homework-04

October 20, 2023

1 Homework 4

1.1 References

- Lectures 13-16 (inclusive).

1.2 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.

[ ]:
```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
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)
```

1.3 Student details

- **First Name:** Stav
- **Last Name:** Zeliger
- **Email:** szeliger@purdue.edu

2 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 mannually 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 just drag and drop the file). Some other options are here.
- Downloading the file to the working directory of this notebook with this code:

[ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
↪lecturebook/data/stress_strain.txt"
download(url)
```

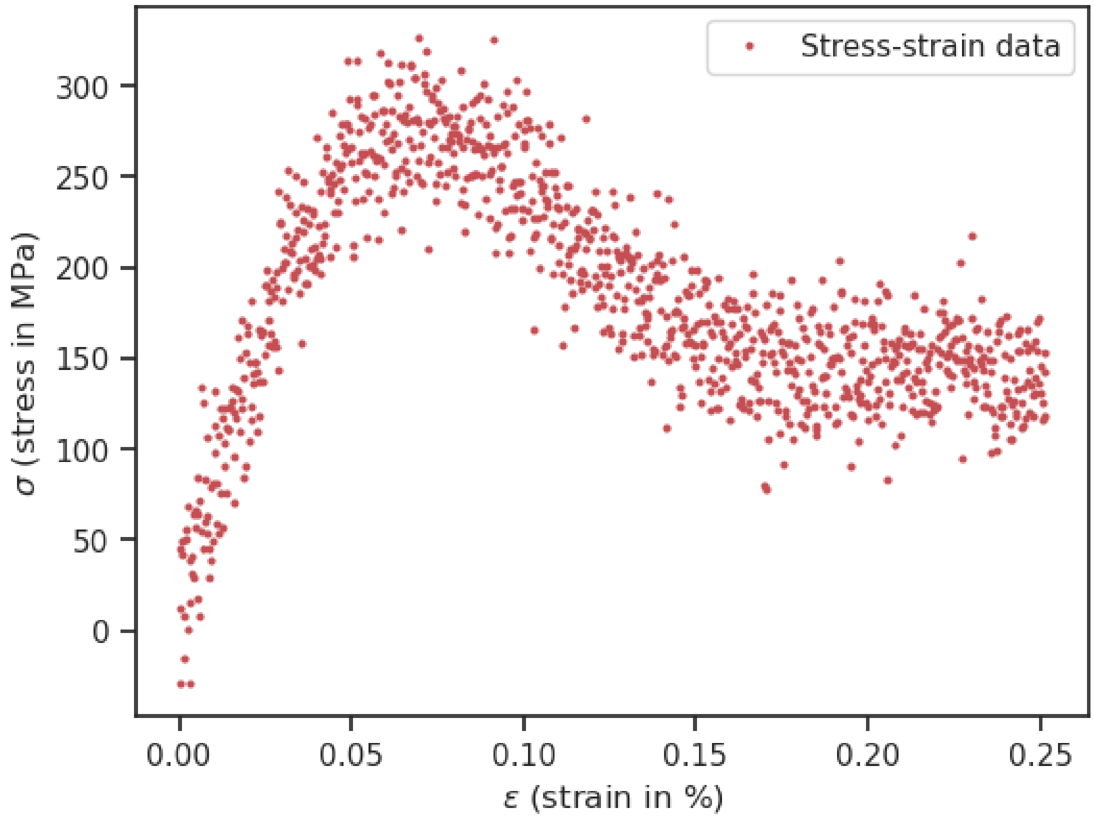
It's up to you what you choose to do. If the file is in the right place, the following code should work:

[ ]:
```
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 materail and the corresponding strain of the material (percent enlogation in the pulling direction). - Eventually the material entered the plastic regime and then it broke. Here is a visualization of the data:

[ ]:
```
# 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');
```



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

2.1 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:

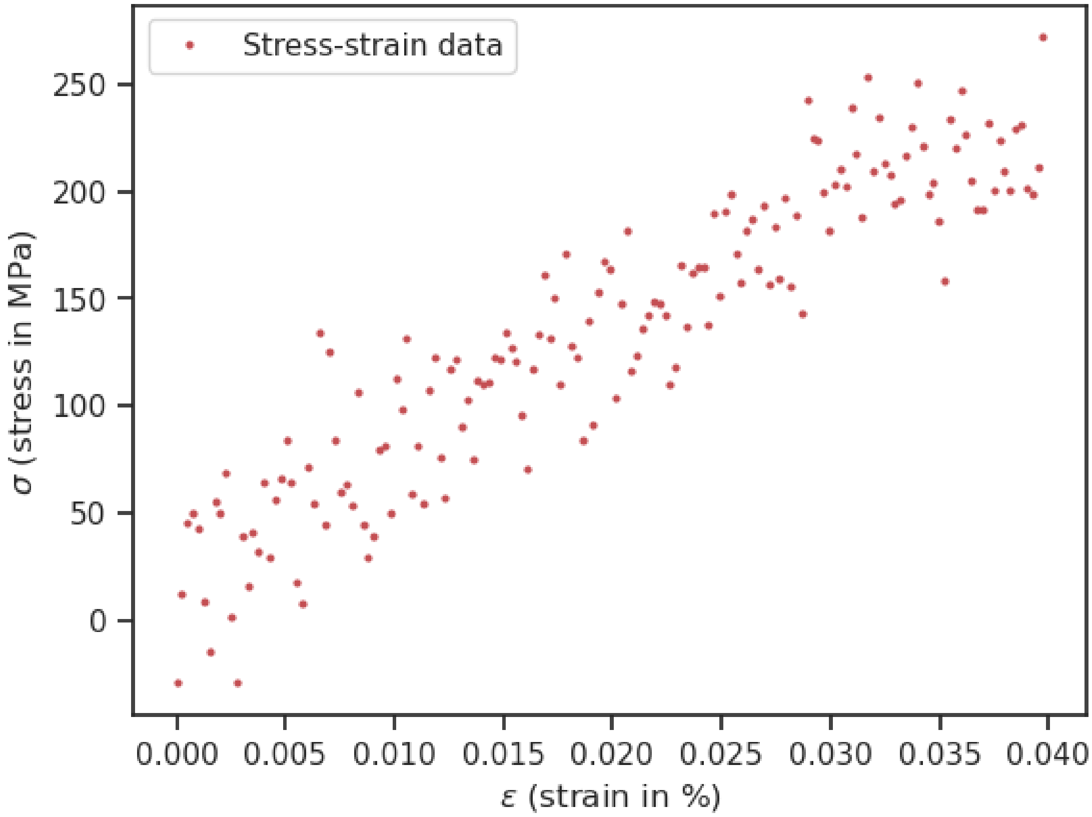
$$\sigma(\epsilon) = E\epsilon.$$

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

2.1.1 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.

[ ]:
```
# 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␣
↪straints)
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');
```



Split your data into training and validation.

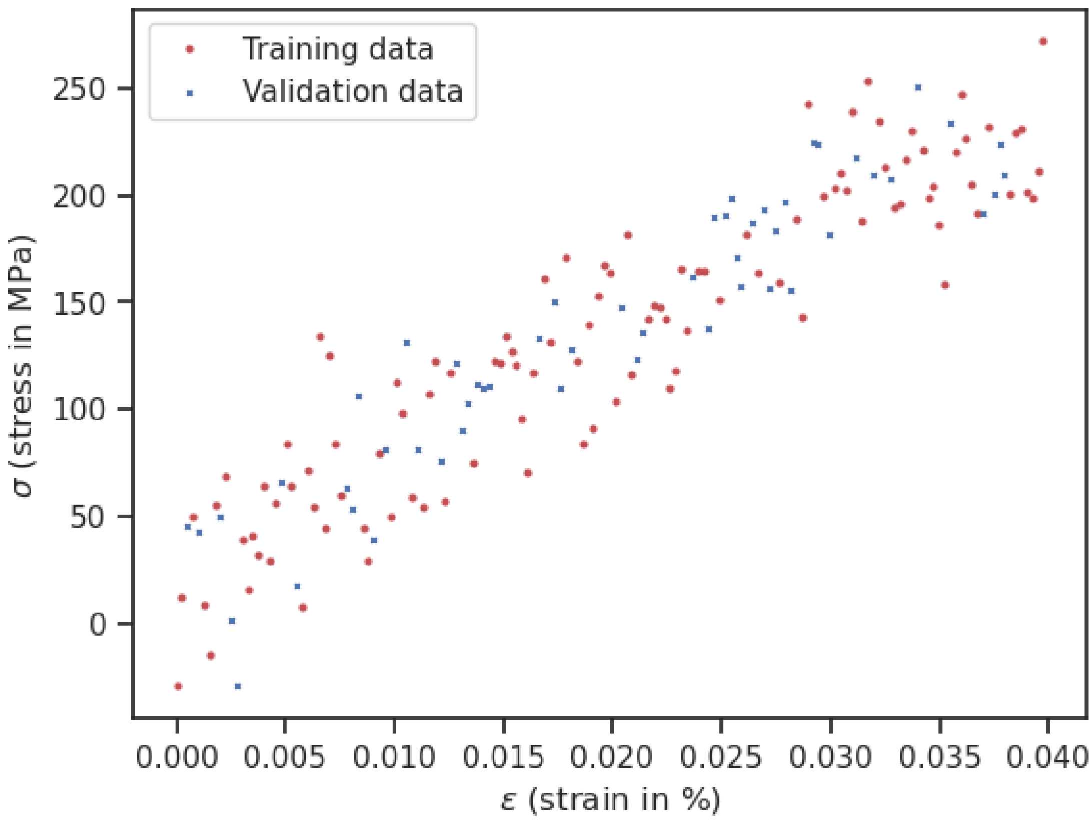
Hint: You may use sklearn.model_selection.train_test_split if you wish.

[ ]:
```
# Split the data into training and validation datasets
# Hint: Consult the lecture notes

import sklearn as skl
x_train, x_valid, y_train, y_valid = skl.model_selection.
 ↪train_test_split(x_rel,y_rel,test_size=0.33)
```

Use the following to visualize your split:

[ ]:
```
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');
```



2.1.2 Subpart A.II

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

[ ]:
```
from sklearn.linear_model import ARDRegression

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)

# Get Phi
Phi = get_polynomial_design_matrix(x_train[:, None], 1)

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

# Get Variance
sigma = np.sqrt(1.0 / regressionModel.alpha_)
lamb = regressionModel.lambda_
print(f'Sigma = {sigma}')
print(f'Hyperparameters = {lamb}')
```

```
Sigma = 27.8869185172524
Hyperparameters = [1.469e-03 3.408e-08]
```

2.1.3 Subpart A.III

Calculate the mean square error of the validation data.

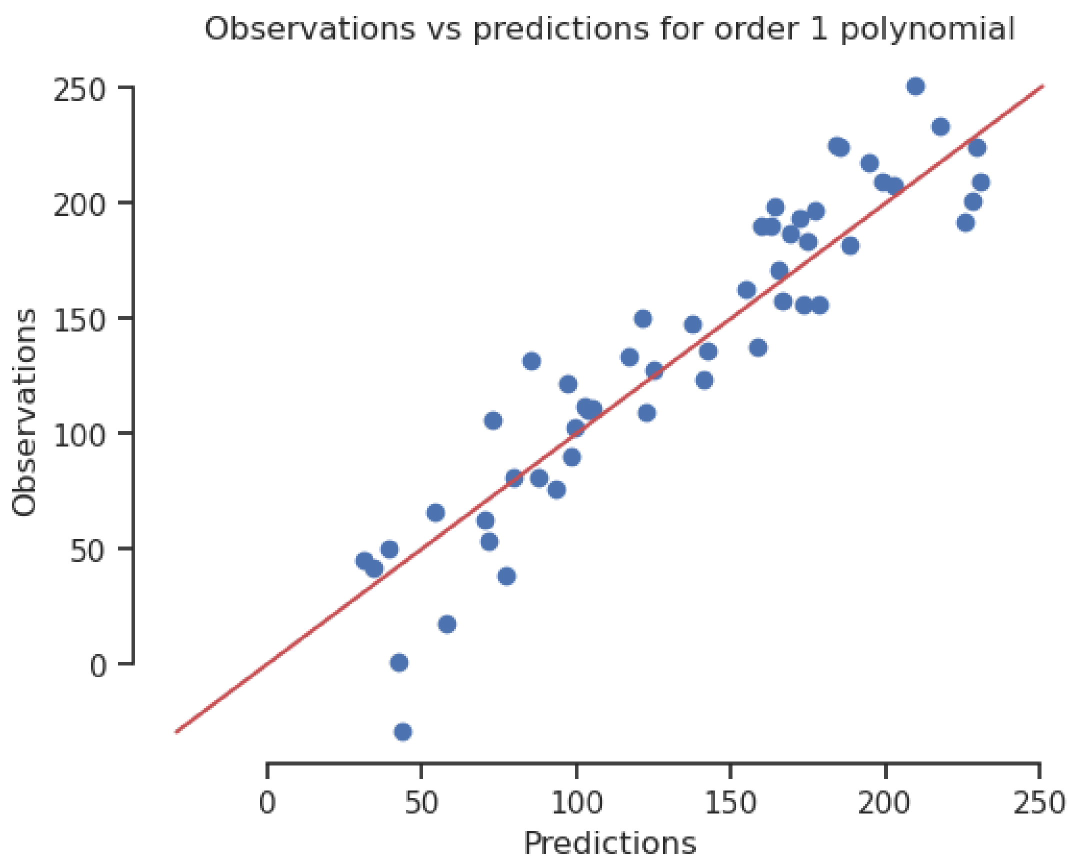
[ ]:
```
# your code here
Phi_valid = get_polynomial_design_matrix(x_valid[:, None], 1)

m = regressionModel.coef_
S = regressionModel.sigma_
w_post = st.multivariate_normal(
    mean=m,
    cov=S + np.eye(S.shape[0])
)
w = w_post.rvs()


y_prediction = Phi_valid @ w
error= np.mean((y_prediction - y_valid)**2)
print(f'Mean Squared Error: {error}')
```

```
Mean Squared Error: 585.0597030865363
```

2.1.4 Subpart A.IV

Make the observations vs predictions plot for the validation data.

