```

It's up to you what you choose to do. If the file is in the right place, the following code should work:

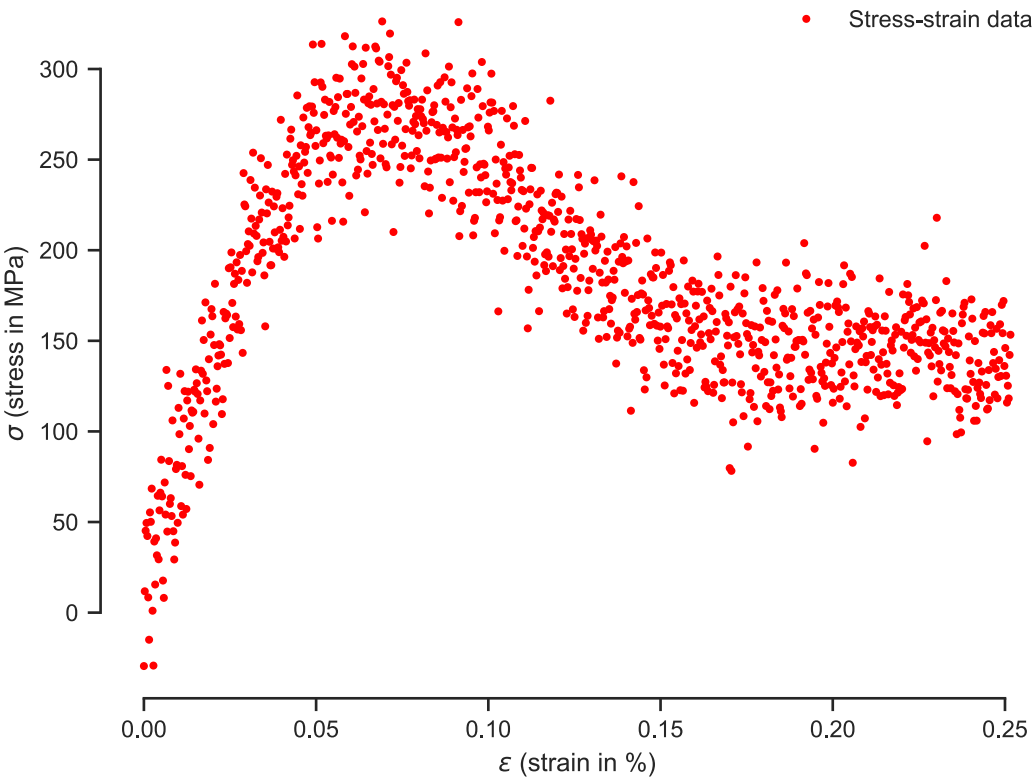
In [ ]:
```
data =  np.loadtxt('stress_strain.txt')
```

The dataset was generated using a molecular dynamics simulation of a plastic material (thanks to Professor Alejandro Strachan for sharing the data!). Specifically, Strachan's group did the following:

- They took a rectangular chunk of the material and marked the position of each one of its atoms;
- They started applying a tensile force along one dimension. The atoms are coupled together through electromagnetic forces, and they must all satisfy Newton's law of motion.
- For each value of the applied tensile force, they marked the stress (force be unit area) in the middle of the material and the corresponding strain of the material (percent elongation in the pulling direction).
- Eventually, the material entered the plastic regime and broke. Here is a visualization of the data:

In [ ]:
```
# Strain
x = data[:, 0]
```

```
# Stress in MPa
y = data[:, 1]

plt.figure()
plt.plot(
    x,
    y,
    'ro',
    markersize=2,
    label='Stress-strain data'
)
plt.xlabel('$\epsilon$ (strain in %)')
plt.ylabel('$\sigma$ (stress in MPa)')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



Note that you don't necessarily get a unique stress for each particular value of the strain. This is because the atoms are jiggling around due to thermal effects. So, there is always this "jiggling" noise when measuring the stress and the strain. We want to process this noise to extract what is known as the stress-strain curve of the material. The stress-strain curve is a macroscopic property of the material, affected by the fine structure, e.g., the chemical bonds, the crystalline structure, any defects, etc. It is a required input to the mechanics of materials.

Part A - Fitting the stress-strain curve in the elastic regime

The very first part of the stress-strain curve should be linear. It is called the *elastic regime*. In that region, say $\epsilon < \epsilon_l = 0.04$, the relationship between stress and strain is:

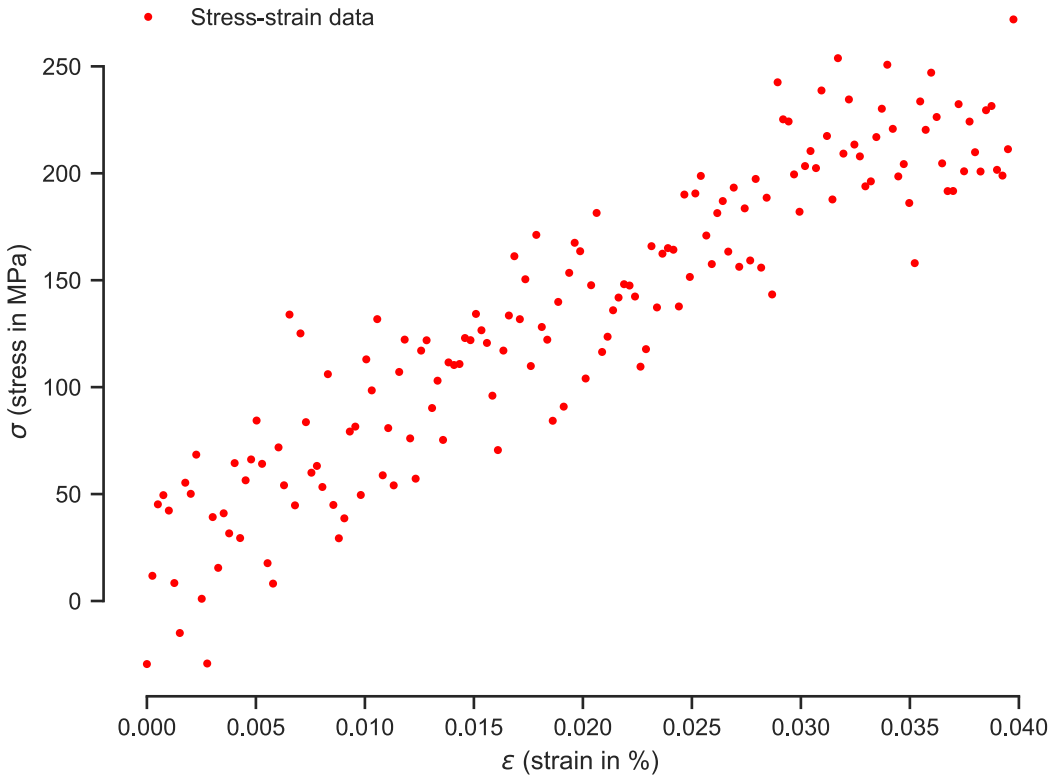
$$\sigma(\epsilon) = E\epsilon.$$

The constant E is known as the *Young modulus* of the material. Assume that you measure ϵ without noise, but your measured σ is noisy.

Subpart A.I

First, extract the relevant data for this problem, split it into training and validation datasets, and visualize the training and validation datasets using different colors.

In [ ]:
```
# The point at which the stress-strain curve stops being linear
epsilon_l = 0.04
# Relevant data (this is nice way to get the linear part of the stresses and strain
x_rel = x[x < 0.04]
y_rel = y[x < 0.04]

# Visualize to make sure you have the right data
plt.figure()
plt.plot(
    x_rel,
    y_rel,
    'ro',
    markersize=2,
    label='Stress-strain data'
)
plt.xlabel('$\epsilon$ (strain in %)')
plt.ylabel('$\sigma$ (stress in MPa)')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



Split your data into training and validation.

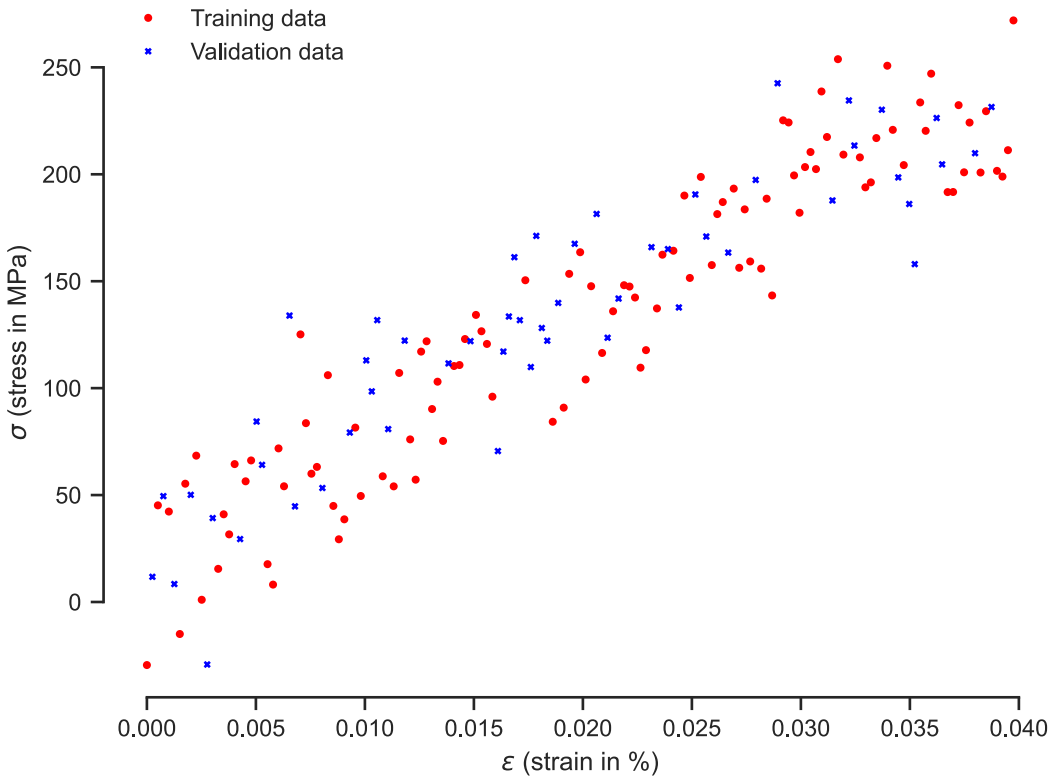
Hint: You may use sklearn.model_selection.train_test_split if you wish.

In [ ]:
```
# Split the data into training and validation datasets
# Hint: Consult the lecture notes
from sklearn.model_selection import train_test_split

x_train, x_valid, y_train, y_valid = train_test_split(x_rel, y_rel, test_size=0.33)
```

Use the following to visualize your split:

In [ ]:
```
plt.figure()
plt.plot(
    x_train,
    y_train,
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid,
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.xlabel('$\epsilon$ (strain in %)')
plt.ylabel('$\sigma$ (stress in MPa)')
```

```
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



Subpart A.II

Perform Bayesian linear regression with the evidence approximation to estimate the noise variance and the hyperparameters of the prior.

In [ ]:
```
def get_polynomial_design_matrix(x, degree):
    """Return the polynomial design matrix of ``degree`` evaluated at ``x``.

    Arguments:
    x      -- A 2D array with only one column.
    degree -- An integer greater than zero.
    """
    assert isinstance(x, np.ndarray), 'x is not a numpy array.'
    assert x.ndim == 2, 'You must make x a 2D array.'
    assert x.shape[1] == 1, 'x must be a column.'
    cols = []
    for i in range(degree+1):
        cols.append(x ** i)
    return np.hstack(cols)

def plot_posterior_predictive(
    x_values,
    y_values,
    model,
    xx,
    phi_func,
```

```
    phi_func_args=(),
    y_true=None
):
    """Plot the posterior predictive separating
    aleatory and espitemic uncertainty.

    Arguments:
    model    -- A trained model.
    xx       -- The points on which to evaluate
                the posterior predictive.
    phi_func -- The function to use to compute
                the design matrix.

    Keyword Arguments:
    phi_func_args -- Any arguments passed to the
                     function that calculates the
                     design matrix.
    y_true        -- The true response for plotting.
    """
    Phi_xx = phi_func(
        xx[:, None],
        *phi_func_args
    )
    yy_mean, yy_measured_std = model.predict(
        Phi_xx,
        return_std=True
    )
    sigma = np.sqrt(1.0 / model.alpha_)
    yy_std = np.sqrt(yy_measured_std ** 2 - sigma**2)
    yy_le = yy_mean - 2.0 * yy_std
    yy_ue = yy_mean + 2.0 * yy_std
    yy_lae = yy_mean - 2.0 * yy_measured_std
    yy_uae = yy_mean + 2.0 * yy_measured_std

    fig, ax = plt.subplots()
    ax.plot(xx, yy_mean, 'r', label="Posterior mean")
    ax.fill_between(
        xx,
        yy_le,
        yy_ue,
        color='red',
        alpha=0.25,
        label="95% epistemic credible interval"
    )
    ax.fill_between(
        xx,
        yy_lae,
        yy_le,
        color='green',
        alpha=0.25
    )
    ax.fill_between(
        xx,
        yy_ue,
        yy_uae,
        color='green',
```

```
        alpha=0.25,
        label="95% epistemic + aleatory credible interval"
    )
    ax.plot(x_values, y_values, 'kx', label='Observed data')
    if y_true is not None:
        ax.plot(xx, y_true, "--", label="True response")
    plt.xlabel('$\epsilon$ (strain in %)')
    plt.ylabel('$\sigma$ (stress in MPa)')
    plt.legend(loc="best", frameon=False)
    sns.despine(trim=True);
```

In [ ]:
```
from sklearn.linear_model import BayesianRidge

# Parameters
degree = 1

# Design matrix
Phi = get_polynomial_design_matrix(x_train[:, None], degree)

# Fit
model = BayesianRidge(
    fit_intercept=False
).fit(Phi, y_train)

sigma = np.sqrt(1.0 / model.alpha_)
print(f'sigma = {sigma:1.2f}')

# It calls it lambda...
alpha = np.sqrt(1.0 / model.lambda_)
print(f'alpha = {alpha:1.2f}')
```

```
sigma = 25.77
alpha = 3968.93
```

Subpart A.III

Calculate the mean square error of the validation data.