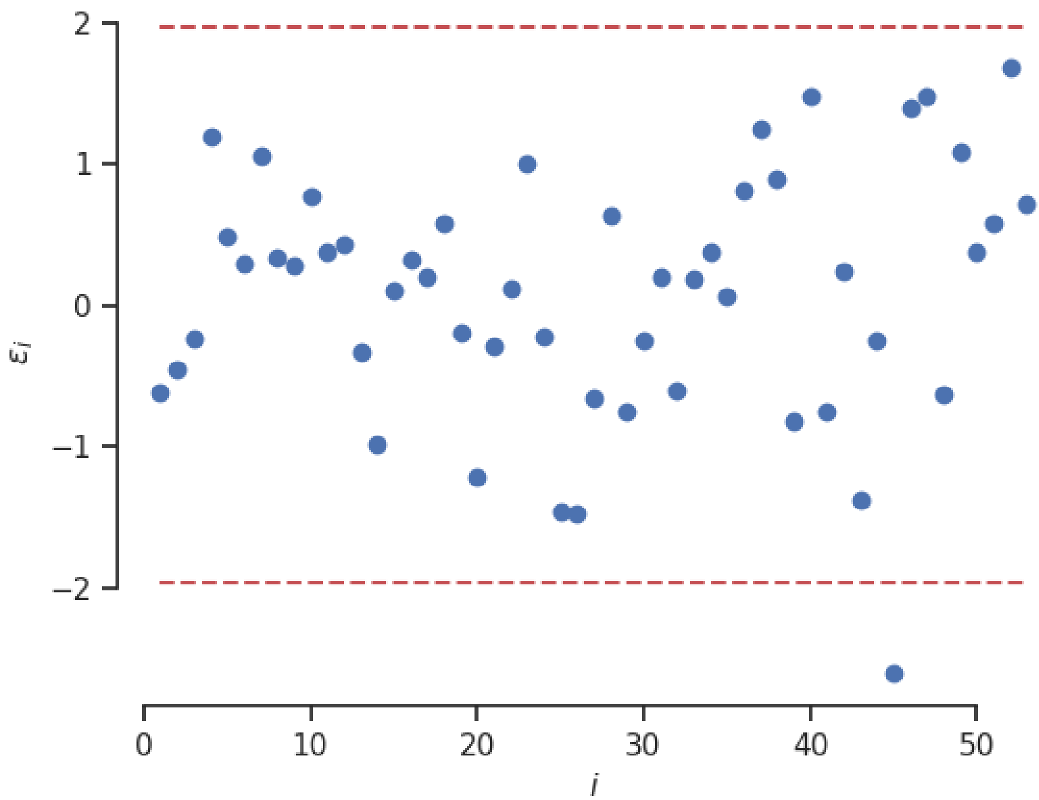
[ ]:
```
# your code here

plt.figure()
plt.plot(y_prediction,y_valid,'bo',)
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
plt.plot(yys, yys, 'r-')

plt.xlabel('Predictions')
plt.ylabel('Observations')
plt.title('Observations vs predictions for order 1 polynomial')
sns.despine(trim=True)
```



2.1.5 Subpart A.V

Compute and plot the standarized errors for the validation data.

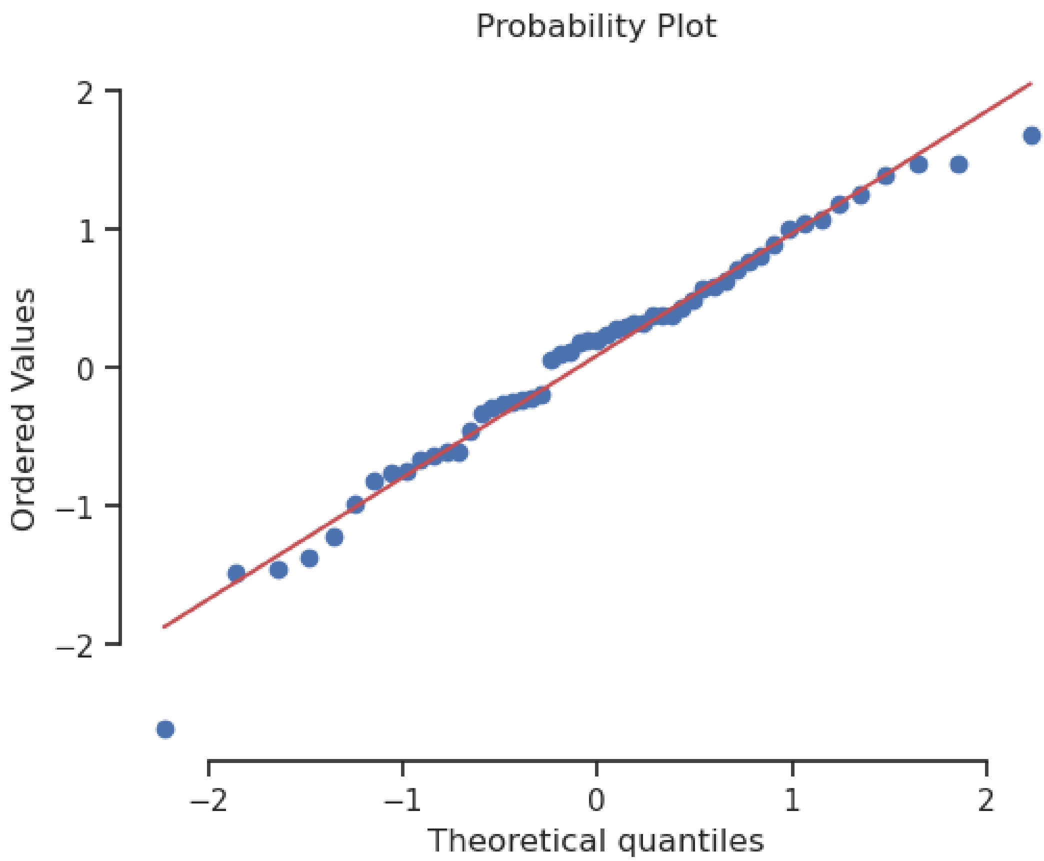
[ ]:
```
# your code here
eps = (y_valid - y_prediction) / sigma

idx = np.arange(1, eps.shape[0] + 1)

fig, ax = plt.subplots()
ax.plot(idx, eps, 'o', label='Standarized errors')
ax.plot(idx, 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(idx, -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$')
sns.despine(trim=True);
```



2.1.6 Subpart A.VI

Make the quantile-quantile plot of the standarized errors.

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



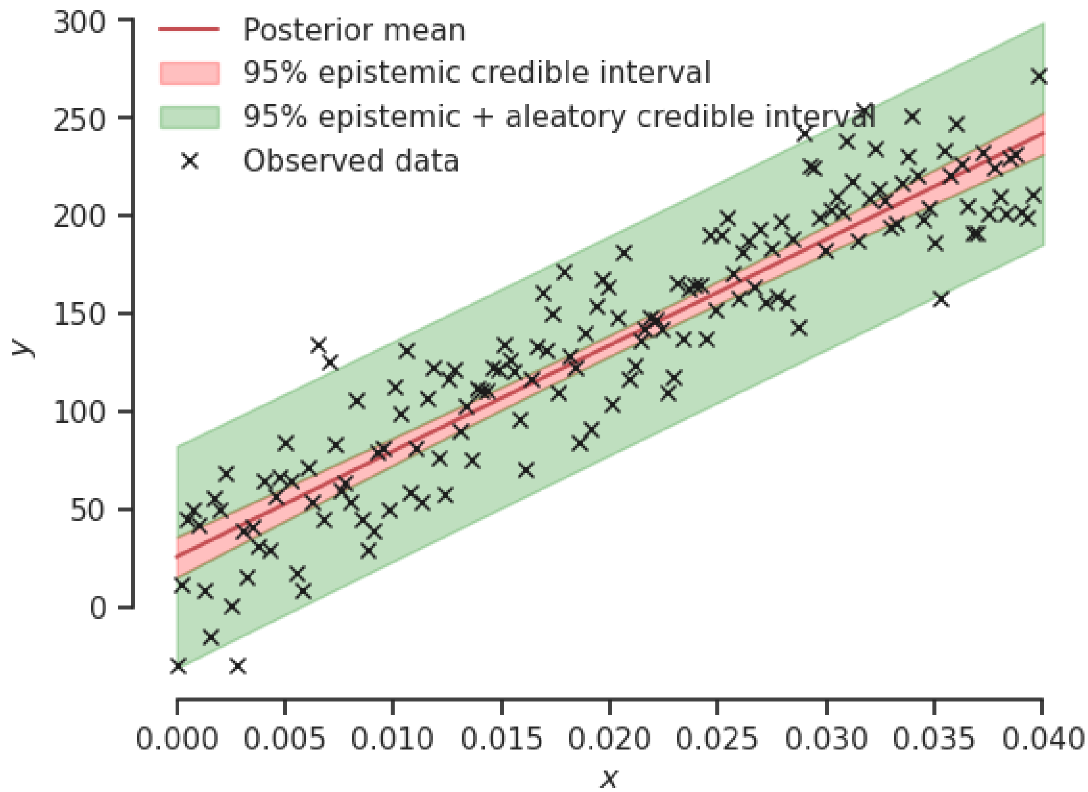
2.1.7 Subpart A.VII

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

[ ]:
```python
# your code here
def plot_posterior_predictive(
    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_rel, y_rel, 'kx', label='Observed data')
    if y_true is not None:
        ax.plot(xx, y_true, "--", label="True response")
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    plt.legend(loc="upper left", frameon=False)
    sns.despine(trim=True)




xx = np.linspace(0, 0.04, 100)

plot_posterior_predictive(
    regressionModel,
    xx,
    get_polynomial_design_matrix,
    phi_func_args=(1,)
)
```



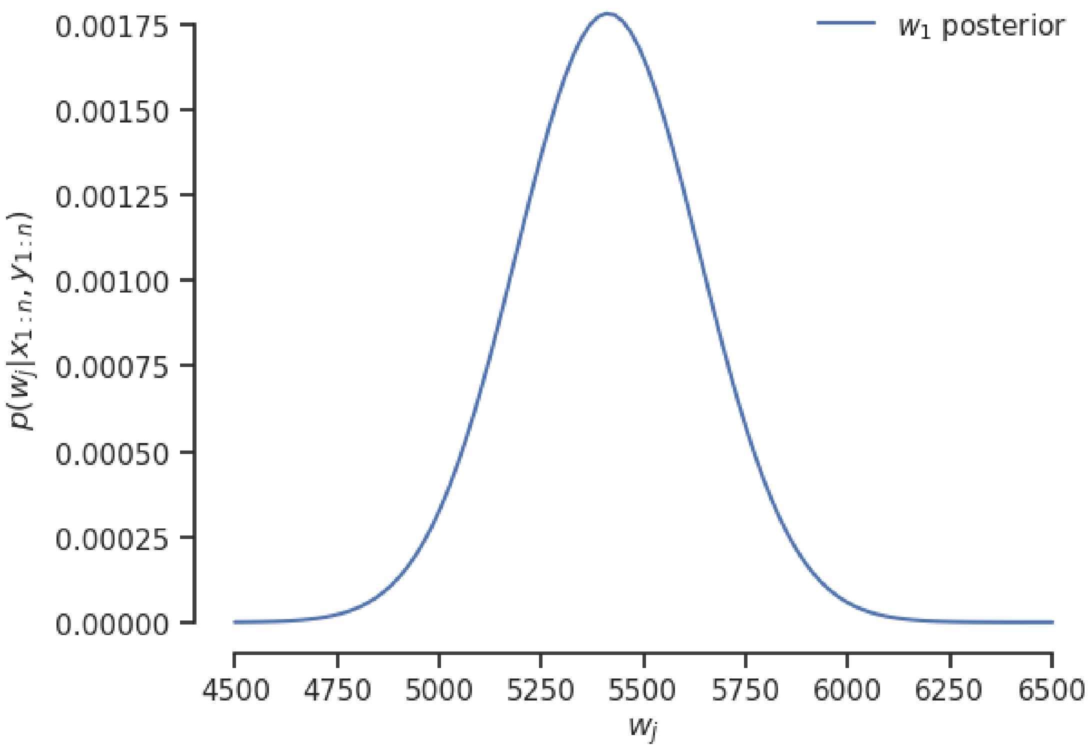
2.1.8 Subpart A. VIII

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

[ ]:
```
# your code here
import scipy.stats as st

sigma = regressionModel.sigma_
print(sigma)
m = regressionModel.coef_
ww = np.linspace(4500, 6500, 100)


fig, ax = plt.subplots()
wj_post = st.norm(
    loc=m[1],
    scale=np.sqrt(sigma[1,1])
)
ax.plot(
    ww,
    wj_post.pdf(ww),
```

```
    label=f'$w_{{{1}}}$ posterior')
ax.set_xlabel("$w_j$")
ax.set_ylabel("$p(w_j|x_{{1:n}}, y_{{1:n}})$")
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```

```
[[ 2.664e+01 -9.851e+02]
 [-9.851e+02  5.008e+04]]
```



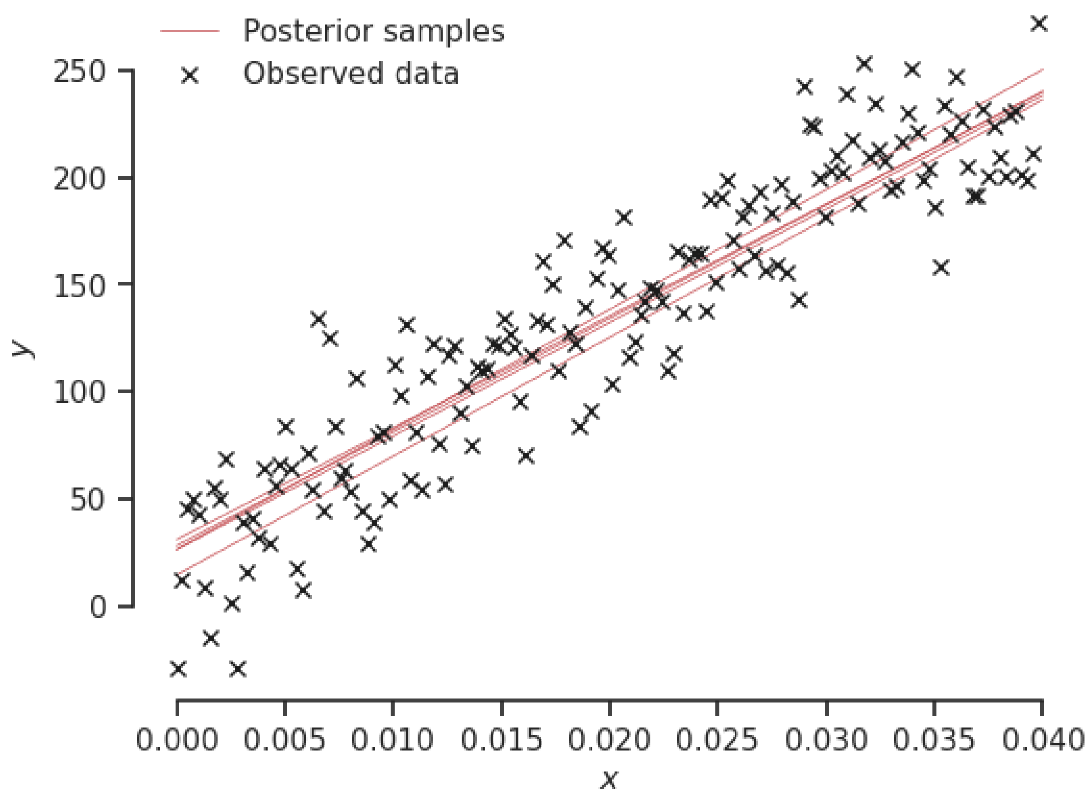
2.1.9 Subpart A.IX

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

[ ]:
```
# your code here

def plot_posterior_samples(
    model,
    xx,
    phi_func,
    phi_func_args=(),
    num_samples=10,
    y_true=None,
```

```python
    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_rel, y_rel, 'kx', label='Observed data')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    plt.legend(loc="best", frameon=False)
    sns.despine(trim=True)

plot_posterior_samples(
    regressionModel,
    xx,
    get_polynomial_design_matrix,
```

```
    phi_func_args=(1,),
    y_true=None,
    num_samples=5
)
```



2.1.10 Subpart A.X

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

[ ]:
```
# your code here
lb = wj_post.ppf(0.05)
ub = wj_post.ppf(0.95)

print(f'lower bound: {lb}')
print(f'upper bound: {ub}')
```

```
lower bound: 5044.555882992406
upper bound: 5780.7513388955695
```

2.1.11 Subpart A.XI

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

[ ]:
```python
YoungModulus = np.mean([ub,lb])
print(f'Mean of upper and lower bound: {YoungModulus}')
```

```
Mean of upper and lower bound: 5412.653610943988
```

I would choose this value for E, since it is the mean of the upper and lower bound of the 95% credible interval for it.

Your answer here

2.2 Part B - Estimate the ultimate strength

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

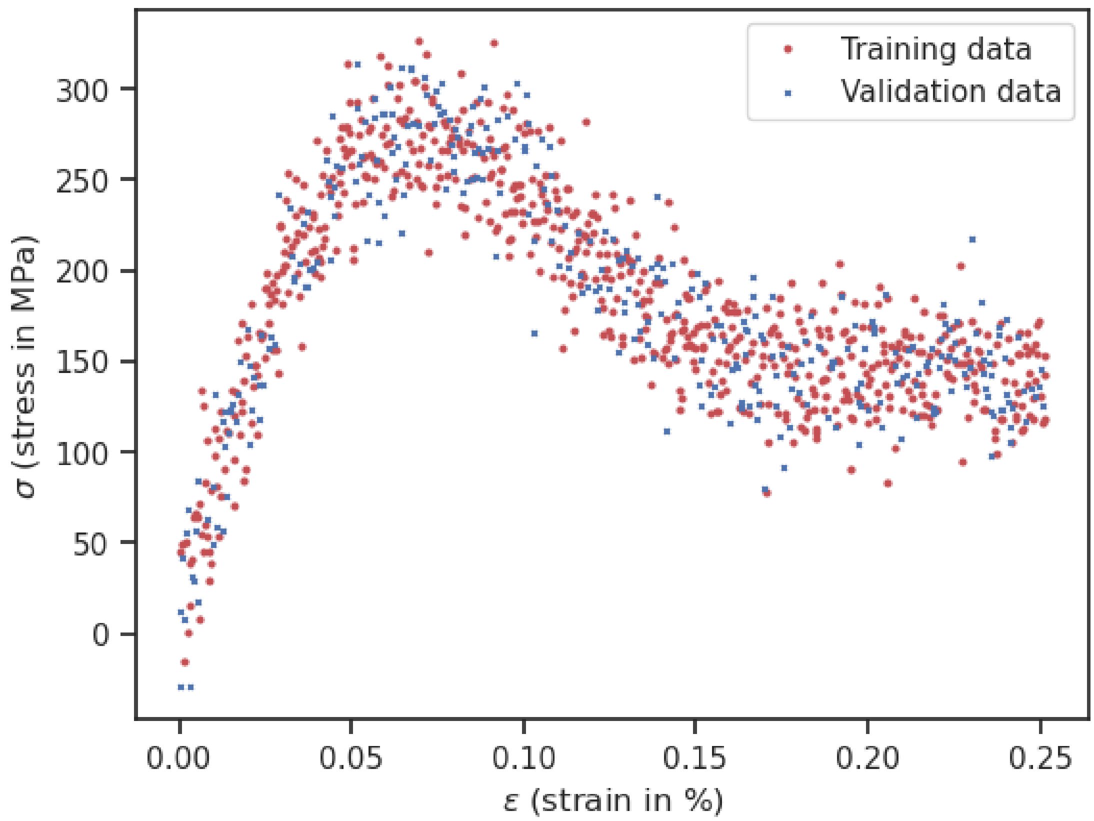
2.2.1 Subpart B.I - Extract training and validation data

Extract training and validation data from the entire dataset.

[ ]:
```python
# your code here - Repeat as many text and code blocks as you like
x_train, x_valid, y_train, y_valid = skl.model_selection.
 ↪train_test_split(x,y,test_size=0.33)
```

Use the following to visualize your split:

[ ]:
```python
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');
```



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

To do this, we will set up a generalized linear model that can 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 clearly has a well-defined elastic regime (see Part A).

I am going to help you set up the right model. We are goint to 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:

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

The reason we include the first derivative is so that 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 $w_2, \dots, w_d$. 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 end result is:

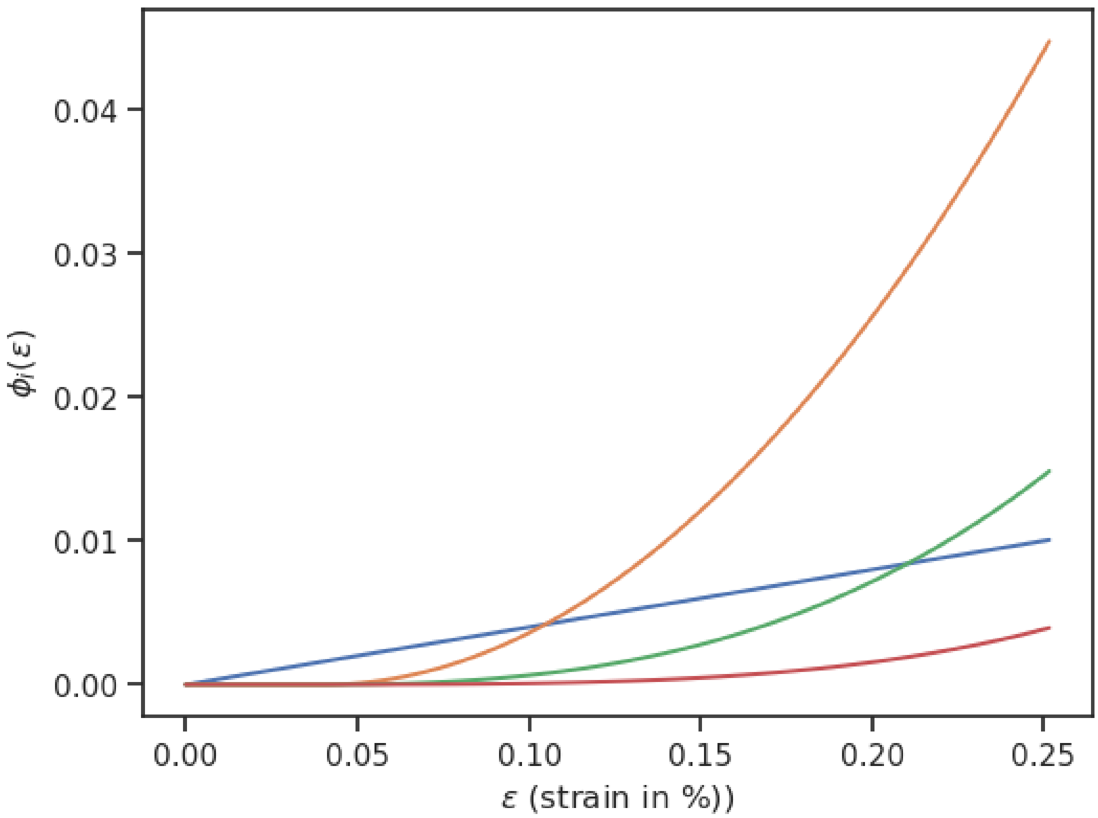
$$f(\epsilon) = E\epsilon + \sum_{i=2}^{d} w_i \left[(i-1)\epsilon_l^i - i\epsilon\epsilon_l^{i-1} + \epsilon^i\right] H(\epsilon - \epsilon_l).$$

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

[ ]:
```
# 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 dimensional'
    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_l*Epsilon
    for i in range(2, d+1):
        Phi[:, i-1] = ((i-1)*epsilon_l**i -␣
↪i*Epsilon*epsilon_l**(i-1)+Epsilon**i) * Step
    return Phi
```

Visualize the basis functions here:

[ ]:
```
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('$\phi_i(\epsilon)$');
```



2.2.3 Subpart B.III

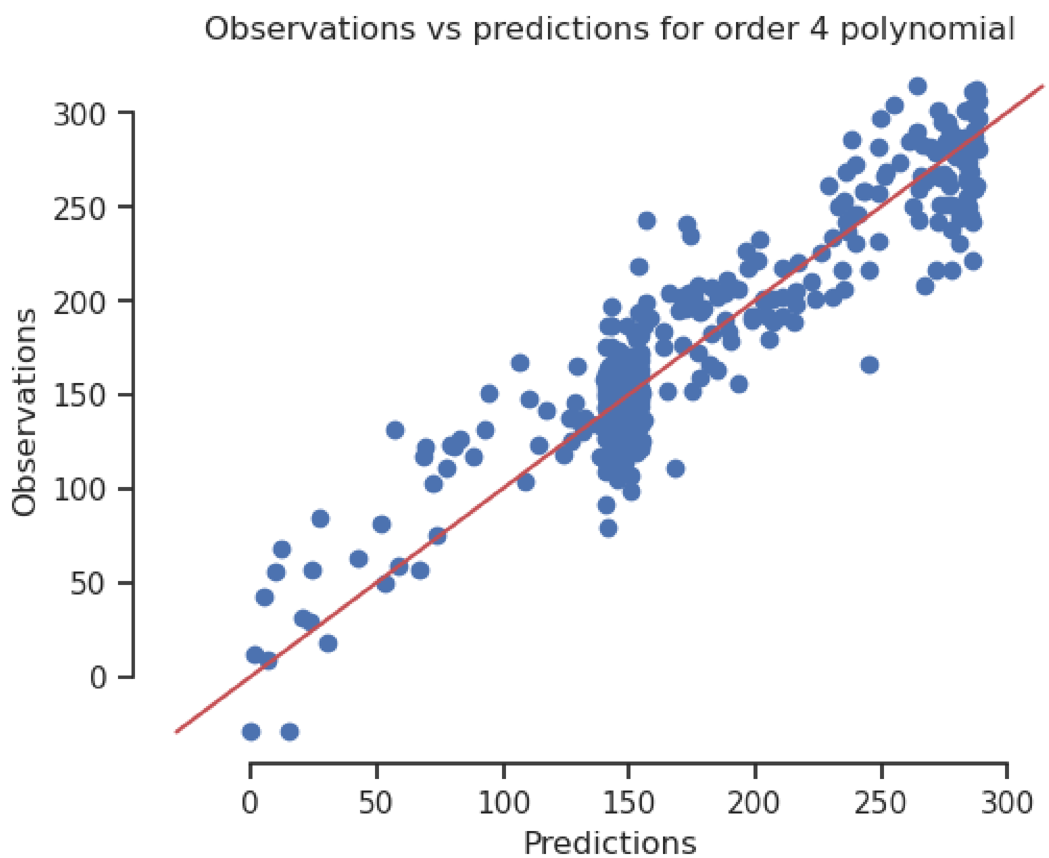
Fit the model using automatic relevance determination and demonstrate that it works well by doing all the things we did above (MSE, observations vs predictions plot, standarized errors, etc.).

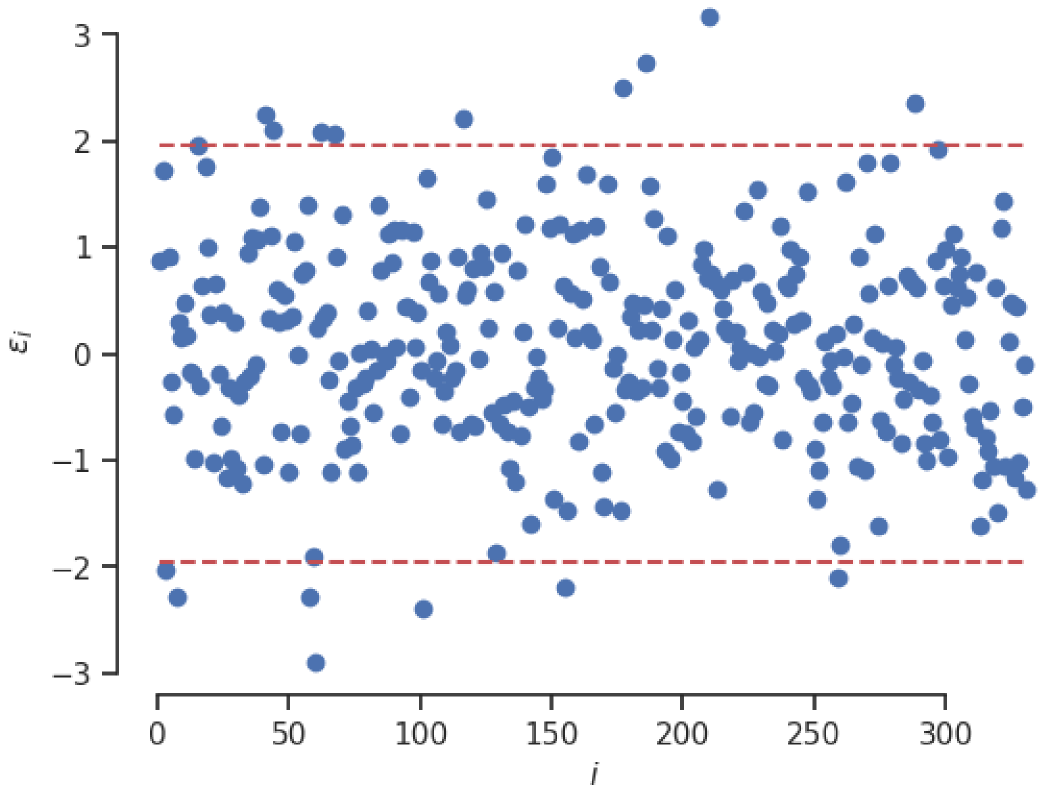
[ ]:
```
#### Sigma and Hyperparams #####
# Get Phis
Phis = compute_design_matrix(x_train, epsilon_l, d)