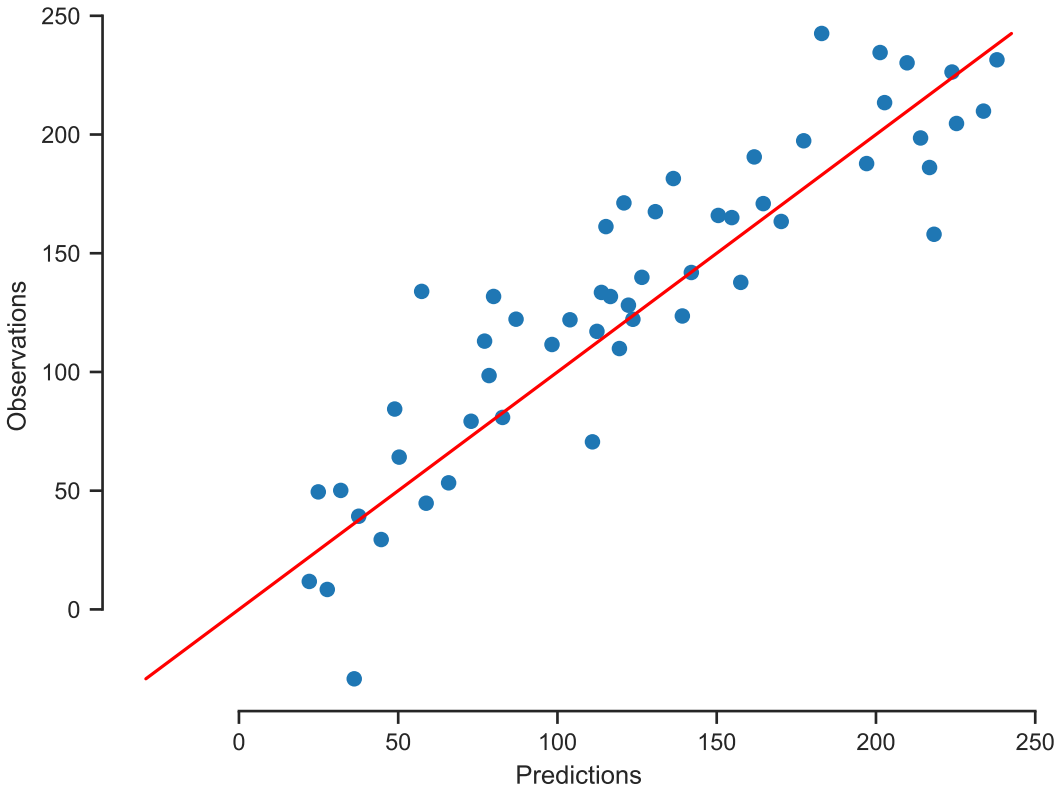
In [ ]:
```
Phi_valid = get_polynomial_design_matrix(x_valid[:, None], degree)

y_predict, y_std = model.predict(
    Phi_valid,
    return_std=True
)

MSE = np.mean((y_predict- y_valid) ** 2)
print(f'MSE = {MSE:1.2f}')
```

```
MSE = 826.43
```

Subpart A.IV

Make the observations vs predictions plot for the validation data.

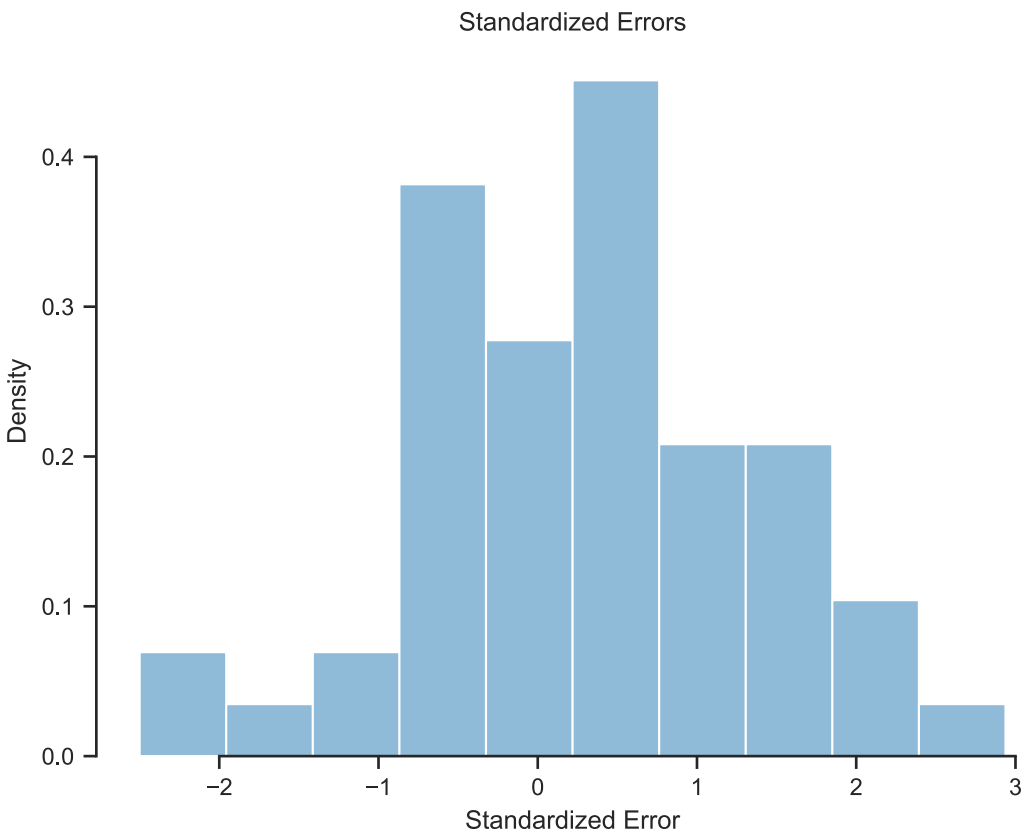
In [ ]:
```
fig, ax = plt.subplots()
ax.plot(y_predict, y_valid, 'o')
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
ax.plot(yys, yys, 'r-')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations')
sns.despine(trim=True);
```



Subpart A.V

Compute and plot the standardized errors for the validation data.

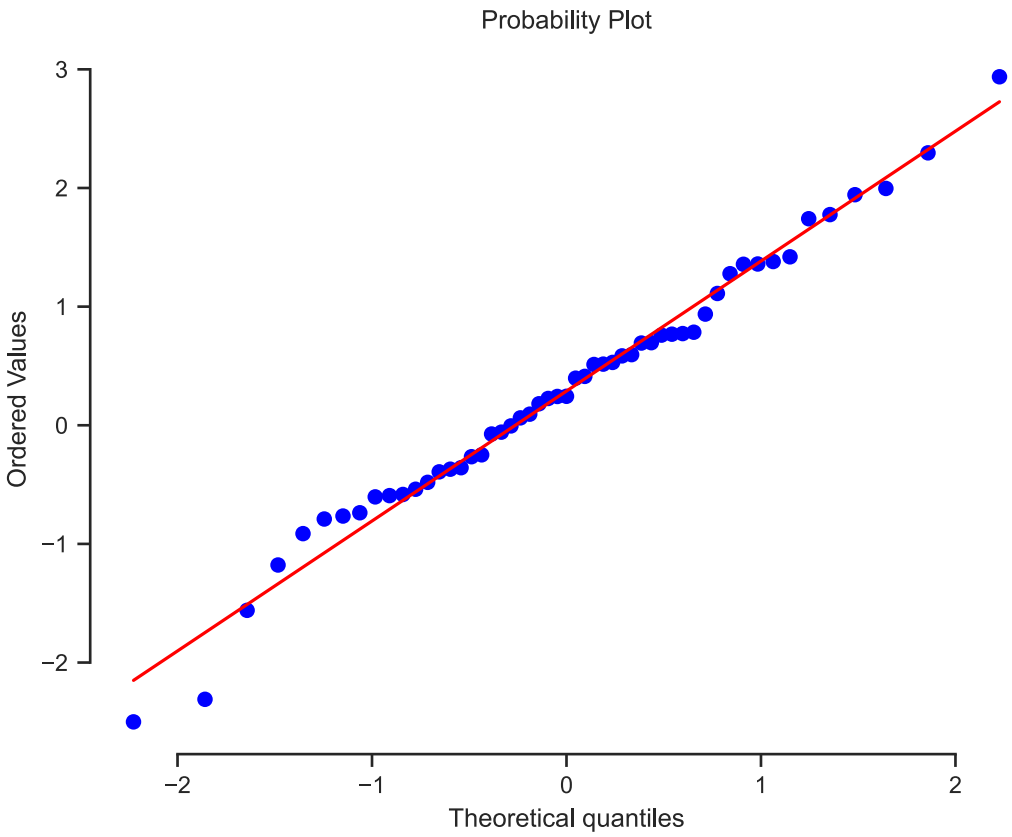
In [ ]:
```
eps = (y_valid - y_predict) / y_std

fig, ax = plt.subplots()
ax.hist(eps, alpha=0.5, density=True)
ax.set_xlabel('Standardized Error')
ax.set_ylabel('Density')
ax.set_title('Standardized Errors')
sns.despine(trim=True);
```



Subpart A.VI

Make the quantile-quantile plot of the standardized errors.

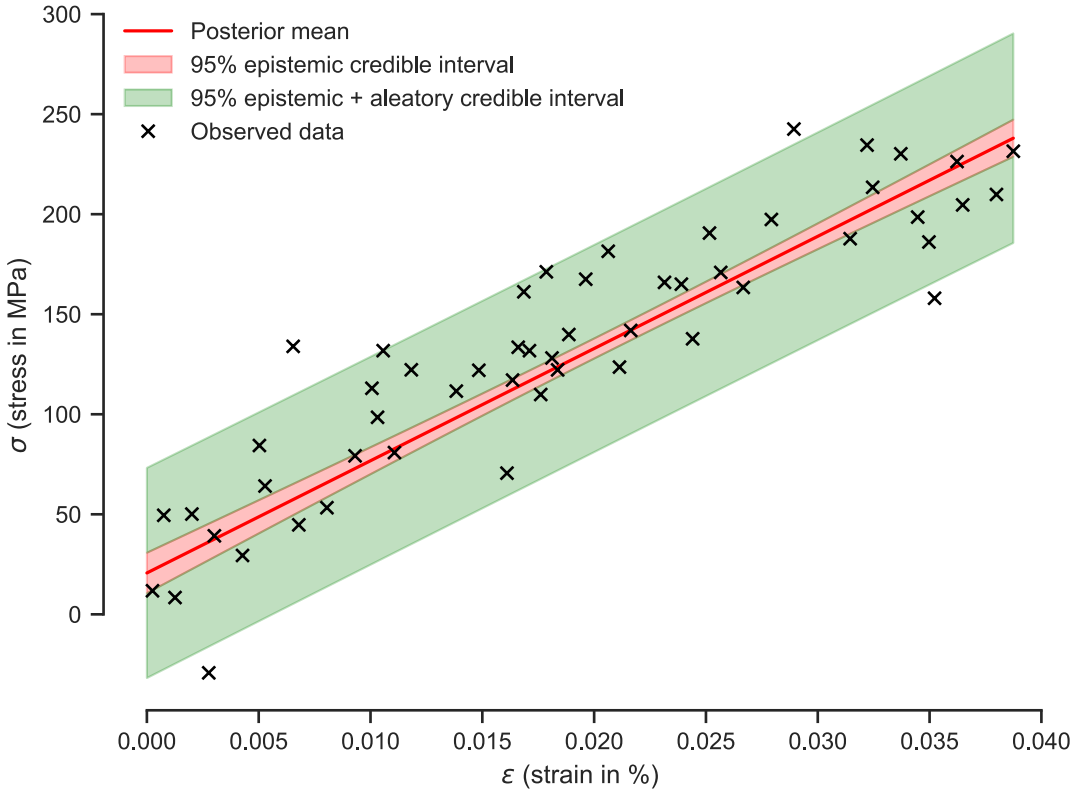
In [ ]:
```
fig, ax = plt.subplots()
st.probplot(eps, dist=st.norm, plot=ax)
sns.despine(trim=True);
```



Subpart A.VII

Visualize your epistemic and the aleatory uncertainty about the stress-strain curve in the elastic regime.