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

# Get Variance
sigma = np.sqrt(1.0 / model.alpha_)
lamb = model.lambda_
print(f'Sigma = {sigma}')
print(f'Hyperparameters = {lamb}')


#### Mean Squared Error ####
```

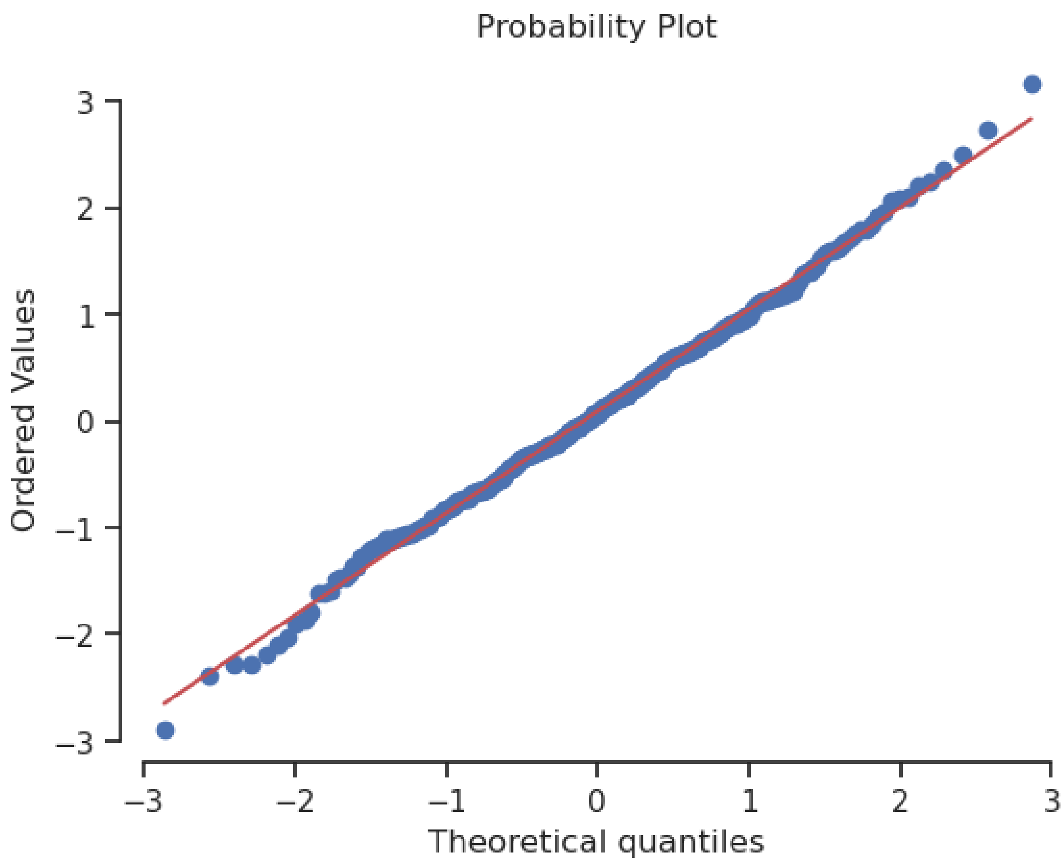
```
Phis_validation = compute_design_matrix(x_valid, epsilon_l, d)

m = model.coef_
S = model.sigma_
w_post = st.multivariate_normal(
    mean=m,
    cov=S + np.eye(S.shape[0])
)
w = w_post.rvs()


y_prediction = Phis_validation @ w
error= np.mean((y_prediction - y_valid)**2)
print(f'Mean Squared Error: {error}')


##### Observations vs Predictions ####
plt.figure()
plt.plot(y_prediction,y_valid,'bo',)
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
plt.plot(yys, yys, 'r-')

plt.xlabel('Predictions')
plt.ylabel('Observations')
plt.title(f'Observations vs predictions for order {d} polynomial')
sns.despine(trim=True)


#### Standardized Errors ####
eps = (y_valid - y_prediction) / sigma

idx = np.arange(1, eps.shape[0] + 1)

fig, ax = plt.subplots()
ax.plot(idx, eps, 'o', label='Standarized errors')
ax.plot(idx, 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(idx, -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$')
sns.despine(trim=True);


fig, ax = plt.subplots()
st.probplot(eps, dist=st.norm, plot=ax)
```

```
sns.despine(trim=True)
```

```
Sigma = 27.245588788453265
Hyperparameters = [5.416e-11 1.691e-11 7.844e-13 3.076e-13]
Mean Squared Error: 678.3994288343642
```




3
2
1
0
−1
−2
−3
ε_i
0
50
100
150
200
250
300
i



2.2.4 Subpart B.IV

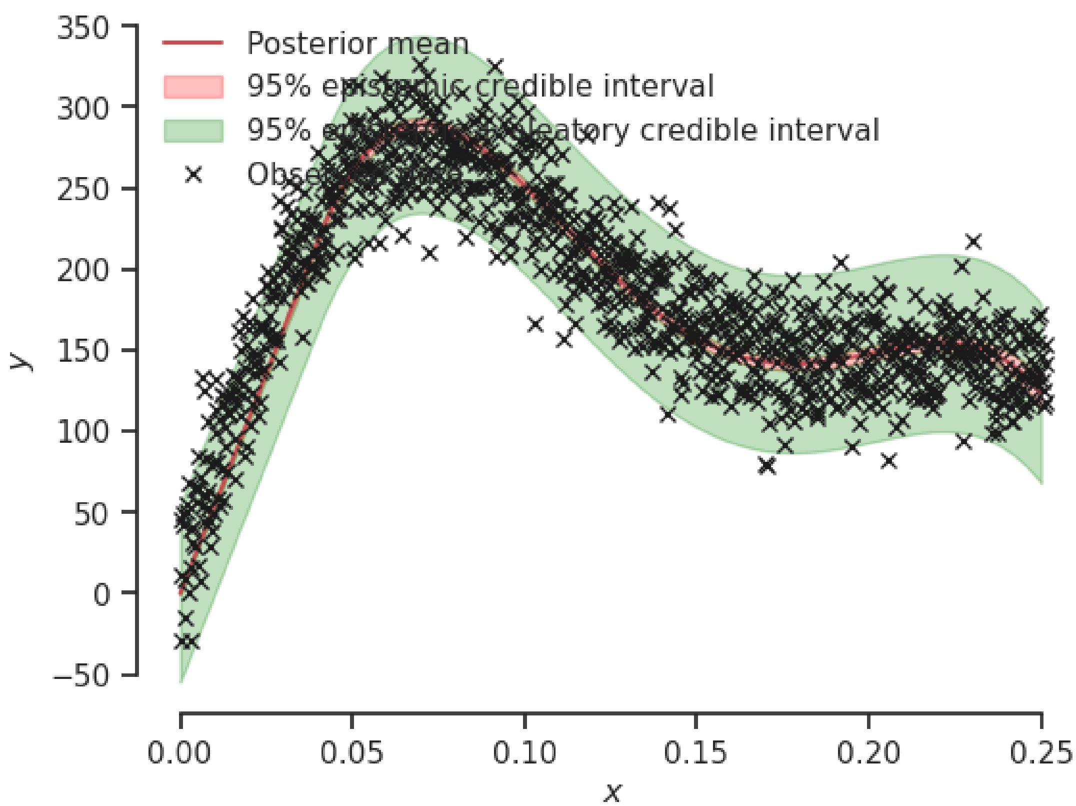
Visualize epistemic and aleatory uncertainty in the stess-strain relation.

[ ]:
```
# Your code here
def plot_posterior_predictive(
    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,
        *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, y, 'kx', label='Observed data')
    if y_true is not None:
        ax.plot(xx, y_true, "--", label="True response")
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    plt.legend(loc="upper left", frameon=False)
    sns.despine(trim=True)
xx = np.linspace(0, 0.25, y.size)
print(epsilon_l)
plot_posterior_predictive(
    model,
    xx,
    compute_design_matrix,
    phi_func_args=(epsilon_l,d),
    y_true=None
)
```

```
0.04
```



2.2.5 Subpart B.V - Extract the ultimate strength

Now, you are going to 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.

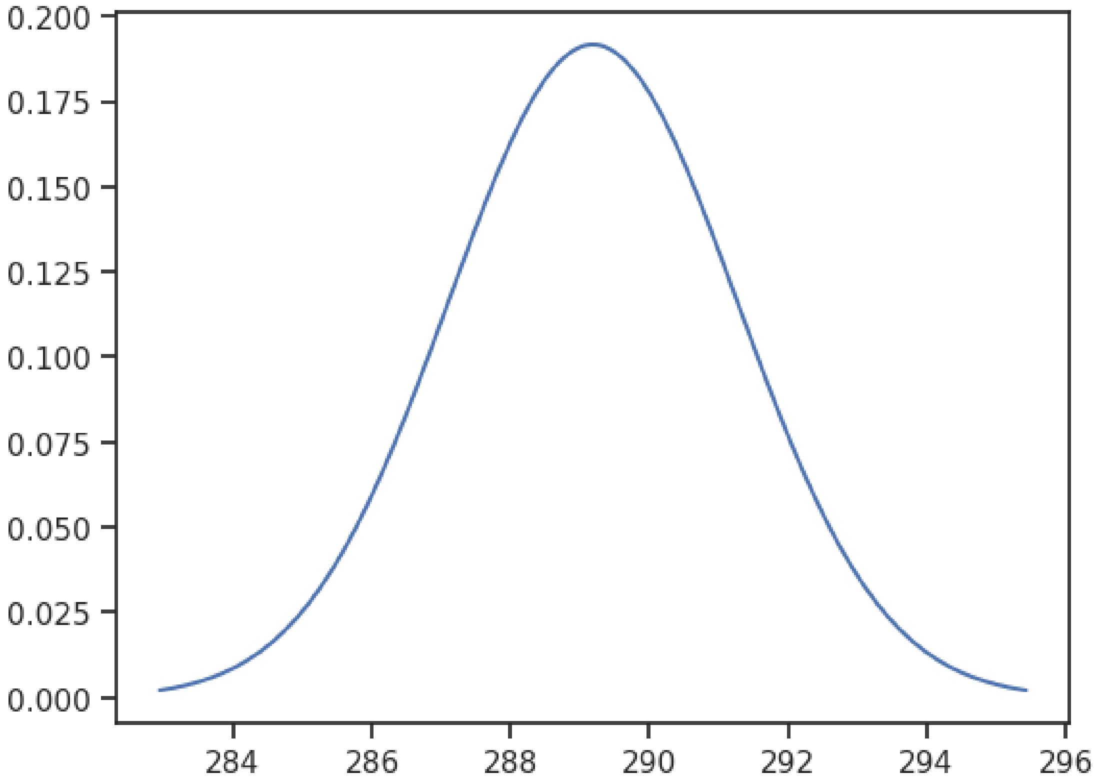
Do the following: - Visualize posterior of the ultimate strength. - Find a 95% credible interval for the ultimate strength. - Pick a value for the ultimate strength.

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.

[ ]:
```
xx = np.linspace(0,0.25,10000)
num_samples = 10000
Phi_xx = compute_design_matrix(xx,epsilon_l,d)
samples = np.zeros(num_samples)
m = model.coef_
S = model.sigma_
w_post = st.multivariate_normal(
    mean=m,
    cov=S + np.eye(S.shape[0])
)
for i in range(num_samples):
    w_sample = w_post.rvs()
    yy_sample = Phi_xx @ w_sample
    samples[i] = np.max(yy_sample)

strengthMean = np.mean(samples)
strengthSigma = np.var(samples)

strength_rv = st.norm(
    loc=strengthMean,
    scale=np.sqrt(strengthSigma)
)

ww = np.linspace(strengthMean - 3*np.sqrt(strengthSigma),strengthMean + 3*np.
 ↪sqrt(strengthSigma),1000)

fig, ax = plt.subplots()
ax.plot(
    ww,
    strength_rv.pdf(ww),
    label=f'$w_{{{1}}}$ posterior')
lb = wj_post.ppf(0.05)
ub = wj_post.ppf(0.95)
```

```
print(f'95% credibile interval [{lb}, {ub}], found using {num_samples} samples')
```

```
95% credibile interval [285.78374458582954, 292.58126472987334], found using
10000 samples
```



3 Problem 2 - Optimizing the performance of a compressor

In this problem we are going to 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:

```
[ ]: url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
       ↪lecturebook/data/compressor_data.xlsx"
     download(url)
```

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

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

[ ]:
```
    T_e  DT_sh  T_c  DT_sc  T_amb   f  m_dot    m_dot.1  Capacity  Power  \
0   -30     11   25      8     35  60   28.8   8.000000      1557    901
1   -30     11   30      8     35  60   23.0   6.388889      1201    881
2   -30     11   35      8     35  60   17.9   4.972222       892    858
3   -25     11   25      8     35  60   46.4  12.888889      2509   1125
4   -25     11   30      8     35  60   40.2  11.166667      2098   1122
..  …      …  …    …      …  ..    …        …        …     …
60   10     11   45      8     35  60  245.2  68.111111     12057   2525
61   10     11   50      8     35  60  234.1  65.027778     10939   2740
62   10     11   55      8     35  60  222.2  61.722222      9819   2929
63   10     11   60      8     35  60  209.3  58.138889      8697   3091
64   10     11   65      8     35  60  195.4  54.277778      7575   3223

    Current   COP  Efficiency
0       4.4  1.73       0.467
1       4.0  1.36       0.425
2       3.7  1.04       0.382
3       5.3  2.23       0.548
4       5.1  1.87       0.519
..      …   …         …
60     11.3  4.78       0.722
61     12.3  3.99       0.719
62     13.1  3.35       0.709
63     13.7  2.81       0.693
64     14.2  2.35       0.672

[65 rows x 13 columns]
```

The data are part of a 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. Our goal is to learn the map between T_e and T_c and measured Capacity and Power (both in W). First, let's see how you can extract only the relevant data.

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

[ ]:
```
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]])
```

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

[ ]:
```
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])
```

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

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

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 make sure that your models make sense. + Use ARD Regression to fit any hyperparameters and the noise.

3.0.1 Subpart A.I - Fit the capacity

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

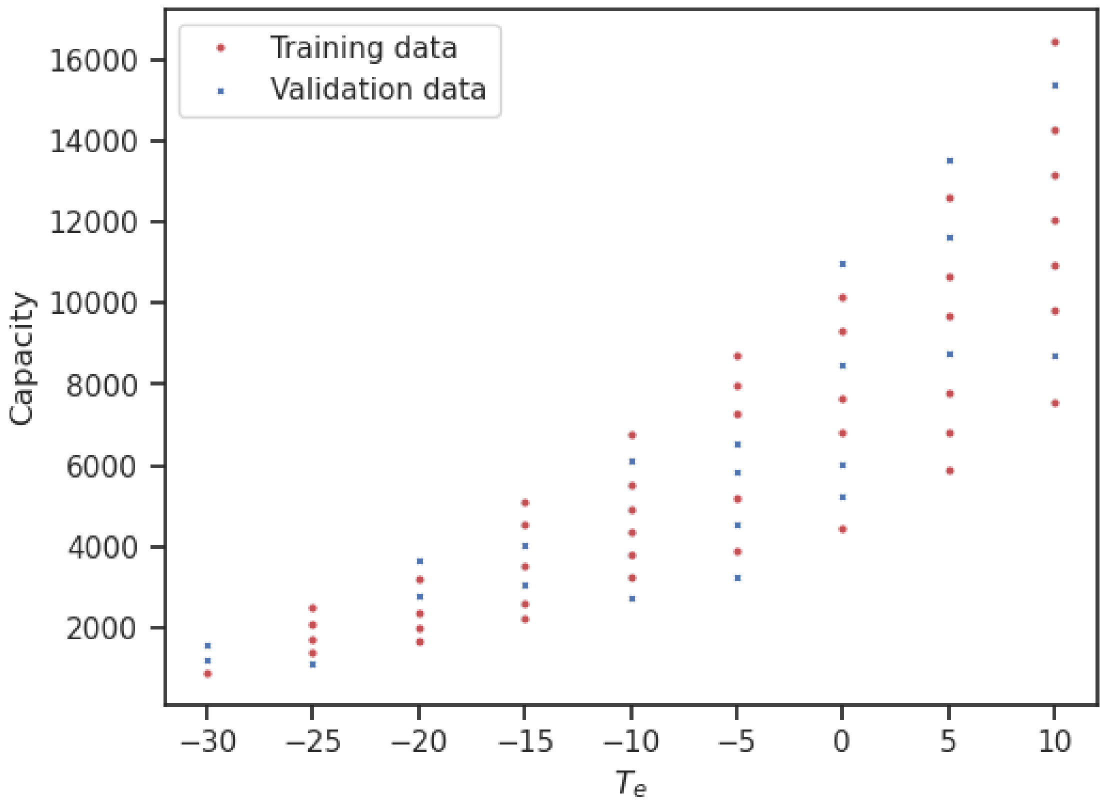
[ ]:
```
# your code here - Repeat as many text and code blocks as you like
x_train, x_valid, y_train, y_valid = skl.model_selection.
 ↪train_test_split(x,y,test_size=0.33)

#### Visaulise training data ####
plt.figure()
plt.plot(
    x_train[:,0],
    y_train,
```

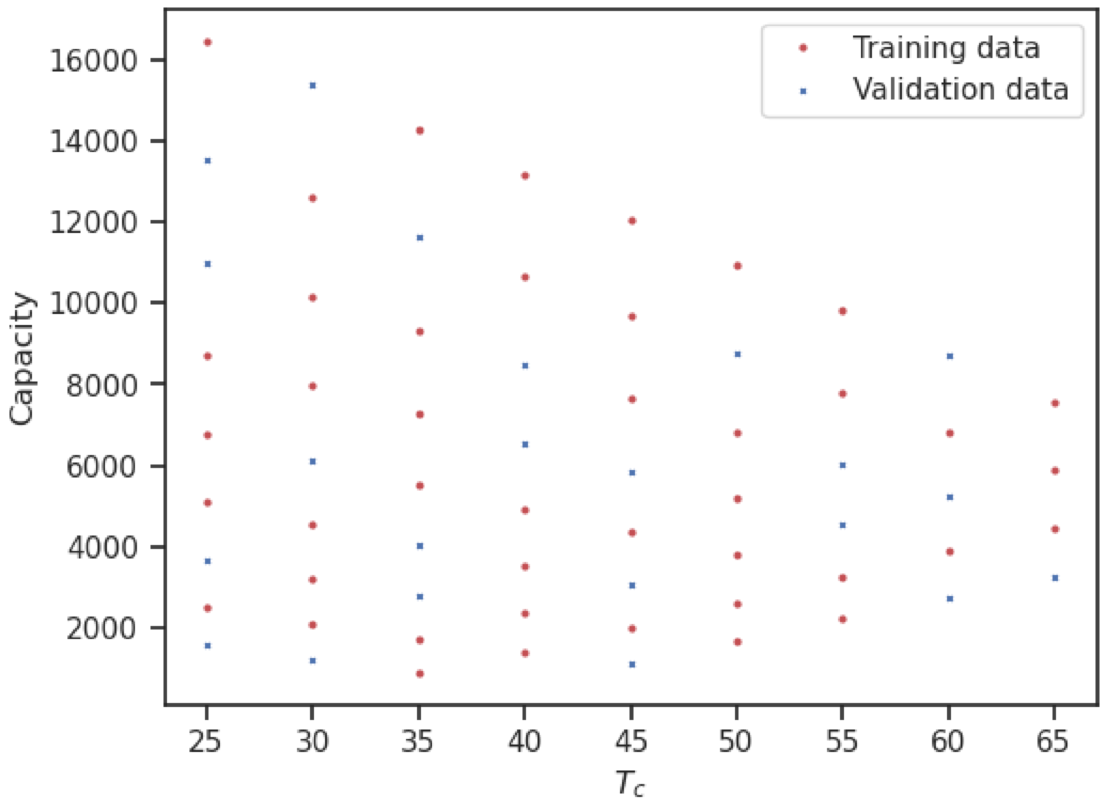
```
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid[:,0],
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.ylabel('Capacity')
plt.xlabel('$T_e$')
plt.legend(loc='best');


plt.figure()
plt.plot(
    x_train[:,1],
    y_train,
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid[:,1],
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.ylabel('Capacity')
plt.xlabel('$T_c$')
plt.legend(loc='best');


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.ylabel('Capacity')
plt.xlabel('$T_c$ and $T_e$')
plt.legend(loc='best');


#### Fitting Our Model and checking variance####
d = 10
import sklearn.preprocessing

polynomial = sklearn.preprocessing.PolynomialFeatures(d)
Phis = polynomial.fit_transform(x_train)

model = ARDRegression(fit_intercept = False).fit(Phis,y_train)

# Get Variance
sigma = np.sqrt(1.0 / model.alpha_)
lamb = model.lambda_
print(f'Sigma = {sigma}')
print(f'Hyperparameters = {lamb}')


#### Mean Squared Error ####
Phis_validation = polynomial.fit_transform(x_valid)
w = model.coef_

y_prediction = Phis_validation @ w
error= np.mean((y_prediction - y_valid)**2)
print(f'Mean Squared Error: {error}')


##### Observations vs Predictions ####
plt.figure()
plt.plot(y_prediction,y_valid,'bo',)
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
plt.plot(yys, yys, 'r-')

plt.xlabel('Predictions')
plt.ylabel('Observations')
plt.title(f'Observations vs predictions for order {d} polynomial')
sns.despine(trim=True)
```

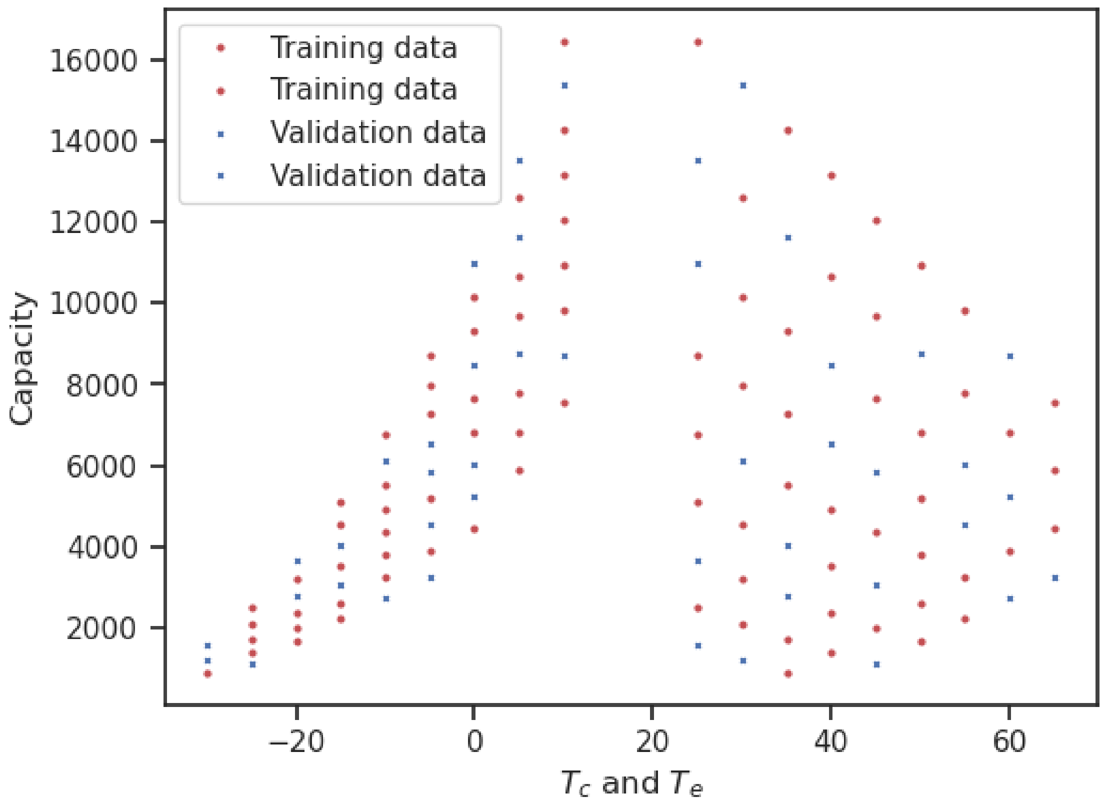
```python
#### Standardized Errors ####
eps = (y_valid - y_prediction) / sigma

idx = np.arange(1, eps.shape[0] + 1)

fig, ax = plt.subplots()
ax.plot(idx, eps, 'o', label='Standarized errors')
ax.plot(idx, 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(idx, -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$')
sns.despine(trim=True);