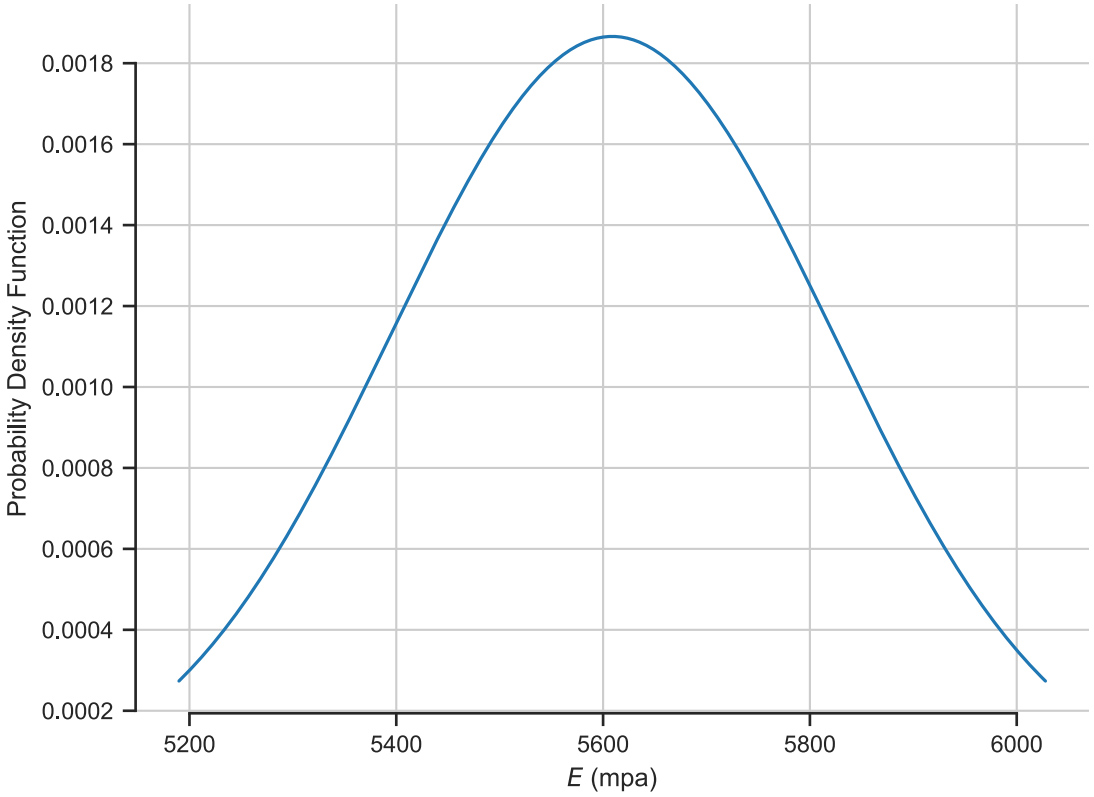
In [ ]:
```
xx = np.linspace(0,max(x_valid),len(y_predict))

plot_posterior_predictive(
    x_valid,
    y_valid,
    model,
    xx,
    get_polynomial_design_matrix,
    phi_func_args=(degree,),
    y_true=None
)
```



Subpart A. VIII

Visualize the posterior of the Young modulus E conditioned on the data.

In [ ]:
```
E_mean = model.coef_[1]
E_std = np.sqrt(model.sigma_[1][1])

E_dist = st.norm(loc=E_mean, scale=E_std)
xx_E = np.linspace(E_dist.ppf(0.025),E_dist.ppf(0.975),1000)
E_pdf = E_dist.pdf(xx_E)



fig, ax = plt.subplots()
ax.plot(xx_E,E_pdf)
ax.grid()
ax.set_xlabel('$E$ (mpa)')
ax.set_ylabel('Probability Density Function')
sns.despine(trim=True);
```



Subpart A.IX

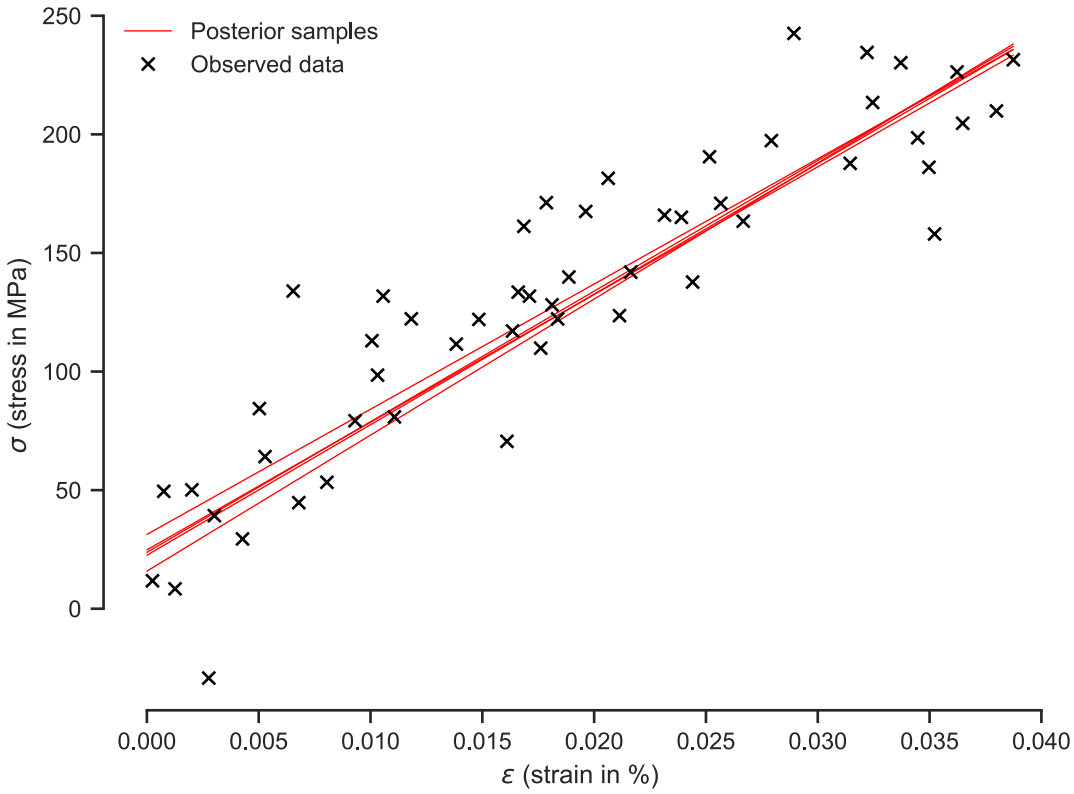
Take five samples of stress-strain curve in the elastic regime and visualize them.

In [ ]:
```
def plot_posterior_samples(
    x_values,
    y_values,
    model,
    xx,
    phi_func,
    phi_func_args=(),
    num_samples=10,
    y_true=None,
    nugget=1e-6
):
    """Plot posterior samples from the model.

    Arguments:
    model    -- A trained model.
    xx       -- The points on which to evaluate
                the posterior predictive.
    phi_func -- The function to use to compute
                the design matrix.

    Keyword Arguments:
    phi_func_args -- Any arguments passed to the
                     function that calculates the
                     design matrix.
    num_samples   -- The number of samples to take.
```

```
    y_true        -- The true response for plotting.
    nugget        -- A small number to add the covariance
                     if it is not positive definite
                     (numerically).
    """
    Phi_xx = phi_func(
        xx[:, None],
        *phi_func_args
    )
    m = model.coef_
    S = model.sigma_
    w_post = st.multivariate_normal(
        mean=m,
        cov=S + nugget * np.eye(S.shape[0])
    )
    fig, ax = plt.subplots()
    for _ in range(num_samples):
        w_sample = w_post.rvs()
        yy_sample = Phi_xx @ w_sample
        ax.plot(xx, yy_sample, 'r', lw=0.5)
    ax.plot([], [], "r", lw=0.5, label="Posterior samples")
    ax.plot(x_values, y_values, 'kx', label='Observed data')
    if y_true is not None:
        ax.plot(xx, y_true, "--", label="True response")
    plt.xlabel('$\epsilon$ (strain in %)')
    plt.ylabel('$\sigma$ (stress in MPa)')
    plt.legend(loc="best", frameon=False)
    sns.despine(trim=True);
```

In [ ]:

```
plot_posterior_samples(
    x_valid,
    y_valid,
    model,
    xx,
    get_polynomial_design_matrix,
    phi_func_args=(degree,),
    y_true=None,
    num_samples=5
)
```



Subpart A.X

Find the 95% centered credible interval for the Young modulus E.

In [ ]:
```
E_mean = model.coef_[1]
E_stdev = np.sqrt(model.sigma_[1][1])

E_dist = st.norm(loc=E_mean, scale=E_stdev)


print(f"95% CI = [{E_dist.ppf(0.025):2f},{E_dist.ppf(0.975):2f}]")
```

```
95% CI = [5189.817565,6027.787012]
```

Subpart A.XI

If you had to pick a single value for the Young modulus E, what would it be and why?

In [ ]:
```
print(f"Mean of E = [{E_mean:2f}]. I chose this value for it to be in the middle of
```

```
Mean of E = [5608.802288]. I chose this value for it to be in the middle of the 95%
interval
```

Your answer here:

I would pick the mean of E. This is in the middle of the Confidence Interval. That value is 5621.647748

Part B - Estimate the ultimate strength

The pick of the stress-strain curve is known as the ultimate strength. We want to estimate it.

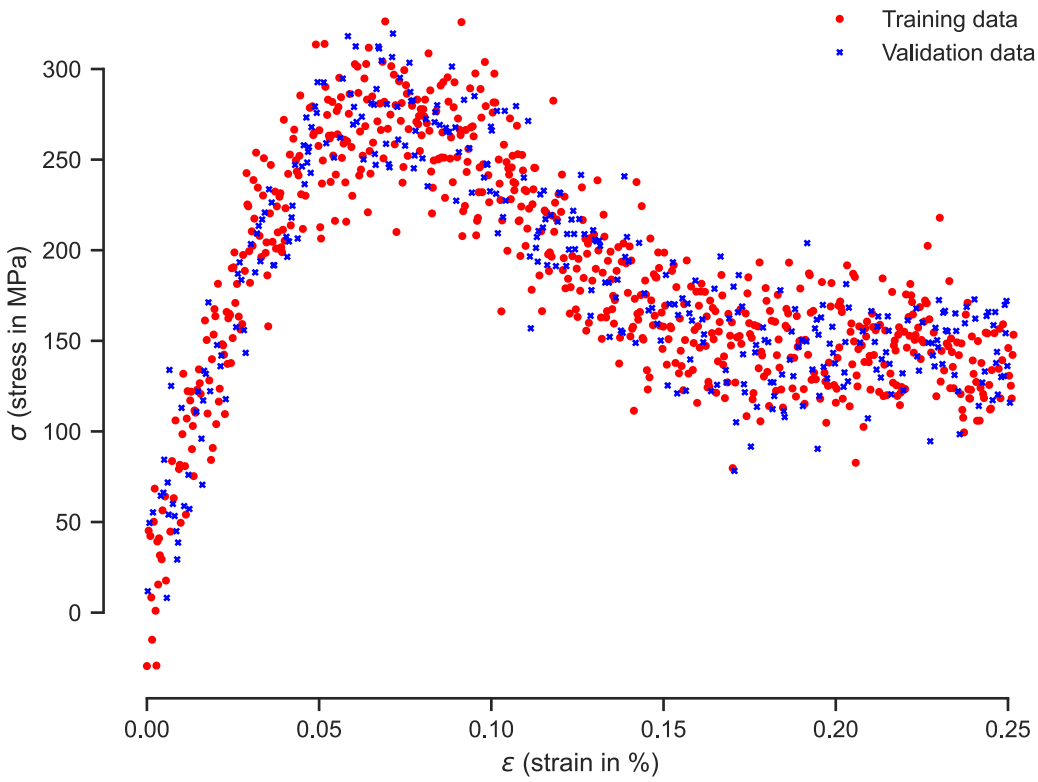
Subpart B.I - Extract training and validation data

Extract training and validation data from the entire dataset.