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

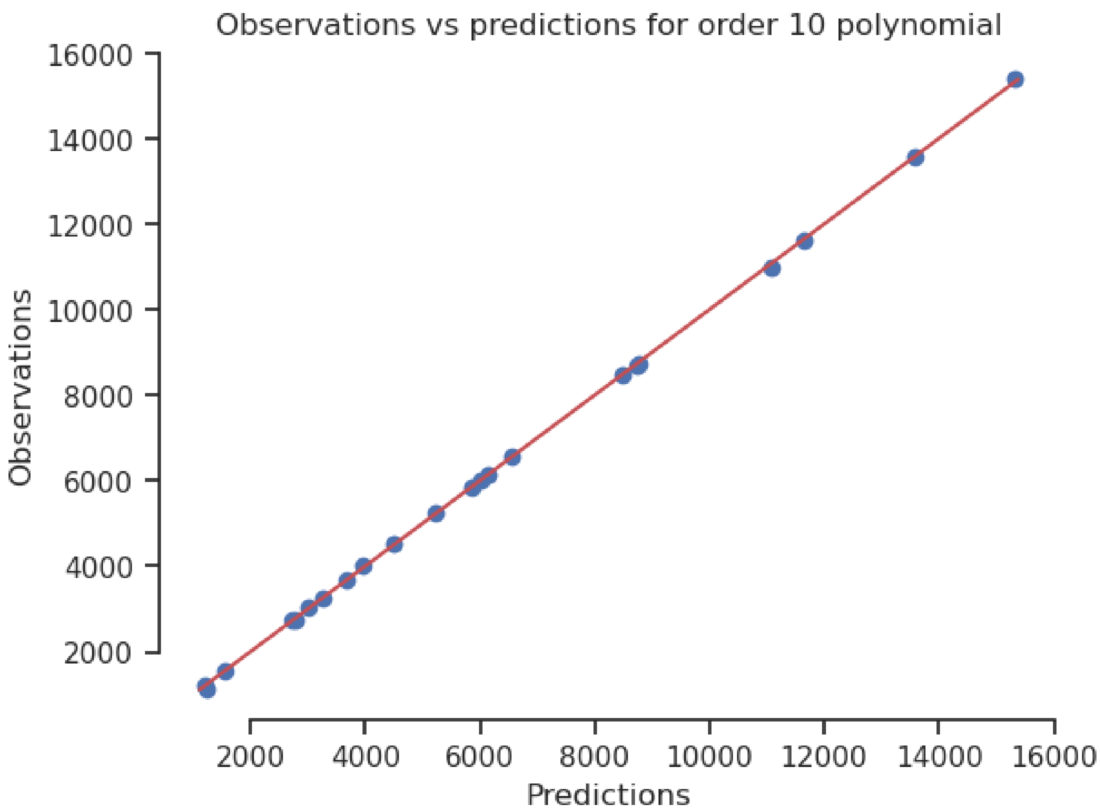
```
Sigma = 45.05145508359989
Hyperparameters = [4.098e-09 3.542e-06 2.830e-05 3.592e-02 8.454e-01 1.598e+01
7.647e+04
 1.247e+04 4.276e+02 1.465e+04 1.302e+04 1.119e+04 2.510e+04 3.954e+05
 4.856e+04 4.914e+05 1.443e+04 3.514e+04 4.931e+05 4.979e+05 4.864e+05
 1.642e+04 4.459e+05 4.975e+05 1.503e+05 4.605e+05 4.817e+05 1.229e+05
 6.114e+04 1.126e+05 3.283e+05 3.830e+04 3.955e+05 4.797e+05 4.723e+05
 2.681e+05 1.409e+05 2.427e+05 1.283e+04 2.967e+04 9.599e+04 4.136e+05
 2.922e+05 3.377e+05 5.479e+04 7.544e+04 1.765e+04 4.283e+05 1.403e+04
 2.706e+04 2.515e+05 3.710e+05 4.660e+05 4.743e+05 4.754e+05 3.191e+04
 4.464e+04 3.820e+05 1.746e+04 4.156e+04 1.669e+05 1.686e+04 6.377e+04
 4.744e+05 9.093e+04 4.760e+05]
Mean Squared Error: 1868.095105165281
```

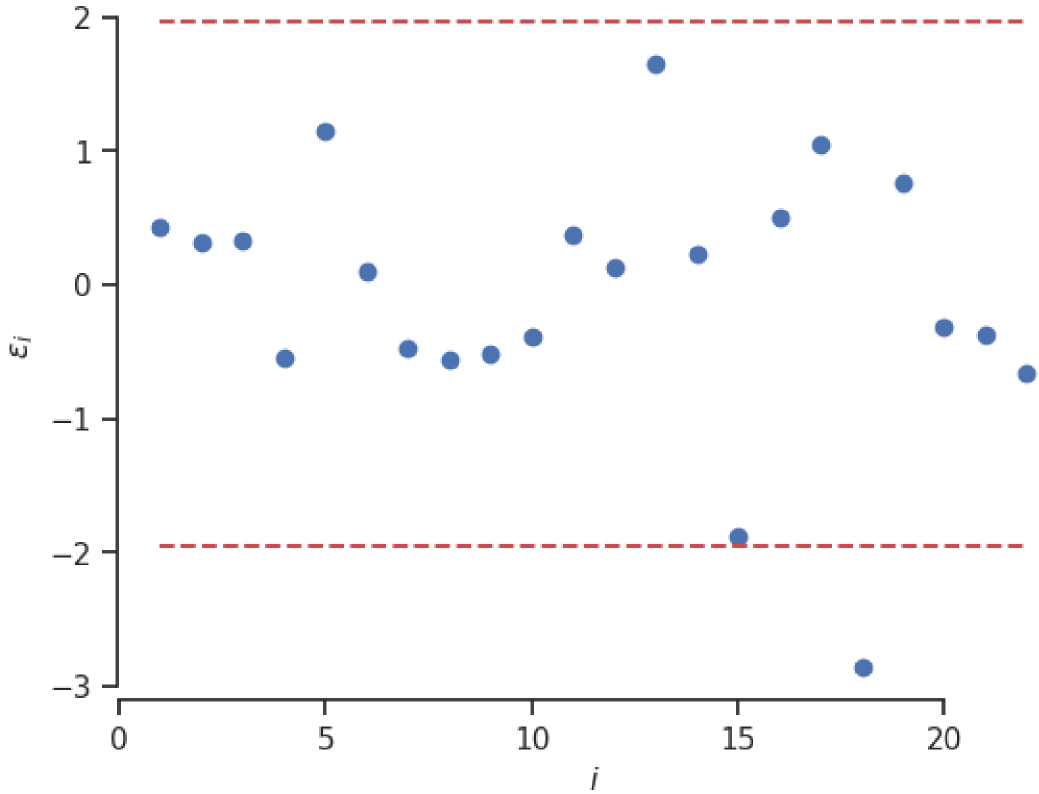

Training data
Validation data
Capacity
T_e
16000
14000
12000
10000
8000
6000
4000
2000
−30
−25
−20
−15
−10
−5
0
5
10



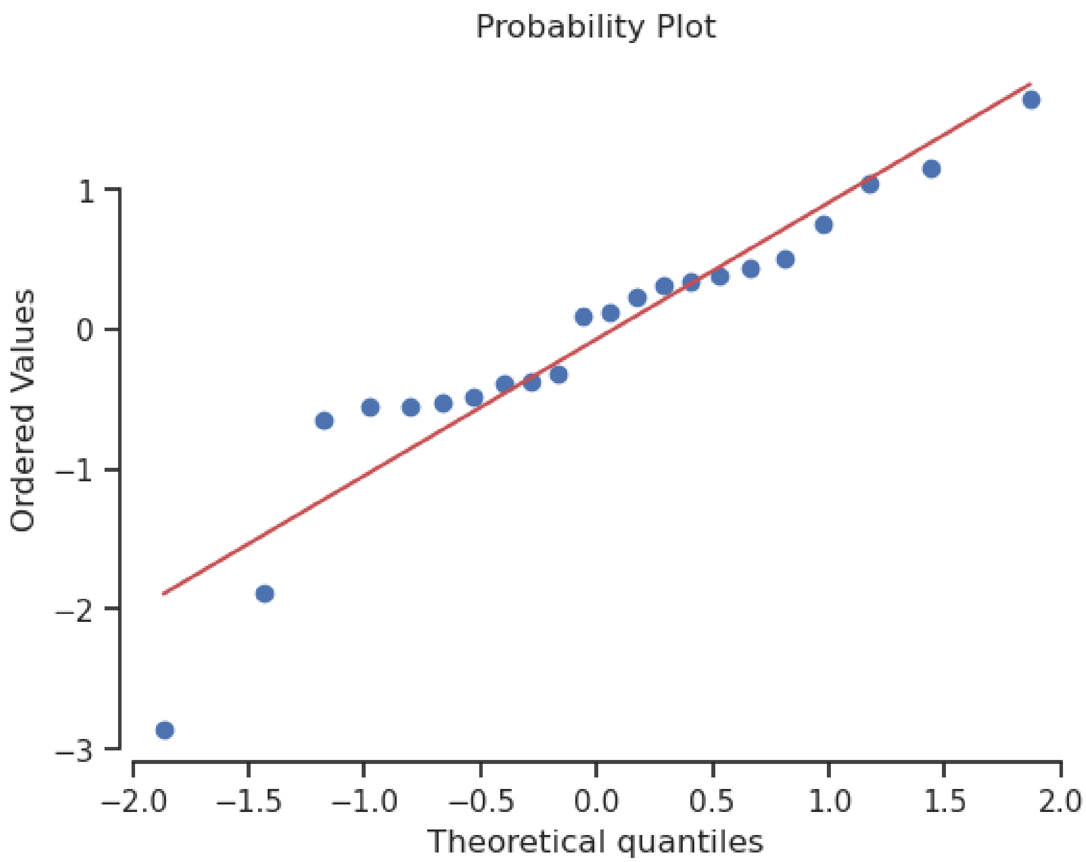
Training data
Validation data
Capacity
16000
14000
12000
10000
8000
6000
4000
2000
25
30
35
40
45
50
55
60
65
T_c


Training data
Training data
Validation data
Validation data
16000
14000
12000
10000
8000
6000
4000
2000
Capacity
−20
0
20
40
60
T_c and T_e


Observations vs predictions for order 10 polynomial
Observations
16000
14000
12000
10000
8000
6000
4000
2000
Predictions
2000
4000
6000
8000
10000
12000
14000
16000



2
1
0
−1
−2
−3
ε_i
0
5
10
15
20
i



3.0.2 Subpart A.II

What is the noise variance you estimated for the Capacity?

[ ]:
```
# your code here
print(f'Noise variance = {sigma}')
```

```
Noise variance = 39.280027521624866
```

3.0.3 Subpart A.III

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

[ ]:
```
# your code here
# No clue what we're supposed to do here
```

3.0.4 Subpart B.I - Fit the Power

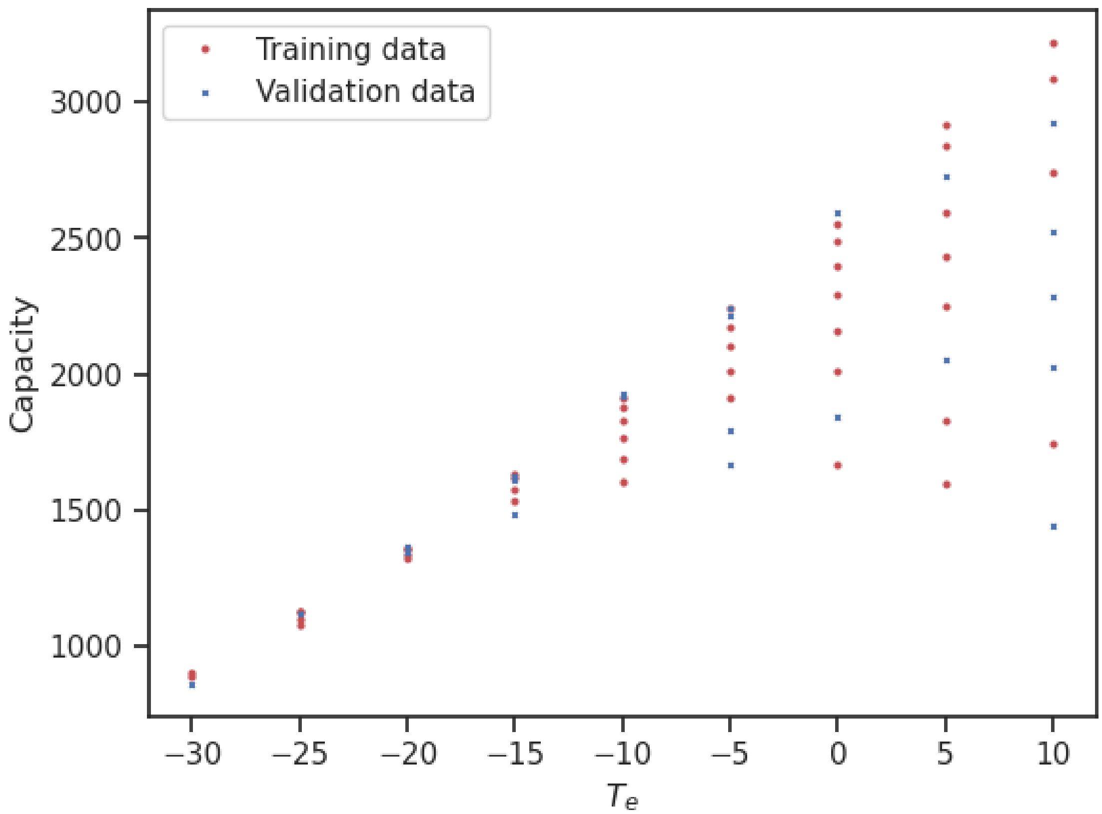
Please don't just fit blindly. Split in training and test and use all the usual diagnostics.

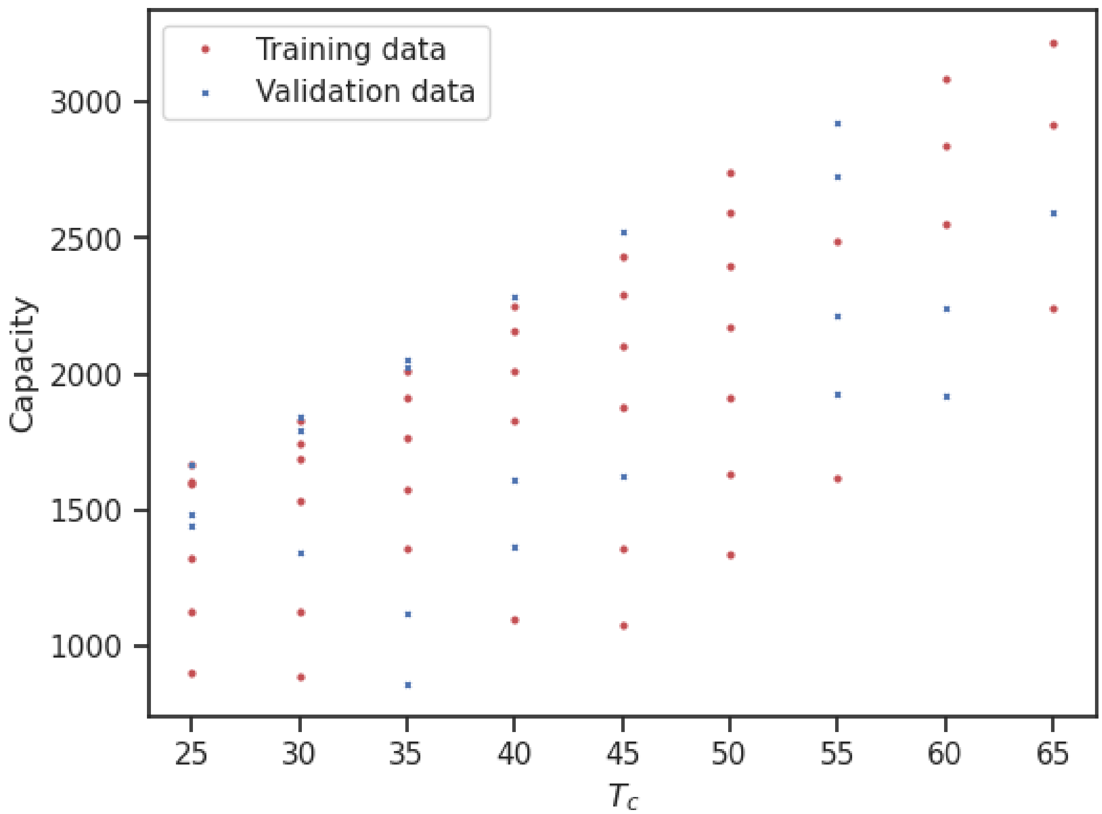
[ ]:
```python
# your code here - Repeat as many text and code blocks as you like
y = data['Power'].values
y

# your code here - Repeat as many text and code blocks as you like
x_train, x_valid, y_train, y_valid = skl.model_selection.
 ↪train_test_split(x,y,test_size=0.33)

#### Visaulise training data ####
plt.figure()
plt.plot(
    x_train[:,0],
    y_train,
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid[:,0],
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.ylabel('Power')
plt.xlabel('$T_e$')
plt.legend(loc='best');


plt.figure()
plt.plot(
    x_train[:,1],
    y_train,
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid[:,1],
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.ylabel('Power')
plt.xlabel('$T_c$')
plt.legend(loc='best');
```

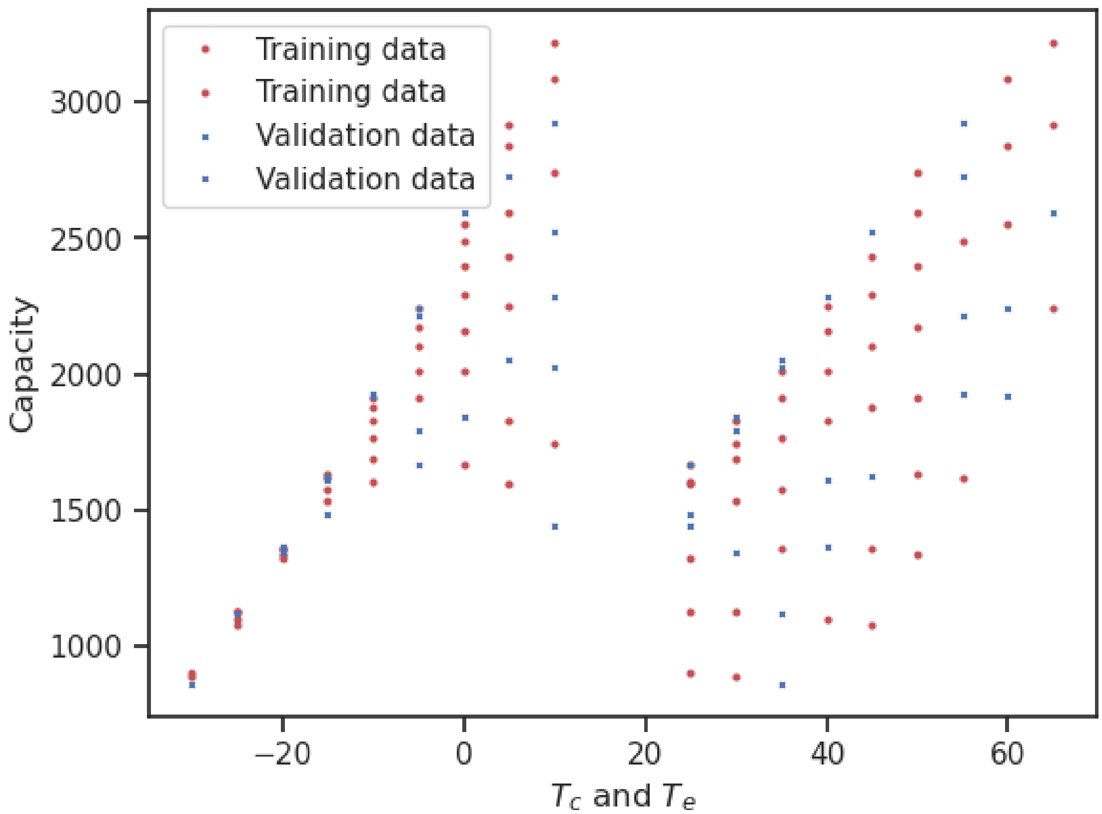
```python
plt.figure()
plt.plot(
    x_train,
    y_train,
    'ro',
    markersize=2,
    label='Training data'
)
plt.plot(
    x_valid,
    y_valid,
    'bx',
    markersize=2,
    label='Validation data'
)
plt.ylabel('Power')
plt.xlabel('$T_c$ and $T_e$')
plt.legend(loc='best');


#### Fitting Our Model and checking variance####
d = 10
import sklearn.preprocessing

polynomial = sklearn.preprocessing.PolynomialFeatures(d)
Phis = polynomial.fit_transform(x_train)

model = ARDRegression(fit_intercept = False).fit(Phis,y_train)

# Get Variance
sigma = np.sqrt(1.0 / model.alpha_)
lamb = model.lambda_
print(f'Sigma = {sigma}')
print(f'Hyperparameters = {lamb}')


#### Mean Squared Error ####
Phis_validation = polynomial.fit_transform(x_valid)
w = model.coef_

y_prediction = Phis_validation @ w
error= np.mean((y_prediction - y_valid)**2)
print(f'Mean Squared Error: {error}')
```

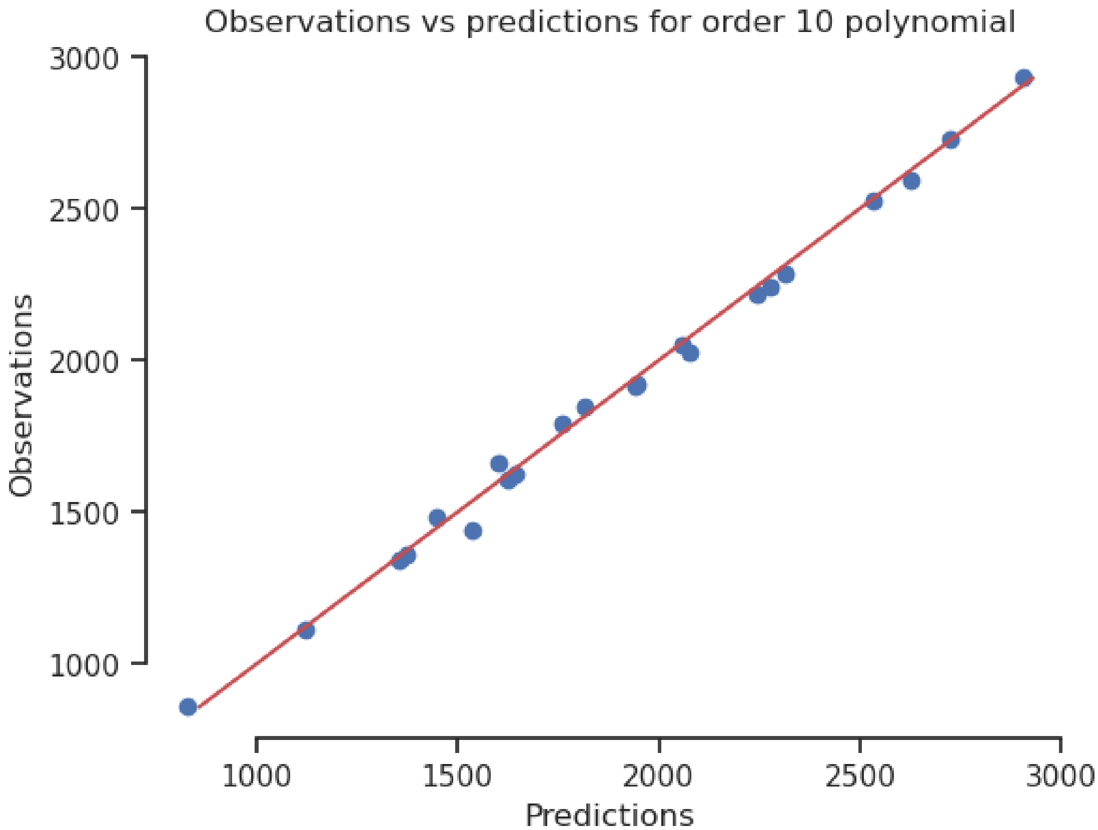
```python
##### Observations vs Predictions ####
plt.figure()
plt.plot(y_prediction,y_valid,'bo',)
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
plt.plot(yys, yys, 'r-')

plt.xlabel('Predictions')
plt.ylabel('Observations')
plt.title(f'Observations vs predictions for order {d} polynomial')
sns.despine(trim=True)


#### Standardized Errors ####
eps = (y_valid - y_prediction) / sigma

idx = np.arange(1, eps.shape[0] + 1)

fig, ax = plt.subplots()
ax.plot(idx, eps, 'o', label='Standarized errors')
ax.plot(idx, 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(idx, -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$')
sns.despine(trim=True);


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

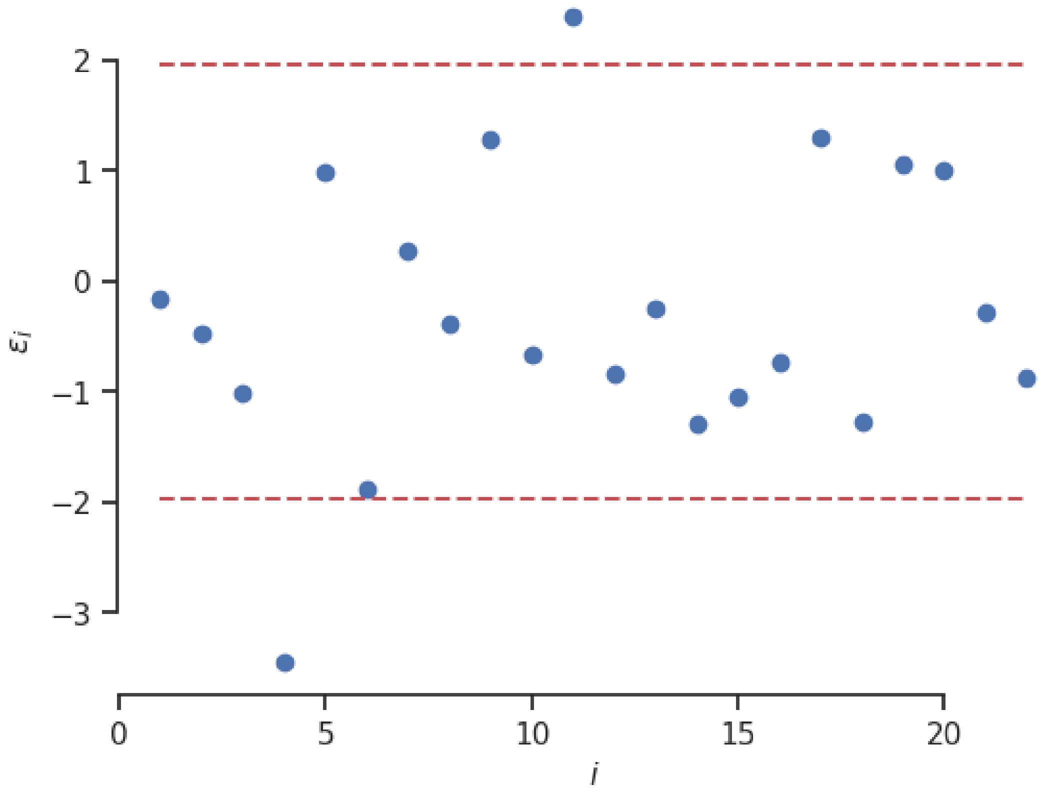
```
Sigma = 26.56830046488261
Hyperparameters = [1.109e-05 6.004e-04 2.473e-04 1.655e+00 3.853e-01 5.506e+00
9.351e+04
 4.105e+05 1.175e+04 3.805e+05 1.374e+04 1.022e+04 6.636e+04 1.286e+05
 4.113e+04 4.600e+04 4.980e+05 1.706e+04 1.385e+05 4.845e+05 4.839e+05
 3.526e+04 2.894e+04 3.589e+04 1.964e+05 4.267e+05 2.398e+05 4.596e+05
 1.629e+05 1.251e+05 1.388e+05 1.227e+05 1.411e+04 2.531e+05 2.590e+04
 4.463e+05 1.417e+04 1.295e+04 2.774e+04 7.674e+04 4.098e+05 4.472e+05
 1.242e+05 3.960e+05 2.585e+05 3.096e+05 3.846e+04 4.307e+04 2.345e+05
 4.318e+05 3.489e+05 1.481e+05 4.770e+05 4.718e+05 4.756e+05 1.332e+05
 3.399e+05 1.223e+04 3.857e+04 1.543e+05 4.722e+05 3.849e+05 4.682e+05
 2.097e+04 3.975e+05 4.797e+05]
Mean Squared Error: 1157.553416405095
```


Training data
Validation data
Capacity
3000
2500
2000
1500
1000
−30
−25
−20
−15
−10
−5
0
5
10
T_e

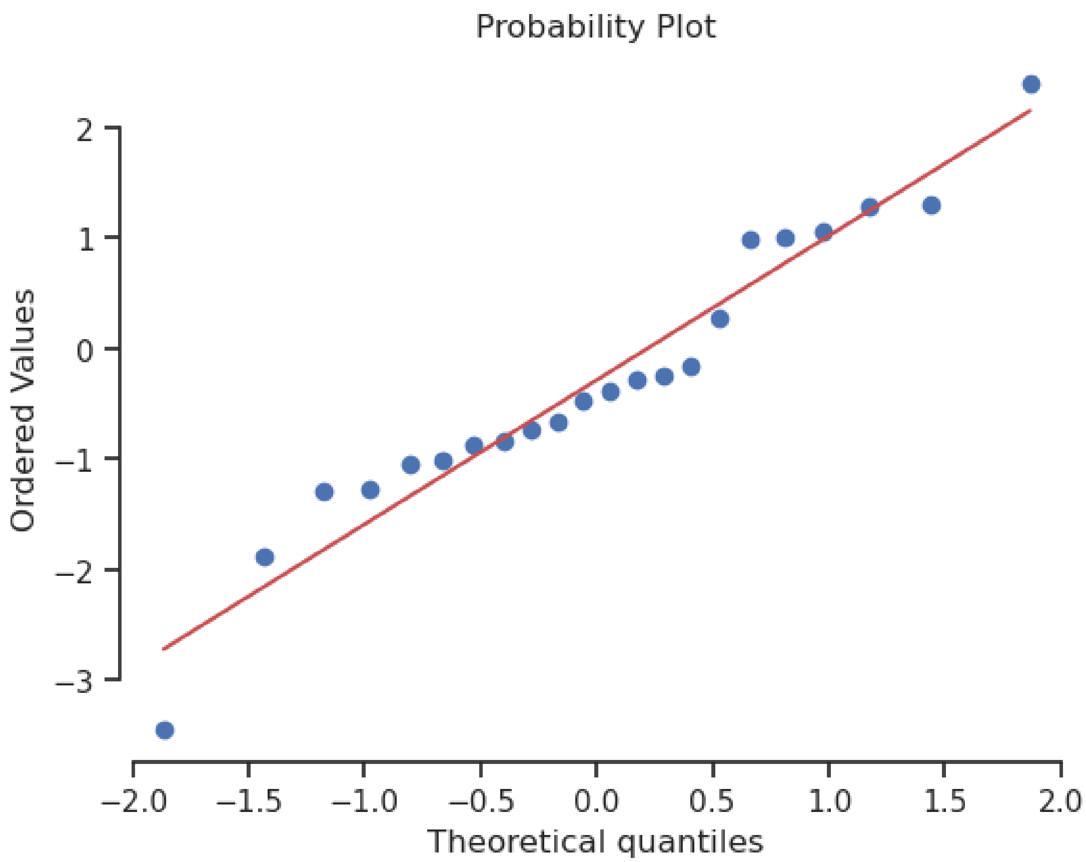

Training data
Validation data
Capacity
3000
2500
2000
1500
1000
25
30
35
40
45
50
55
60
65
T_c


Training data
Training data
Validation data
Validation data
Capacity
3000
2500
2000
1500
1000
−20
0
20
40
60
T_c and T_e


Observations vs predictions for order 10 polynomial
3000
2500
2000
1500
1000
Observations
1000
1500
2000
2500
3000
Predictions



2
1
0
−1
−2
−3
ε_i
0
5
10
15
20
i



3.0.5 Subpart B.II

What is the noise variance you estimated for the Power?

[ ]:
```
# your code here
print(f'Noise variance = {sigma}')
```

3.0.6 Subpart B.III

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

[ ]:
```
# your code here
# No idea what do do here
```

4 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 on 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 propelant through the nozzles thus producing thrust).

It turns out that the performance of the O-ring material was particularly sensitive on 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).

[ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
 ↪lecturebook/data/challenger_data.csv"
download(url)
```