In [ ]:
```
# your code here - Repeat as many text and code blocks as you like

x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size=0.33)
```

Use the following to visualize your split:

In [ ]:
```
plt.figure()
plt.plot(
    x_train,
    y_train,
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid,
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.xlabel('$\epsilon$ (strain in %)')
plt.ylabel('$\sigma$ (stress in MPa)')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



Subpart B.II - Model the entire stress-strain relationship.

To do this, we will set up a generalized linear model to capture the entire stress-strain relationship. Remember, you can use any model you want as soon as:

- It is linear in the parameters to be estimated,
- It has a well-defined elastic regime (see Part A).

I am going to help you set up the right model. We will use the Heavide step function to turn on or off models for various ranges of ϵ. The idea is quite simple: We will use a linear model for the elastic regime, and we are going to turn to a non-linear model for the non-linear regime. Here is a model that has the right form in the elastic regime and an arbitrary form in the non-linear regime:

$$f(\epsilon; E, \mathbf{w}_g) = E\epsilon\left[(1 - H(\epsilon - \epsilon_l)\right] + g(\epsilon; \mathbf{w}_g)H(\epsilon - \epsilon_l),$$

where

$$H(x) = \begin{cases} 0, \text{ if } x < 0 \\ 1, \text{ otherwise,} \end{cases}$$

and g is any function linear in the parameters $\mathbf{w}_g$.

You can use any model you like for the non-linear regime, but let's use a polynomial of degree d:

$$g(\epsilon) = \sum_{i=0}^{d} w_i \epsilon^i.$$

The full model can be expressed as:

$$\begin{aligned} f(\epsilon) &= \begin{cases} h(\epsilon) = E\epsilon, \ \epsilon < \epsilon_l, \\ g(\epsilon) = \sum_{i=0}^{d} w_i \epsilon^i, \epsilon \geq \epsilon_l \end{cases} \\ &= E\epsilon \left(1 - H(\epsilon - \epsilon_l)\right) + \sum_{i=0}^{d} w_i \epsilon^i H(\epsilon - \epsilon_l). \end{aligned}$$

We could proceed with this model, but there is a small problem: It is discontinuous at $\epsilon = \epsilon_l$. This is unphysical. We can do better than that!

To make the model nice, we force the h and g to match up to the first derivative, i.e., we demand that:

$$\begin{aligned} h(\epsilon_l) &= g(\epsilon_l) \\ h'(\epsilon_l) &= g'(\epsilon_l). \end{aligned}$$

We include the first derivative because we don't have a kink in the stress-strain. That would also be unphysical. The two equations above become:

$$\begin{aligned} E\epsilon_l &= \sum_{i=0}^{d} w_i \epsilon_l^i \\ E &= \sum_{i=1}^{d} i w_i \epsilon_l^{i-1}. \end{aligned}$$

We can use these two equations to eliminate two weights. Let's eliminate w_0 and w_1. All you have to do is express them in terms of E and $w_2, \ldots, w_d$. So, there remain d parameters to estimate. Let's get back to the stress-strain model.

Our stress-strain model was:

$$f(\epsilon) = E\epsilon \left(1 - H(\epsilon - \epsilon_l)\right) + \sum_{i=0}^{d} w_i \epsilon^i H(\epsilon - \epsilon_l).$$

We can now use the expressions for w_0 and w_1 to rewrite this using only all the other parameters. I am going to spare you the details. The result is:

$$f(\epsilon) = E\epsilon + \sum_{i=2}^{d} w_i \left[(i-1)\epsilon_l^i - i\epsilon\epsilon_l^{i-1} + \epsilon^i\right] H(\epsilon - \epsilon_l).$$

Okay. This is still a generalized linear model. This is nice. Write code for the design matrix:

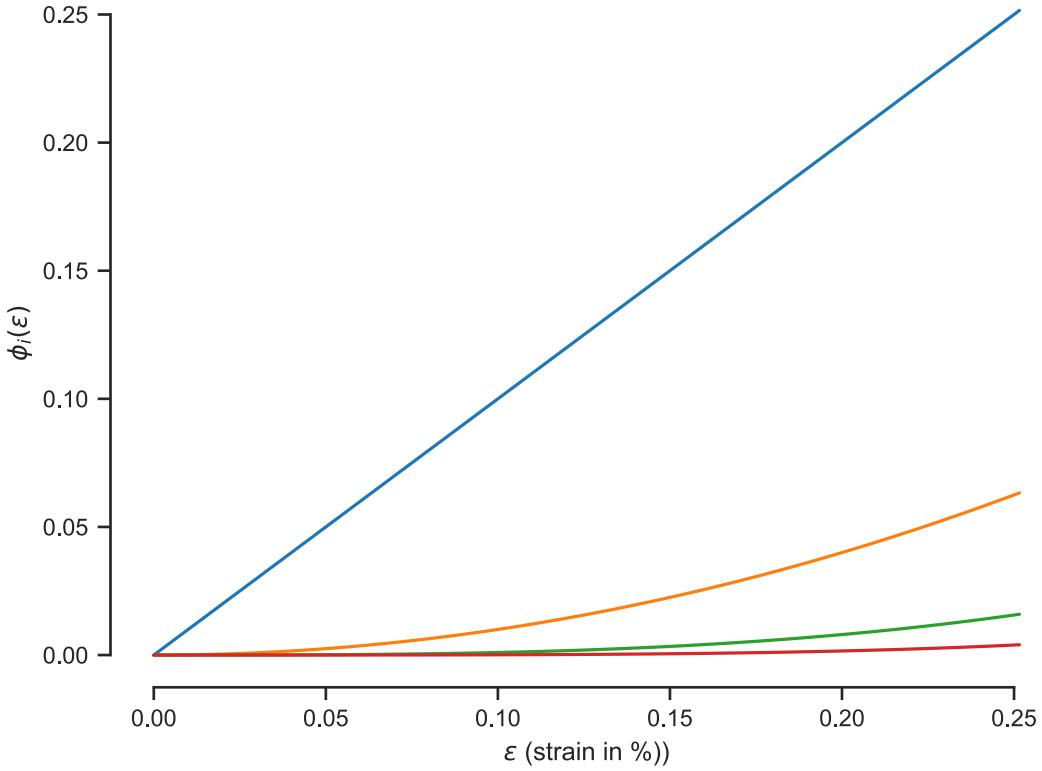
In [ ]:
```
# Complete this code to make your model:
def compute_design_matrix(Epsilon, epsilon_l, d):
```

```
    """Compute the design matrix for the stress-strain curve problem.

    Arguments:
        Epsilon     -     A 1D array of dimension N.
        epsilon_l   -     The strain signifying the end of the elastic regime.
        d           -     The polynomial degree.

    Returns:
        A design matrix N x d
    """
    # Sanity check
    assert isinstance(Epsilon, np.ndarray)
    assert Epsilon.ndim == 1, 'Pass the array as epsilon.flatten(), if it is two di
    n = Epsilon.shape[0]
    # The design matrix:
    Phi = np.ndarray((n, d))
    # The step function evaluated at all the elements of Epsilon.
    # You can use it if you want.
    Step = np.ones(n)
    Step[Epsilon < epsilon_l] = 0
    # Build the design matrix
    Phi[:, 0] = Epsilon
    for i in range(2, d+1):
        Phi[:, i-1] = Epsilon**i
    return Phi
```

Visualize the basis functions here:

In [ ]:

```
d = 4
eps = np.linspace(0, x.max(), 100)
Phis = compute_design_matrix(eps, epsilon_l, d)
fig, ax = plt.subplots(dpi=100)
ax.plot(eps, Phis)
ax.set_xlabel('$\epsilon$ (strain in %))')
ax.set_ylabel('$\phi_i(\epsilon)$')
sns.despine(trim=True);
```



Subpart B.III

Fit the model using automatic relevance determination and demonstrate that it works well by doing everything we did above (MSE, observations vs. predictions plot, standardized errors, etc.).

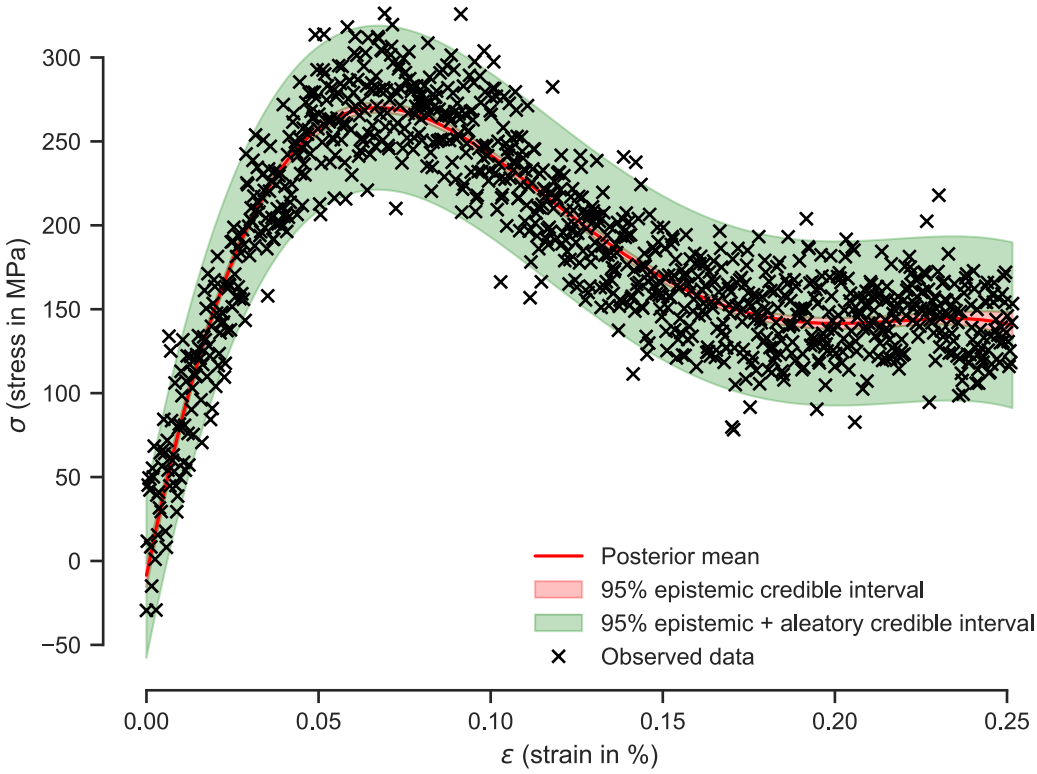
In [ ]:
```
# Your code here - Use as many blocks as you need!
from sklearn.linear_model import ARDRegression

# Parameters
degree = 4

# Design matrix
Phi = get_polynomial_design_matrix(x[:, None], degree)

# Fit
model = ARDRegression(
    fit_intercept=False
).fit(Phi, y)
```

Subpart B.IV

Visualize the epistemic and aleatory uncertainty in the stess-strain relation.

In [ ]:
```
xx = np.linspace(0,max(x),len(x))

plot_posterior_predictive(
```

```
    x,
    y,
    model,
    xx,
    get_polynomial_design_matrix,
    phi_func_args=(degree,),
    y_true=None
)
```



Subpart B.V - Extract the ultimate strength

Now, you will quantify your epistemic uncertainty about the ultimate strength. The ultimate strength is the maximum of the stress-strain relationship. Since you have epistemic uncertainty about the stress-strain relationship, you also have epistemic uncertainty about the ultimate strength.

Do the following:

- Visualize the posterior of the ultimate strength.
- Find a 95% credible interval for the ultimate strength.
- Pick a value for the ultimate strength.

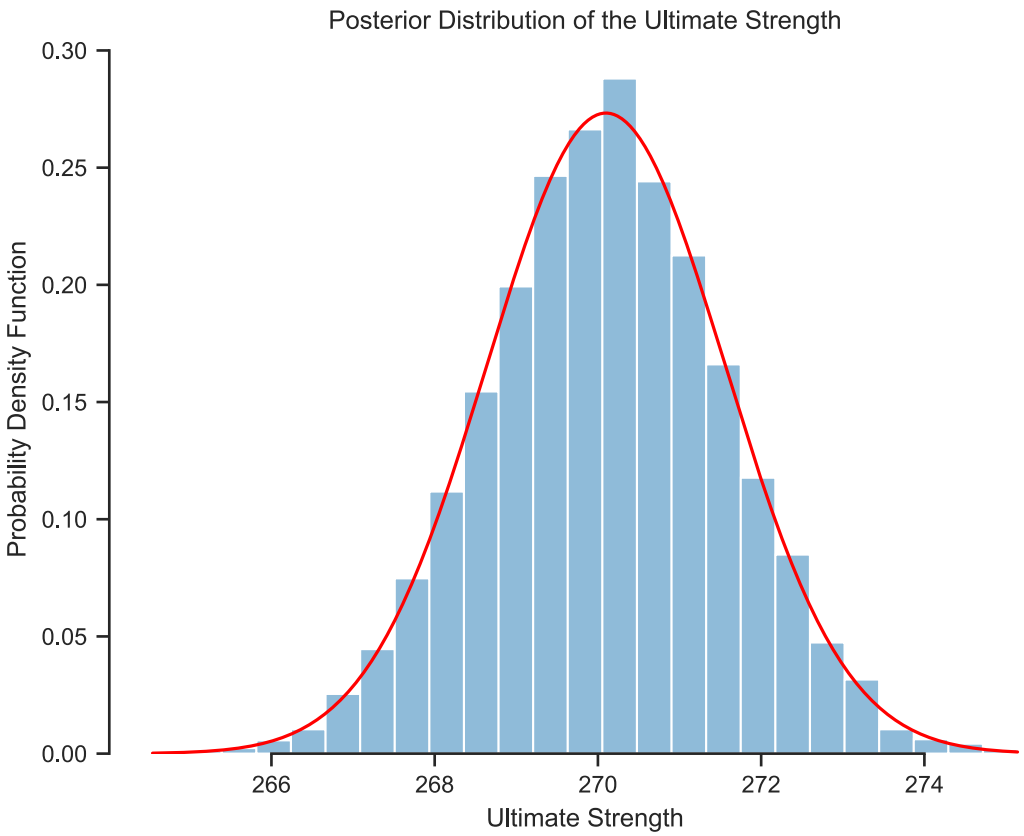
Hint: To characterize your epistemic uncertainty about the ultimate strength, you would have to do the following:

- Define a dense set of strain points between 0 and 0.25.
- Repeatedly:

- Sample from the posterior of the weights of your model
- For each sample, evaluate the stresses at the dense set of strain points defined earlier
- For each sampled stress vector, find the maximum. This is a sample of the ultimate strength.

In [ ]:
```
# default arguments from professor's functions
phi_func_args=()
num_samples=10000
y_true=None
nugget=1e-6

# used to find w_post
m = model.coef_
S = model.sigma_
w_post = st.multivariate_normal(
    mean=m,
    cov=S + nugget * np.eye(S.shape[0])
)


ult_strengths_list = []
for i in range(num_samples):
    w_sample = w_post.rvs() # sample
    stresses = Phi @ w_sample # execute phi
    ultimate_strength = np.max(stresses) # take the max
    ult_strengths_list.append(ultimate_strength) #continue the list


ult_strengths_list = np.array(ult_strengths_list)


mu, std = st.norm.fit(ult_strengths_list) # norm dist fit

fig, ax = plt.subplots()
ax.hist(ult_strengths_list, bins=25, alpha=0.5, density=True)
ee = np.linspace(ult_strengths_list.min(), ult_strengths_list.max(), 1000)
ax.plot(ee, st.norm.pdf(ee, mu, std), 'r')
ax.set_xlabel('Ultimate Strength')
ax.set_ylabel('Probability Density Function')
ax.set_title('Posterior Distribution of the Ultimate Strength')
sns.despine(trim=True);



mu = np.median(ult_strengths_list, axis=0)
mu_025 = np.percentile(ult_strengths_list, 2.5, axis=0)
mu_975 = np.percentile(ult_strengths_list, 97.5, axis=0)
credible_interval = np.percentile(ult_strengths_list, [2.5, 97.5])

print(f"95% Credible Interval = [{mu_025:2f},{mu_975:2f}]")
print(f"Median Credible Interval = {mu:2f}. I am picking this value for the ultimat
```

```
95% Credible Interval = [267.214831,272.972728]
Median Credible Interval = 270.104610. I am picking this value for the ultimate stre
ngth
```



Problem 2 - Optimizing the performance of a compressor

In this problem, we will need this dataset. The dataset was kindly provided to us by Professor Davide Ziviani. As before, you can either put it on your Google Drive or just download it with the code segment below:

In [ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecture
download(url)
```

Note that this is an Excel file, so we need pandas to read it. Here is how:

In [ ]:
```
import pandas as pd
data = pd.read_excel('compressor_data.xlsx')
data
```

Out[ ]:

	T_e	DT_sh	T_c	DT_sc	T_amb	f	m_dot	m_dot.1	Capacity	Power	Current	COP
0	-30	11	25	8	35	60	28.8	8.000000	1557	901	4.4	1.73
1	-30	11	30	8	35	60	23.0	6.388889	1201	881	4.0	1.36
2	-30	11	35	8	35	60	17.9	4.972222	892	858	3.7	1.04
3	-25	11	25	8	35	60	46.4	12.888889	2509	1125	5.3	2.23
4	-25	11	30	8	35	60	40.2	11.166667	2098	1122	5.1	1.87
...	...	...	...	...	...	...	...	...	...	...	...	...
60	10	11	45	8	35	60	245.2	68.111111	12057	2525	11.3	4.78
61	10	11	50	8	35	60	234.1	65.027778	10939	2740	12.3	3.99
62	10	11	55	8	35	60	222.2	61.722222	9819	2929	13.1	3.35
63	10	11	60	8	35	60	209.3	58.138889	8697	3091	13.7	2.81
64	10	11	65	8	35	60	195.4	54.277778	7575	3223	14.2	2.35

65 rows × 13 columns

The data are part of an experimental study of a variable-speed reciprocating compressor. The experimentalists varied two temperatures, T_e and T_c (both in C), and they measured various other quantities. We aim to learn the map between T_e and T_c and measure Capacity and Power (both in W). First, let's see how you can extract only the relevant data.

In [ ]:
```
# Here is how to extract the T_e and T_c columns and put them in a single numpy arr
x = data[['T_e','T_c']].values
x
```

Out[ ]:
```
array([[-30,  25],
       [-30,  30],
       [-30,  35],
       [-25,  25],
       [-25,  30],
       [-25,  35],
       [-25,  40],
       [-25,  45],
       [-20,  25],
       [-20,  30],
       [-20,  35],
       [-20,  40],
       [-20,  45],
       [-20,  50],
       [-15,  25],
       [-15,  30],
       [-15,  35],
       [-15,  40],
       [-15,  45],
       [-15,  50],
       [-15,  55],
       [-10,  25],
       [-10,  30],
       [-10,  35],
       [-10,  40],
       [-10,  45],
       [-10,  50],
       [-10,  55],
       [-10,  60],
       [ -5,  25],
       [ -5,  30],
       [ -5,  35],
       [ -5,  40],
       [ -5,  45],
       [ -5,  50],
       [ -5,  55],
       [ -5,  60],
       [ -5,  65],
       [  0,  25],
       [  0,  30],
       [  0,  35],
       [  0,  40],
       [  0,  45],
       [  0,  50],
       [  0,  55],
       [  0,  60],
       [  0,  65],
       [  5,  25],
       [  5,  30],
       [  5,  35],
       [  5,  40],
       [  5,  45],
       [  5,  50],
       [  5,  55],
       [  5,  60],
       [  5,  65],
```

```
       [ 10,  25],
       [ 10,  30],
       [ 10,  35],
       [ 10,  40],
       [ 10,  45],
       [ 10,  50],
       [ 10,  55],
       [ 10,  60],
       [ 10,  65]], dtype=int64)
```

In [ ]:
```
# Here is how to extract the Capacity
y = data['Capacity'].values
y
```

Out[ ]:
```
array([ 1557,  1201,   892,  2509,  2098,  1726,  1398,  1112,  3684,
        3206,  2762,  2354,  1981,  1647,  5100,  4547,  4019,  3520,
        3050,  2612,  2206,  6777,  6137,  5516,  4915,  4338,  3784,
        3256,  2755,  8734,  7996,  7271,  6559,  5863,  5184,  4524,
        3883,  3264, 10989, 10144,  9304,  8471,  7646,  6831,  6027,
        5237,  4461, 13562, 12599, 11633, 10668,  9704,  8743,  7786,
        6835,  5891, 16472, 15380, 14279, 13171, 12057, 10939,  9819,
        8697,  7575], dtype=int64)
```

Fit the following multivariate polynomial model to **both the Capacity and the Power**:

$$y = w_1 + w_2 T_e + w_3 T_c + w_4 T_e T_c + w_5 T_e^2 + w_6 T_c^2 + w_7 T_e^2 T_c + w_8 T_e T_c^2 + w_9 T_e^3 + w_{10} T_c^3$$

where ϵ is a Gaussian noise term with unknown variance.

Hints:

- You may use sklearn.preprocessing.PolynomialFeatures to construct the design matrix of your polynomial features. Do not program the design matrix by hand.
- You should split your data into training and validation and use various validation metrics to ensure your models make sense.
- Use ARD Regression to fit any hyperparameters and the noise.

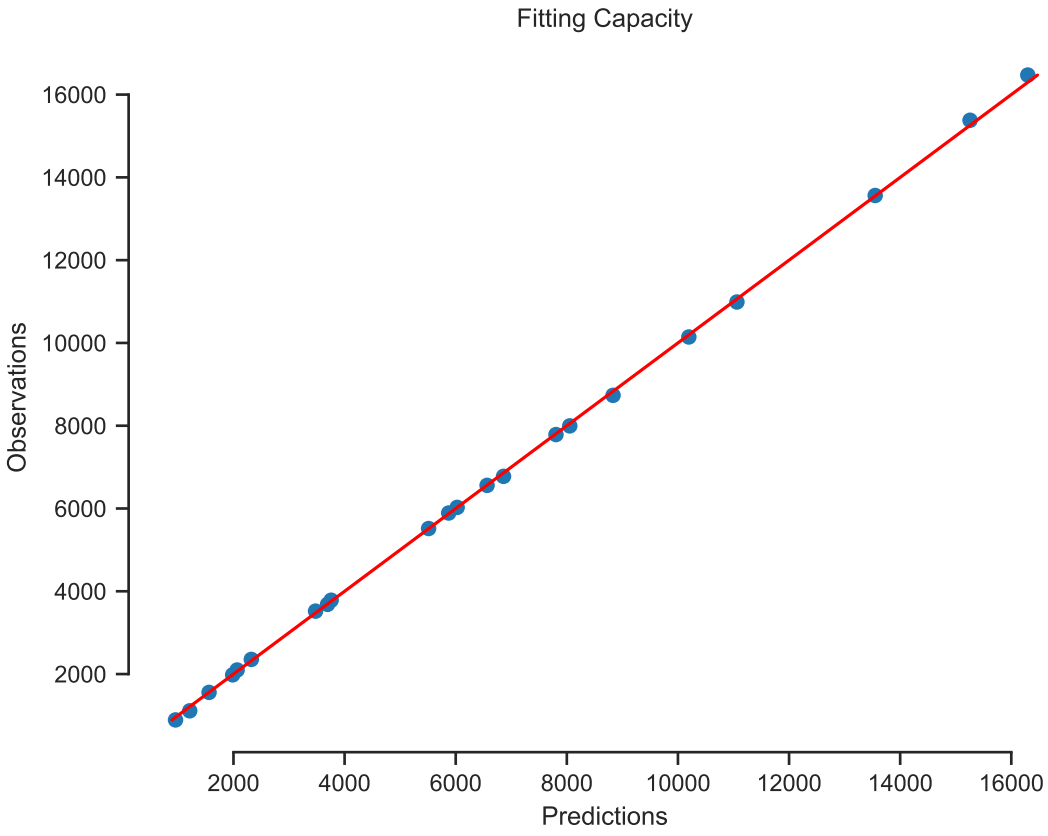
Part A - Fit the capacity

Subpart A.I

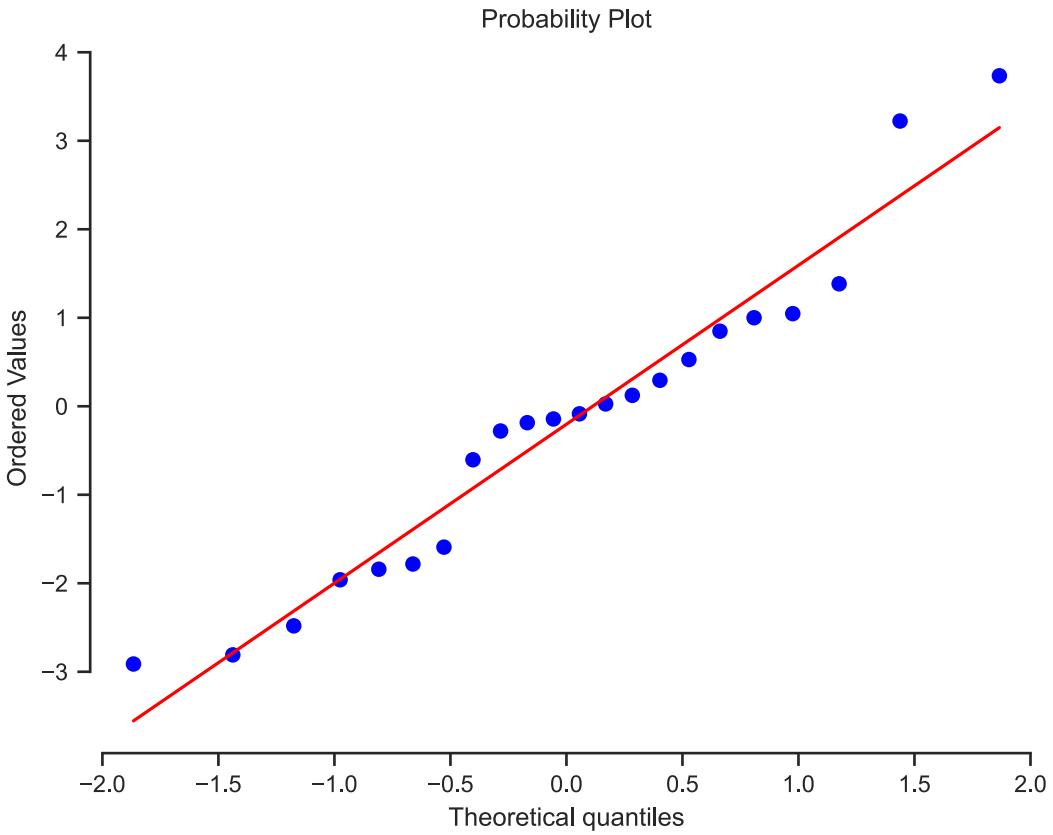
Please don't just fit. Split in training and test and use all the usual diagnostics.