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

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

[ ]:

	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` is has a binary coding (0=no damage, 1=damage). The very last row is the day of the Challenger accident.

We are going to 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 we proceed to the analysis of the data, let's clean the data up.

First, we drop all the bad records:

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

[ ]:

	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. Just remember that the temperature the day of the Challenger accident was 31 degrees F.

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

[ ]:

	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:

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

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

[ ]:
```
y = clean_data['Damage Incident'].values.astype(float)
y
```

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

4.1 Part A - Perform logistic regression

Perform logistic regression between the temperature (x) and the damage label (y). Do not bother doing a validation because there are not a lot of data. Just use a very simple model so that you don't overfit.

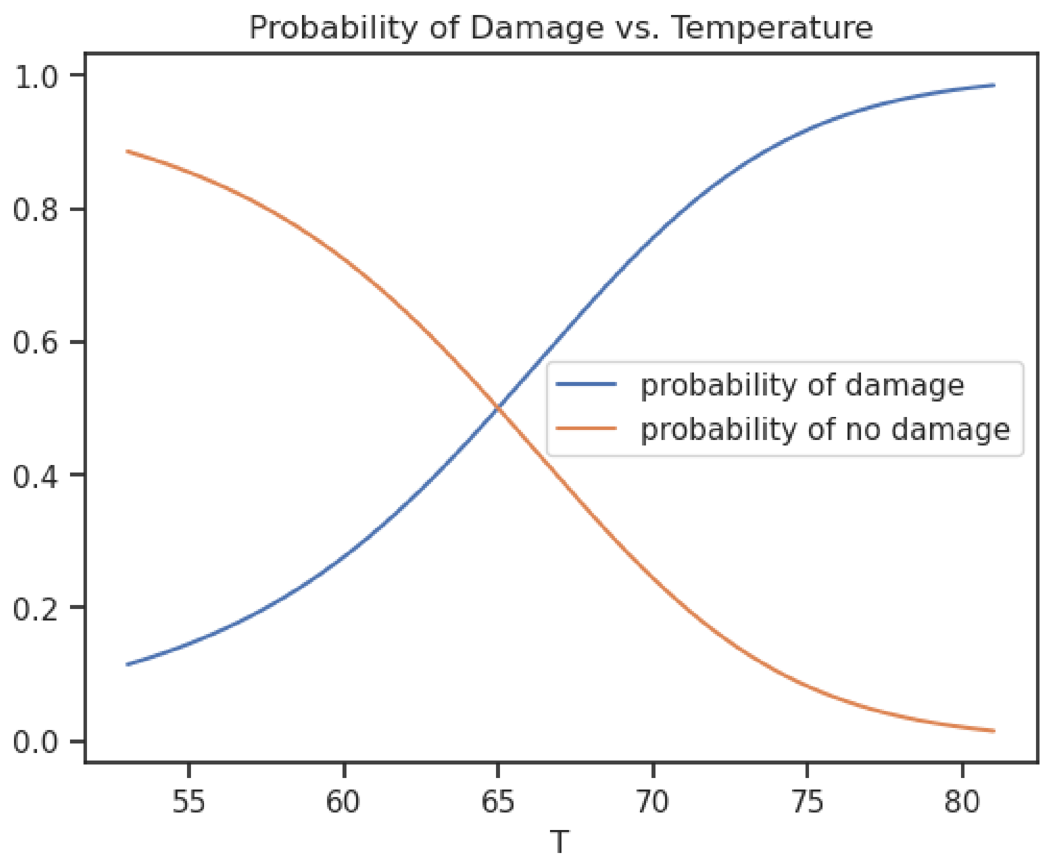
[ ]:
```
# your code here - Repeat as many text and code blocks as you like
#### Fitting Our Model and checking variance####
d = 2
import sklearn.preprocessing
import sklearn.linear_model

polynomial = sklearn.preprocessing.PolynomialFeatures(d)
Phis = polynomial.fit_transform(x[:,None])
```

```
model = sklearn.linear_model.LogisticRegression(fit_intercept = False, penalty␣
    ↪= None).fit(Phis,y)
```

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

Plot the probability of damage as a function of temperature.

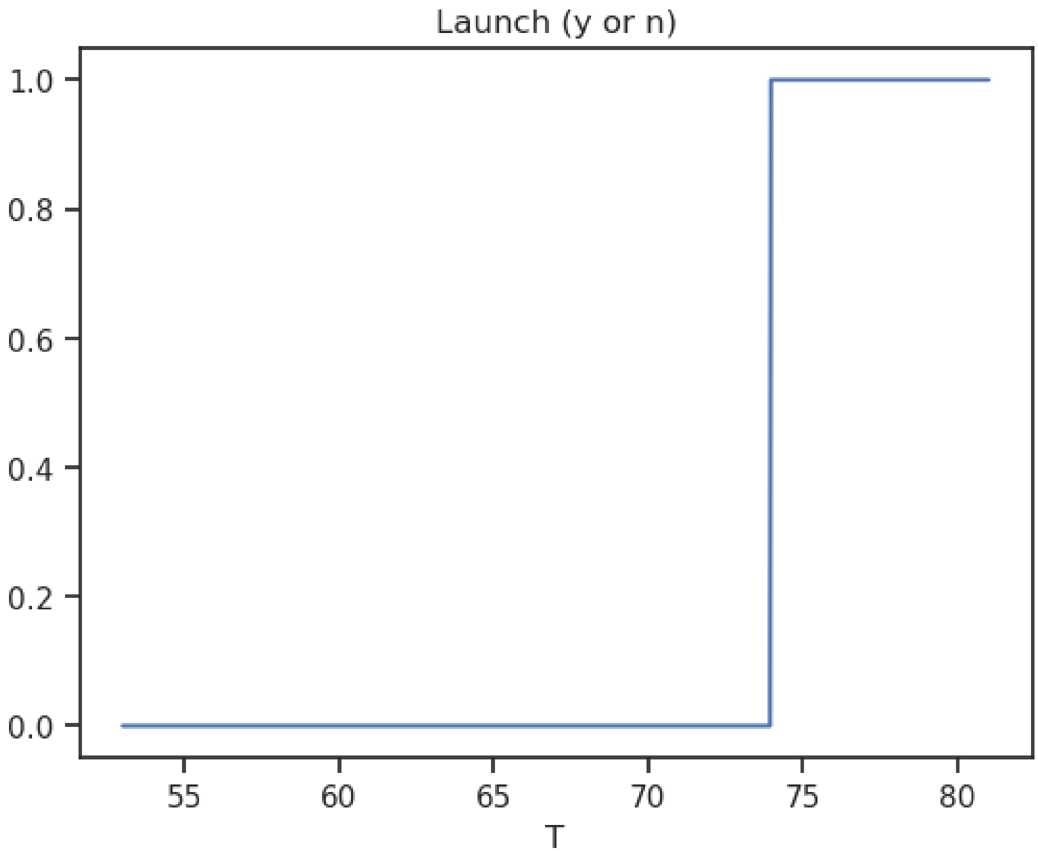
```
[ ]: # your code here
xx = np.linspace(np.min(x),np.max(x),1000)
Phi_xx = polynomial.fit_transform(xx[:,None])
pred = model.predict_proba(Phi_xx)

fig,ax = plt.subplots()
ax.plot(xx,pred[:,0],label='probability of damage')
ax.plot(xx,pred[:,1],label='probability of no damage')
ax.legend(loc='best')
ax.set_xlabel('T')
ax.set_title('Probability of Damage vs. Temperature');
```



4.3 Part C - Decide whether or not to launch

The temperature 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.

[ ]:
```
# your code here - Repeat as many text and code blocks as you like
scrubCost = 1.5*10**6 # Estimated cost of a scrub found online
shuttleCost = 444*10**6 # Estimated cost to manufacture a space shuttle
personCost = 7.5*10**6 #Estimate of human life cost according to FEMA in 2020
crew = 7

failureCost = shuttleCost + crew*personCost
launch = np.zeros(1000)

for i in range(1000):
    cost = failureCost * pred[i,0]
    if cost > shuttleCost:
        launch[i] = 1

fig,ax = plt.subplots()
ax.plot(xx,launch,label='launch?')
ax.set_xlabel('T')
ax.set_title('Launch (y or n)');
```


Launch (y or n)
1.0
0.8
0.6
0.4
0.2
0.0
55
60
65
70
75
80
T