In [ ]:
```
from sklearn.preprocessing import PolynomialFeatures
y = data['Capacity'].values

x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size=0.33)


# Parameters
degrees = 3
```

```
# Design matrix
poly = PolynomialFeatures(degree=degrees)
Phi = poly.fit_transform(x_train)
Phi_val = poly.transform(x_valid)

# Fit
model = ARDRegression(
    fit_intercept=False
).fit(Phi, y_train)


y_predict, y_std = model.predict(
    Phi_val,
    return_std=True
)

fig, ax = plt.subplots()
ax.plot(y_predict, y_valid, 'o')
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
plt.title('Fitting Capacity')
ax.plot(yys, yys, 'r-')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations')
sns.despine(trim=True);
```



In [ ]:
```
eps = (y_valid - y_predict) / y_std


fig, ax = plt.subplots()
st.probplot(eps, dist=st.norm, plot=ax)
sns.despine(trim=True);
```



Subpart A.II

What is the noise variance you estimated for the Capacity?

In [ ]:
```
# your code here
sigma = np.sqrt(1.0 / model.alpha_)
print(f'Noise variance -- sigma = {sigma:1.2f}')
```

```
Noise variance -- sigma = 29.60
```

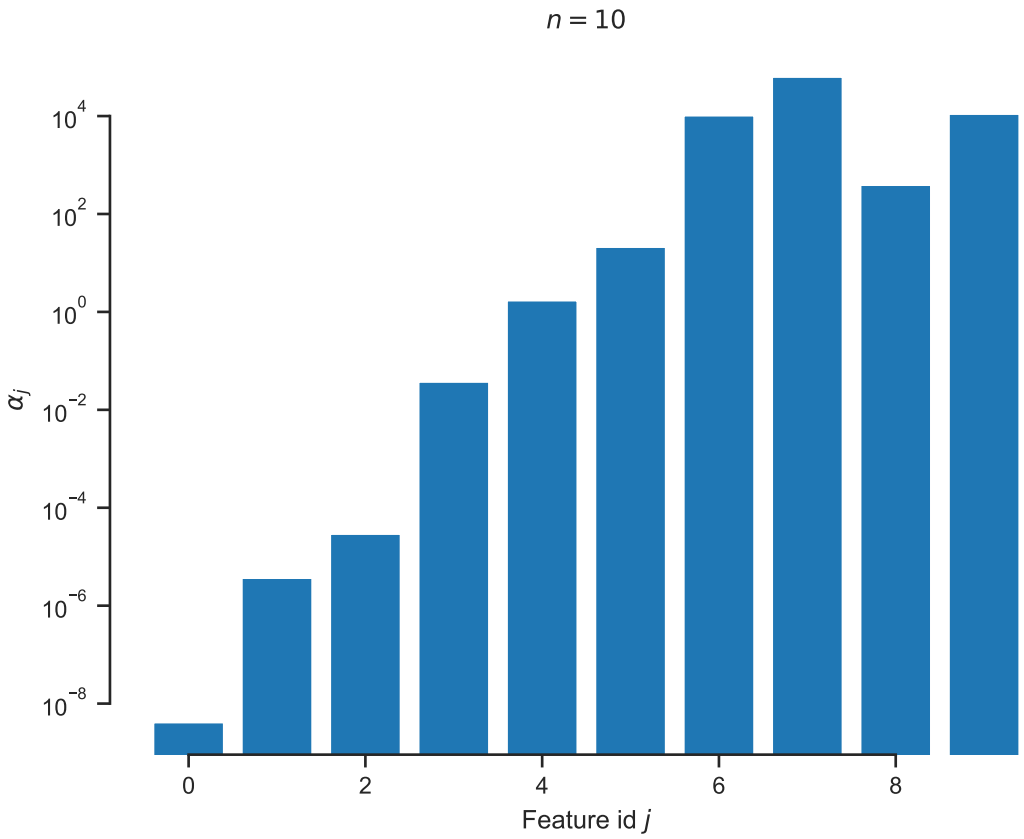
Subpart A.III

Which features of the temperatures (basis functions of your model) are the most important for predicting the Capacity?

In [ ]:
```
alpha = model.lambda_
print(f"Prior w precision: {alpha}")

fig, ax = plt.subplots()
ax.bar(np.arange(alpha.shape[0]), alpha)
```

```
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$')
ax.set_yscale("log")
ax.set_title(f'$n={alpha.shape[0]}$')
sns.despine(trim=True);


m = model.coef_
print(f"Posterior mean w: {m}")
```

```
Prior w precision: [4.13592065e-09 3.68554765e-06 2.92673240e-05 3.74080953e-02
 1.71577342e+00 2.12588098e+01 1.02832928e+04 6.32779759e+04
 3.91852190e+02 1.10793574e+04]
Posterior mean w: [ 1.55490781e+04  5.20851691e+02 -1.84790696e+02  5.17012459e+00
 -7.01861091e-01  2.11769240e-01  0.00000000e+00  0.00000000e+00
 -5.03841302e-02  0.00000000e+00]
```



From hands-on-15.2: Remember that the higher the prior precision α_j of a weight, the more its prior (and consequently the posterior) concentrates about zero. The corresponding weight is essentially zero for extremely high values of prior precision.

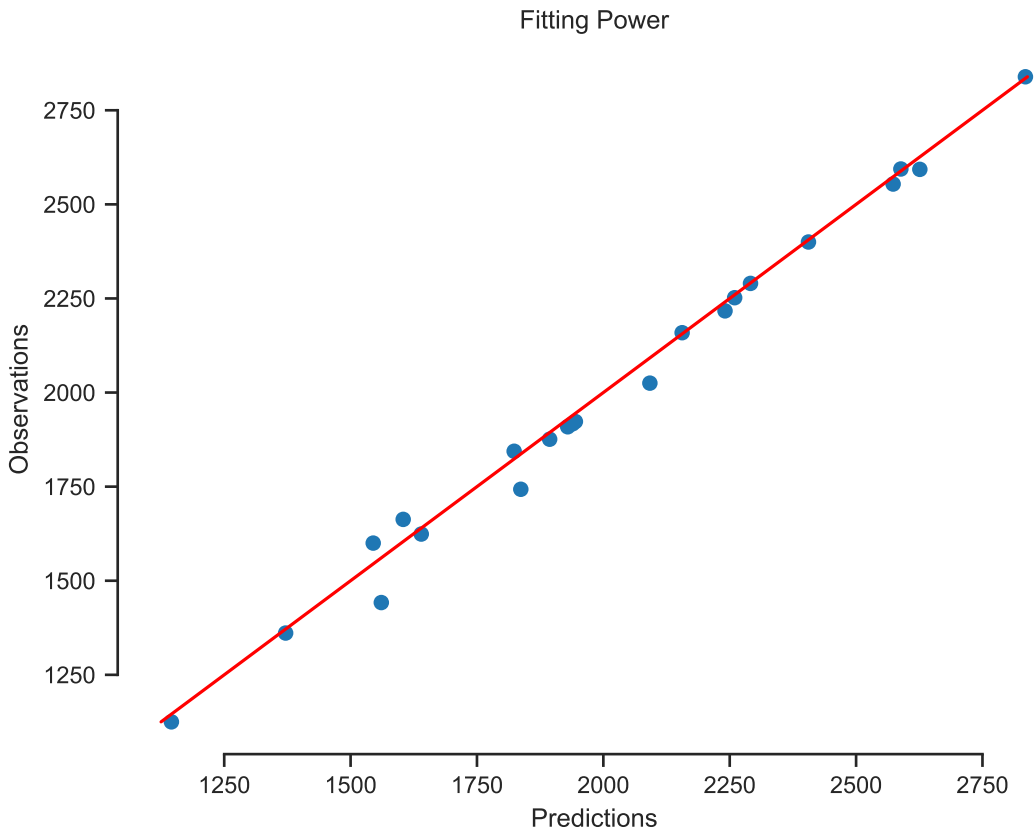
This means that the ARD is saying we do not have to include 4th, 5th, 6th, 7th, 8th, and 9th-degree monomials.

Part B - Fit the Power

Subpart B.I

Please don't just fit. Split in training and test and use all the usual diagnostics.

In [ ]:
```
y = data['Power'].values

x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size=0.33)


# Parameters
degrees = 3


# Design matrix
poly = PolynomialFeatures(degree=degrees)
Phi = poly.fit_transform(x_train)
Phi_val = poly.transform(x_valid)

# Fit
model = ARDRegression(
    fit_intercept=False
).fit(Phi, y_train)


y_predict, y_std = model.predict(
    Phi_val,
    return_std=True
)

fig, ax = plt.subplots()
ax.plot(y_predict, y_valid, 'o')
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
ax.plot(yys, yys, 'r-')
plt.title('Fitting Power')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations')
sns.despine(trim=True);
```



Subpart B.II

What is the noise variance you estimated for the Power?

In [ ]:
```
# your code here
sigma = np.sqrt(1.0 / model.alpha_)
print(f'Noise variance -- sigma = {sigma:1.2f}')
```

```
Noise variance -- sigma = 23.81
```

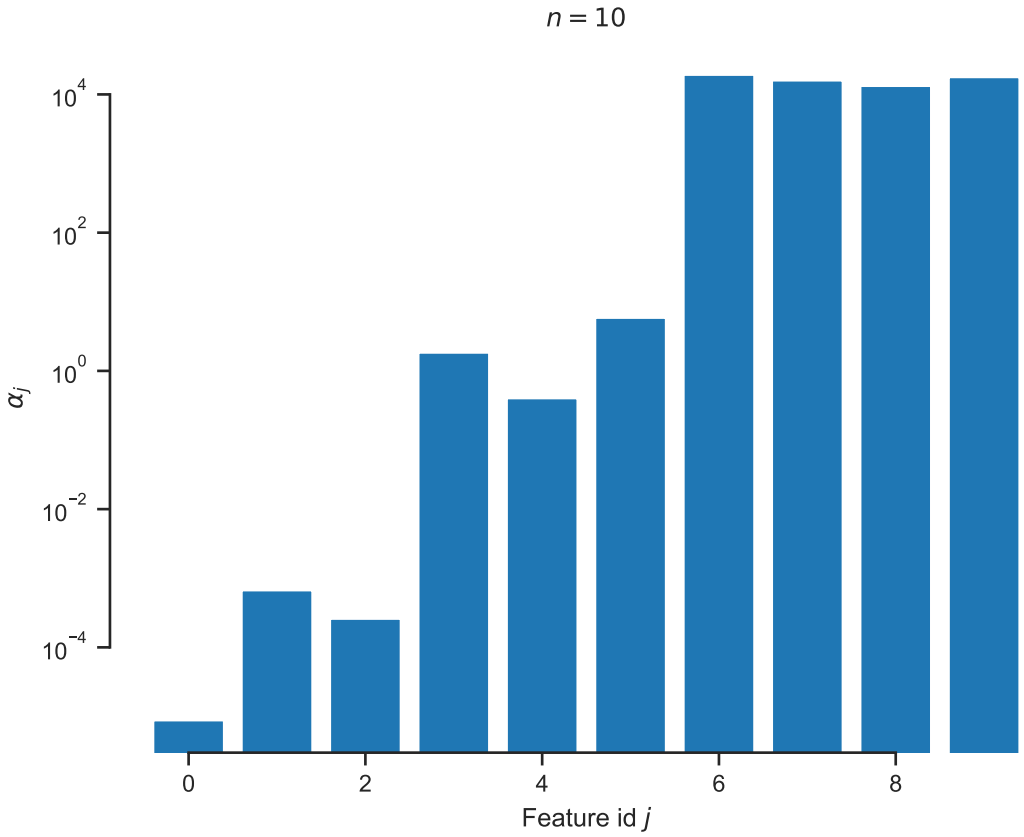
Subpart B.III

Which features of the temperatures (basis functions of your model) are the most important for predicting the Power?

In [ ]:
```
alpha = model.lambda_
print(f"Prior w precision: {alpha}")

fig, ax = plt.subplots()
ax.bar(np.arange(alpha.shape[0]), alpha)
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$')
ax.set_yscale("log")
ax.set_title(f'$n={alpha.shape[0]}$')
sns.despine(trim=True);
```

```
m = model.coef_
print(f"Posterior mean w: {m}")
```

```
Prior w precision: [8.78730916e-06 6.68035576e-04 2.59020254e-04 1.83501075e+00
 4.01450485e-01 5.85807966e+00 1.92034668e+04 1.59092710e+04
 1.32614161e+04 1.77643131e+04]
Posterior mean w: [332.18676453 -38.64472282  62.07405979  -0.73754343   1.5778292
  -0.41205066   0.           0.           0.           0.        ]
```



From hands-on-15.2: Remember that the higher the prior precision α_j of a weight, the more its prior (and consequently the posterior) concentrates about zero. The corresponding weight is essentially zero for extremely high values of prior precision.

This means that the ARD is saying we do not have to include 4th, 5th, 6th, 7th, 8th, and 9th-degree monomials.

Problem 3 - Explaining the Challenger disaster

On January 28, 1986, the Space Shuttle Challenger disintegrated after 73 seconds from launch. The failure can be traced to the rubber O-rings, which were used to seal the joints of the solid rocket boosters (required to force the hot, high-pressure gases generated by the burning solid propellant through the nozzles, thus producing thrust).

The performance of the O-ring material was sensitive to the external temperature during launch. This dataset contains records of different experiments with O-rings recorded at various times between 1981 and 1986. Download the data the usual way (either put them on Google Drive or run the code cell below).

In [ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecture
download(url)
```

Even though this is a CSV file, you should load it with pandas because it contains some special characters.

In [ ]:
```
raw_data = pd.read_csv('challenger_data.csv')
raw_data
```

Out[ ]:

	Date	Temperature	Damage Incident
0	04/12/1981	66	0
1	11/12/1981	70	1
2	3/22/82	69	0
3	6/27/82	80	NaN
4	01/11/1982	68	0
5	04/04/1983	67	0
6	6/18/83	72	0
7	8/30/83	73	0
8	11/28/83	70	0
9	02/03/1984	57	1
10	04/06/1984	63	1
11	8/30/84	70	1
12	10/05/1984	78	0
13	11/08/1984	67	0
14	1/24/85	53	1
15	04/12/1985	67	0
16	4/29/85	75	0
17	6/17/85	70	0
18	7/29/85	81	0
19	8/27/85	76	0
20	10/03/1985	79	0
21	10/30/85	75	1
22	11/26/85	76	0
23	01/12/1986	58	1
24	1/28/86	31	Challenger Accident

The first column is the date of the record. The second column is the external temperature of that day in degrees F. The third column labeled `Damage Incident` has a binary coding (0=no damage, 1=damage). The very last row is the day of the Challenger accident.

We will use the first 23 rows to solve a binary classification problem that will give us the probability of an accident conditioned on the observed external temperature in degrees F. Before proceeding to the data analysis, let's clean the data up.

First, we drop all the bad records:

In [ ]:
```
clean_data_0 = raw_data.dropna()
clean_data_0
```

Out[ ]:

	Date	Temperature	Damage Incident
0	04/12/1981	66	0
1	11/12/1981	70	1
2	3/22/82	69	0
4	01/11/1982	68	0
5	04/04/1983	67	0
6	6/18/83	72	0
7	8/30/83	73	0
8	11/28/83	70	0
9	02/03/1984	57	1
10	04/06/1984	63	1
11	8/30/84	70	1
12	10/05/1984	78	0
13	11/08/1984	67	0
14	1/24/85	53	1
15	04/12/1985	67	0
16	4/29/85	75	0
17	6/17/85	70	0
18	7/29/85	81	0
19	8/27/85	76	0
20	10/03/1985	79	0
21	10/30/85	75	1
22	11/26/85	76	0
23	01/12/1986	58	1
24	1/28/86	31	Challenger Accident

We also don't need the last record. Remember that the temperature on the day of the Challenger accident was 31 degrees F.

In [ ]:
```
clean_data = clean_data_0[:-1]
clean_data
```

Out[ ]:

	Date	Temperature	Damage Incident
0	04/12/1981	66	0
1	11/12/1981	70	1
2	3/22/82	69	0
4	01/11/1982	68	0
5	04/04/1983	67	0
6	6/18/83	72	0
7	8/30/83	73	0
8	11/28/83	70	0
9	02/03/1984	57	1
10	04/06/1984	63	1
11	8/30/84	70	1
12	10/05/1984	78	0
13	11/08/1984	67	0
14	1/24/85	53	1
15	04/12/1985	67	0
16	4/29/85	75	0
17	6/17/85	70	0
18	7/29/85	81	0
19	8/27/85	76	0
20	10/03/1985	79	0
21	10/30/85	75	1
22	11/26/85	76	0
23	01/12/1986	58	1

Let's extract the features and the labels:

In [ ]:
```
x = clean_data['Temperature'].values
x
```

Out[ ]:
```
array([66, 70, 69, 68, 67, 72, 73, 70, 57, 63, 70, 78, 67, 53, 67, 75, 70,
       81, 76, 79, 75, 76, 58], dtype=int64)
```

In [ ]:
```
y = clean_data['Damage Incident'].values.astype(np.float64)
y
```

Out[ ]:
```
array([0., 1., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 0., 0., 0.,
       0., 0., 0., 1., 0., 1.])
```

Part A - Perform logistic regression

Perform logistic regression between the temperature (x) and the damage label (y). Refrain from validating because there is little data. Just use a simple model so that you don't overfit.

In [ ]:
```
from sklearn.linear_model import LogisticRegression

# Design matrix
poly = PolynomialFeatures(1) #Chose 1 to have only one crossing point in the model
Phi = poly.fit_transform(x[:, None])

# Train the model (penalty = 'none' means that we do not add a prior on the weights
# we are effectively just maximizing the likelihood of the data
model = LogisticRegression(
    penalty=None,
    fit_intercept=False
).fit(Phi, y);
```

In [ ]:
```
model.coef_
```

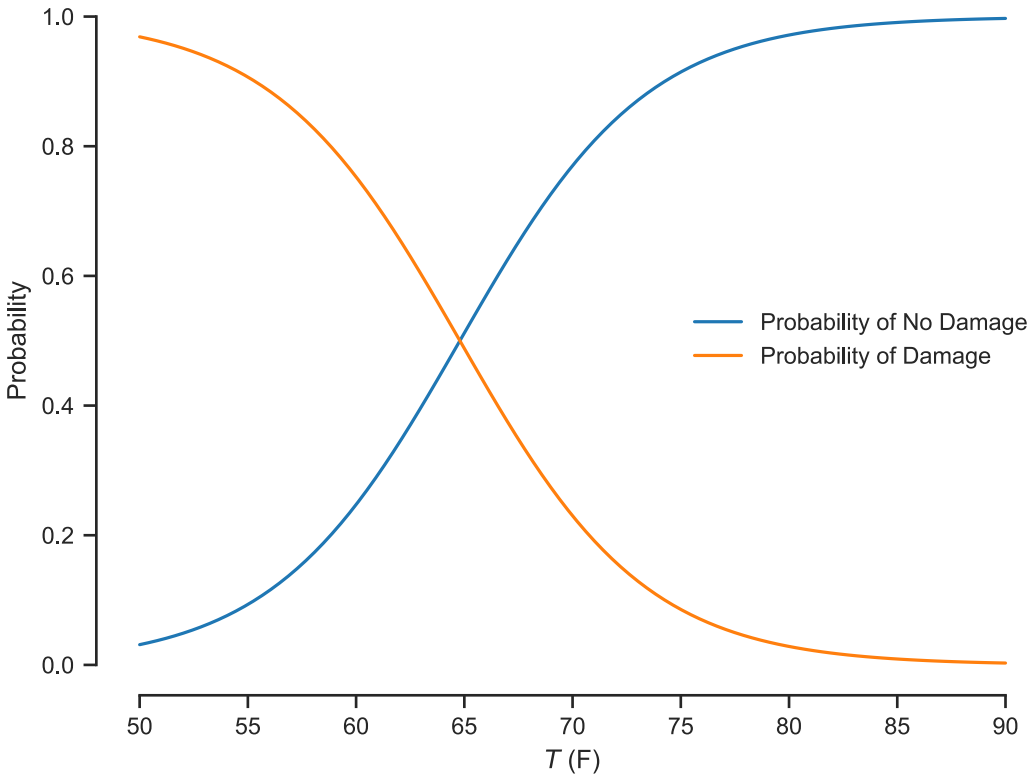
Out[ ]:
```
array([[15.04290143, -0.23216274]])
```

Part B - Plot the probability of damage as a function of temperature

Plot the probability of damage as a function of temperature.

In [ ]:
```
fig, ax = plt.subplots()
xx = np.linspace(50, 90, 100)
Phi_xx = poly.fit_transform(xx[:, None])
predictions_xx = model.predict_proba(Phi_xx)
ax.plot(
    xx,
    predictions_xx[:, 0],
    label='Probability of No Damage'
)
ax.plot(
    xx,
    predictions_xx[:, 1],
    label='Probability of Damage'
)
ax.set_xlabel('$T$ (F)')
ax.set_ylabel('Probability')
```

```
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



Part C - Decide whether or not to launch

The temperature on the day of the Challenger accident was 31 degrees F. Start by calculating the probability of damage at 31 degrees F. Then, use formal decision-making (i.e., define a cost matrix and make decisions by minimizing the expected loss) to decide whether or not to launch on that day. Also, plot your optimal decision as a function of the external temperature.

In [ ]:
```
# your code here - Repeat as many text and code blocks as you like
def expected_cost(cost_matrix, prediction_prob):
    """Calculate the expected cost of each decision.

    Arguments
    cost_matrix     --  A D x D matrix. `cost_matrix[i, j]`
                        is the cost of picking `i` and then
                        `j` happens.
    prediction_prob --  An array with D elements containing
                        the probability that each event
                        happens.
    """
    assert cost_matrix.ndim == 2
    D = cost_matrix.shape[0]
    assert cost_matrix.shape[1] == D
    assert prediction_prob.ndim == 1
    assert prediction_prob.shape[0] == D
```

```
    res = np.zeros((2,))
    for i in range(2):
        res[i] = (
            cost_matrix[i, 0] * prediction_prob[0]
            + cost_matrix[i, 1] * prediction_prob[1]
        )
    return res
```

In [ ]:
```
xx_31 = np.array([31])
Phi_31 = poly.fit_transform(xx_31[:,None])
predictions_31 = model.predict_proba(Phi_31)

print(f'Probability of damage at 31 degrees F = {predictions_31[0,1]:1.2f}')
```

```
Probability of damage at 31 degrees F = 1.00
```

In [ ]:
```
# c_00 = cost of lauching without damage when no damage is expected
# c_01 = cost of lauching without damage when damage is expected
# c_10 = cost of lauching with damage when damage is not expected
# c_11 = cost of lauching with damage when damage is expected

cost_matrix = np.array(
    [
        [0.0, 1.0],
        [1.0, 0.0]
    ]
)
```

In [ ]:
```
# Expected loss calculation
exp_c = expected_cost(cost_matrix, predictions_31[0])

print(f'Probability of damage at 31 degrees F = {exp_c[0]:1.2f}')
```

```
Probability of damage at 31 degrees F = 1.00
```

In [ ]:
```
xx = np.linspace(0, 100, 101)
Phi_xx = poly.fit_transform(xx[:, None])
predictions = model.predict_proba(Phi_xx)


print('Temp\tCost lauching with damage\tCost lauching without damage\tChoice')
print('-' * 80)
exp_c_list_0 = []
exp_c_list_1 = []
choice_31_degrees = None
choice_first_launch = None
for i in range(xx.shape[0]):
    exp_c = expected_cost(cost_matrix, predictions[i])

    line = f'{xx[i]:1.2f}\t{exp_c[0]:1.2f}'
    tmp = f'\t\t{exp_c[1]:1.2f}'
    if i == 31:
        if exp_c[0] > exp_c[1]:
            choice_31_degrees =  f'No Launch'
        else:
            choice_31_degrees =  f'Launch'
```

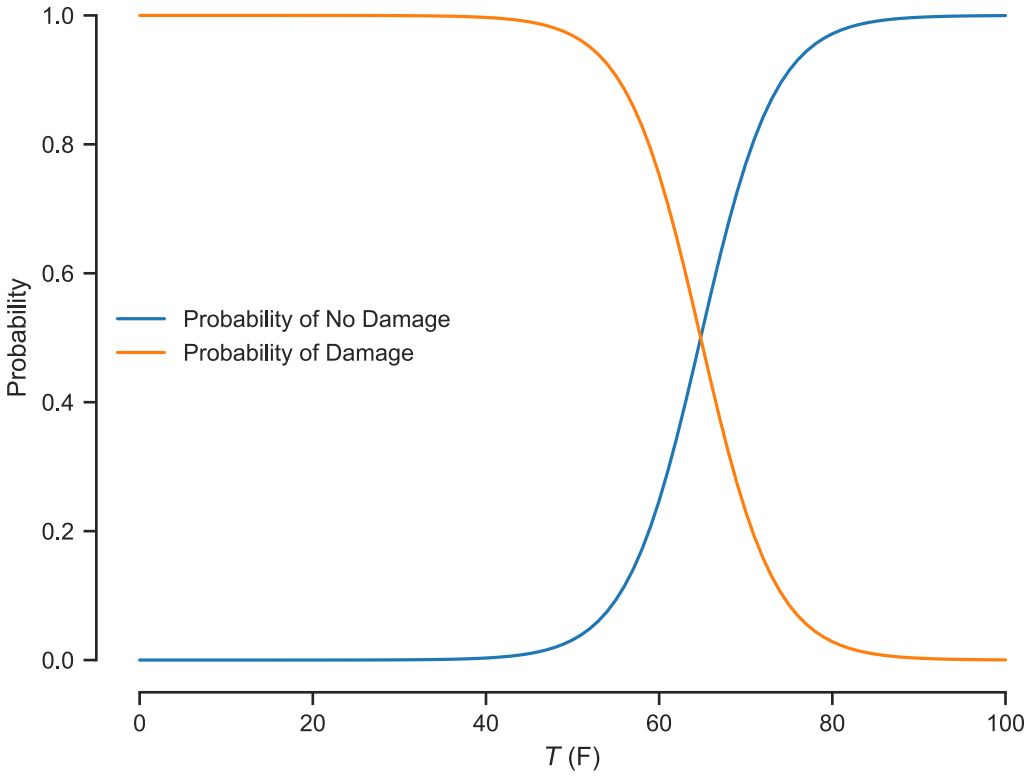
```
    if exp_c[0] > exp_c[1]:
        line += tmp + f'\t\tNo Launch'
    else:
        line += tmp + f'\t\tLaunch'
    if not (exp_c[0] > exp_c[1]) and choice_first_launch == None:
        choice_first_launch = i
    exp_c_list_0.append(exp_c[0])
    exp_c_list_1.append(exp_c[1])
    print(line)
print(f'Choice to launch if the temperature is 31 F: {choice_31_degrees}')
print(f'The first instance in which we would launch is when the temperature is at l
```

Temp	Cost lauching with damage	Cost lauching without damage	Choice
0.00	1.00	0.00	No Launch
1.00	1.00	0.00	No Launch
2.00	1.00	0.00	No Launch
3.00	1.00	0.00	No Launch
4.00	1.00	0.00	No Launch
5.00	1.00	0.00	No Launch
6.00	1.00	0.00	No Launch
7.00	1.00	0.00	No Launch
8.00	1.00	0.00	No Launch
9.00	1.00	0.00	No Launch
10.00	1.00	0.00	No Launch
11.00	1.00	0.00	No Launch
12.00	1.00	0.00	No Launch
13.00	1.00	0.00	No Launch
14.00	1.00	0.00	No Launch
15.00	1.00	0.00	No Launch
16.00	1.00	0.00	No Launch
17.00	1.00	0.00	No Launch
18.00	1.00	0.00	No Launch
19.00	1.00	0.00	No Launch
20.00	1.00	0.00	No Launch
21.00	1.00	0.00	No Launch
22.00	1.00	0.00	No Launch
23.00	1.00	0.00	No Launch
24.00	1.00	0.00	No Launch
25.00	1.00	0.00	No Launch
26.00	1.00	0.00	No Launch
27.00	1.00	0.00	No Launch
28.00	1.00	0.00	No Launch
29.00	1.00	0.00	No Launch
30.00	1.00	0.00	No Launch
31.00	1.00	0.00	No Launch
32.00	1.00	0.00	No Launch
33.00	1.00	0.00	No Launch
34.00	1.00	0.00	No Launch
35.00	1.00	0.00	No Launch
36.00	1.00	0.00	No Launch
37.00	1.00	0.00	No Launch
38.00	1.00	0.00	No Launch
39.00	1.00	0.00	No Launch
40.00	1.00	0.00	No Launch
41.00	1.00	0.00	No Launch
42.00	0.99	0.01	No Launch
43.00	0.99	0.01	No Launch
44.00	0.99	0.01	No Launch
45.00	0.99	0.01	No Launch
46.00	0.99	0.01	No Launch
47.00	0.98	0.02	No Launch
48.00	0.98	0.02	No Launch
49.00	0.98	0.02	No Launch
50.00	0.97	0.03	No Launch
51.00	0.96	0.04	No Launch
52.00	0.95	0.05	No Launch
53.00	0.94	0.06	No Launch

```
54.00   0.92            0.08            No Launch
55.00   0.91            0.09            No Launch
56.00   0.89            0.11            No Launch
57.00   0.86            0.14            No Launch
58.00   0.83            0.17            No Launch
59.00   0.79            0.21            No Launch
60.00   0.75            0.25            No Launch
61.00   0.71            0.29            No Launch
62.00   0.66            0.34            No Launch
63.00   0.60            0.40            No Launch
64.00   0.55            0.45            No Launch
65.00   0.49            0.51            Launch
66.00   0.43            0.57            Launch
67.00   0.37            0.63            Launch
68.00   0.32            0.68            Launch
69.00   0.27            0.73            Launch
70.00   0.23            0.77            Launch
71.00   0.19            0.81            Launch
72.00   0.16            0.84            Launch
73.00   0.13            0.87            Launch
74.00   0.11            0.89            Launch
75.00   0.09            0.91            Launch
76.00   0.07            0.93            Launch
77.00   0.06            0.94            Launch
78.00   0.04            0.96            Launch
79.00   0.04            0.96            Launch
80.00   0.03            0.97            Launch
81.00   0.02            0.98            Launch
82.00   0.02            0.98            Launch
83.00   0.01            0.99            Launch
84.00   0.01            0.99            Launch
85.00   0.01            0.99            Launch
86.00   0.01            0.99            Launch
87.00   0.01            0.99            Launch
88.00   0.00            1.00            Launch
89.00   0.00            1.00            Launch
90.00   0.00            1.00            Launch
91.00   0.00            1.00            Launch
92.00   0.00            1.00            Launch
93.00   0.00            1.00            Launch
94.00   0.00            1.00            Launch
95.00   0.00            1.00            Launch
96.00   0.00            1.00            Launch
97.00   0.00            1.00            Launch
98.00   0.00            1.00            Launch
99.00   0.00            1.00            Launch
100.00  0.00            1.00            Launch
Choice to launch if the temperature is 31 F: No Launch
The first instance in which we would launch is when the temperature is at least: 65.
00 F
```

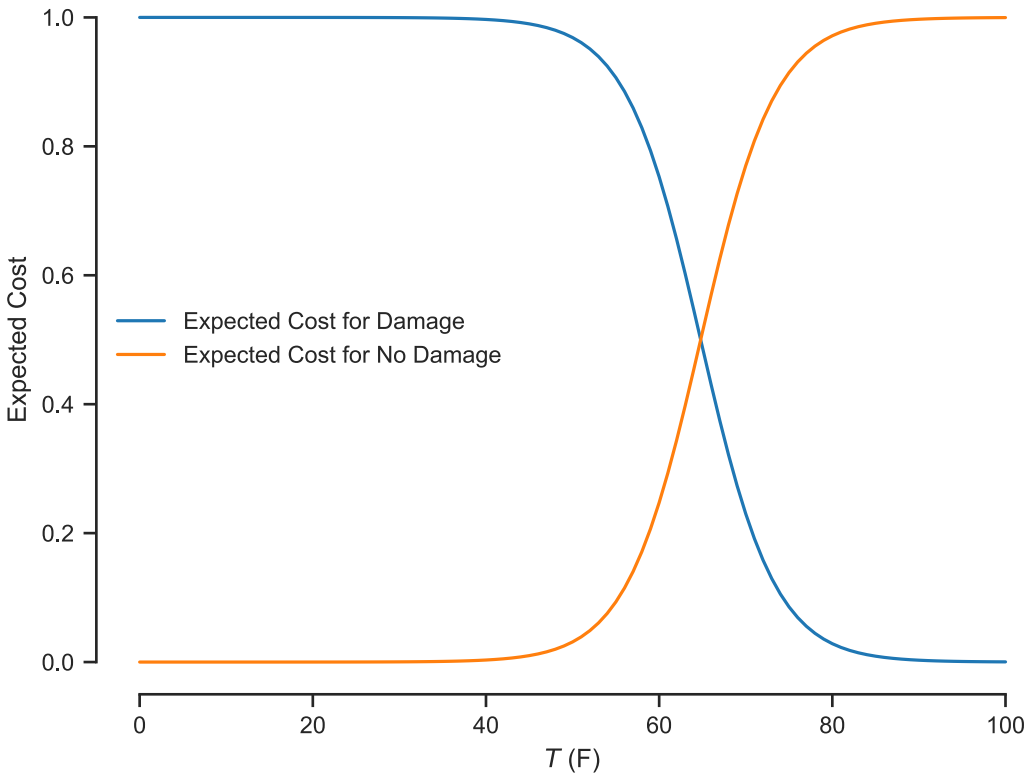
In [ ]:
```
fig, ax = plt.subplots()
ax.plot(
    xx,
    predictions[:, 0],
    label='Probability of No Damage'
```

```
)
ax.plot(
    xx,
    predictions[:, 1],
    label='Probability of Damage'
)
ax.set_xlabel('$T$ (F)')
ax.set_ylabel('Probability')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



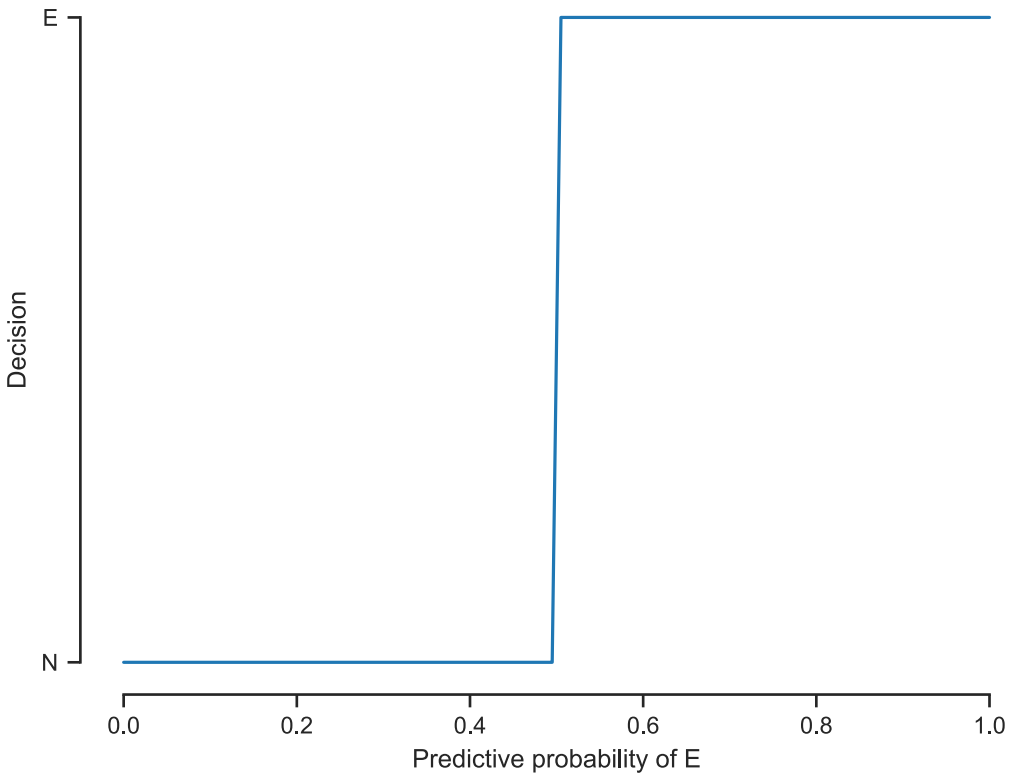
In [ ]:

```
fig, ax = plt.subplots()
ax.plot(
    xx,
    exp_c_list_0,
    label='Expected Cost for Damage'
)
ax.plot(
    xx,
    exp_c_list_1,
    label='Expected Cost for No Damage'
)
ax.set_xlabel('$T$ (F)')
ax.set_ylabel('Expected Cost')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```



In [ ]:

```
fig, ax = plt.subplots()
pE = np.linspace(0, 1, 100)
pN = 1.0 - pE
probs = np.hstack([pN[:, None], pE[:, None]])
exp_cost = np.einsum('ij,kj->ki', cost_matrix, probs)
decision_idx = np.argmin(exp_cost, axis=1)
ax.plot(pE, decision_idx)
ax.set_yticks([0, 1])
ax.set_yticklabels(['N', 'E'])
ax.set_ylabel('Decision')
ax.set_xlabel('Predictive probability of E')
sns.despine(trim=True)
```


E
N
Decision
0.0
0.2
0.4
0.6
0.8
1.0
Predictive probability of E