chandlerrc_hw4

October 23, 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.

[60]:
```
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)


def plot_posterior_predictive(
    model,
    x_eval,
    x_obs,
    y_obs,
    phi_func,
    phi_func_args=(),
    y_true=None,
    xlabel="x",
    ylabel="y",
):
    """Plot the posterior predictive separating
    aleatory and espitemic uncertainty.

    Arguments:
    model    -- A trained model.
    x_eval   -- 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(
        x_eval[:, 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(x_eval, yy_mean, 'r', label="Posterior mean")
    ax.fill_between(
        x_eval,
        yy_le,
        yy_ue,
        color='red',
        alpha=0.25,
        label="95% epistemic credible interval"
    )
    ax.fill_between(
        x_eval,
        yy_lae,
        yy_le,
        color='green',
        alpha=0.25
    )
    ax.fill_between(
        x_eval,
        yy_ue,
        yy_uae,
        color='green',
        alpha=0.25,
        label="95% epistemic + aleatory credible interval"
    )
    ax.plot(x_obs, y_obs, 'kx', label='Observed data')
    if y_true is not None:
        ax.plot(x_eval, y_true, "--", label="True response")
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    plt.legend(loc="best", frameon=False)

def plot_posterior_samples(
    model,
    xx,
    x,
    y,
    phi_func,
    phi_func_args=(),
    num_samples=10,
    y_true=None,
    yy_true=None,
    nugget=1e-6,
    xlabel="x",
    ylabel="y",
):
```

```
    """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.reshape(-1, 1)
        ax.plot(xx, yy_sample, 'r', lw=0.5)
    ax.plot([], [], "r", lw=0.5, label="Posterior samples")
    ax.plot(x, y, 'kx', label='Observed data')
    if y_true is not None:
        ax.plot(xx, yy_true, label='True response surface')
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    plt.legend(loc="best", frameon=False)
```

1.3 Student details

- **First Name:** Robert
- **Last Name:** Chandler
- **Email:** chandl71@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 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:

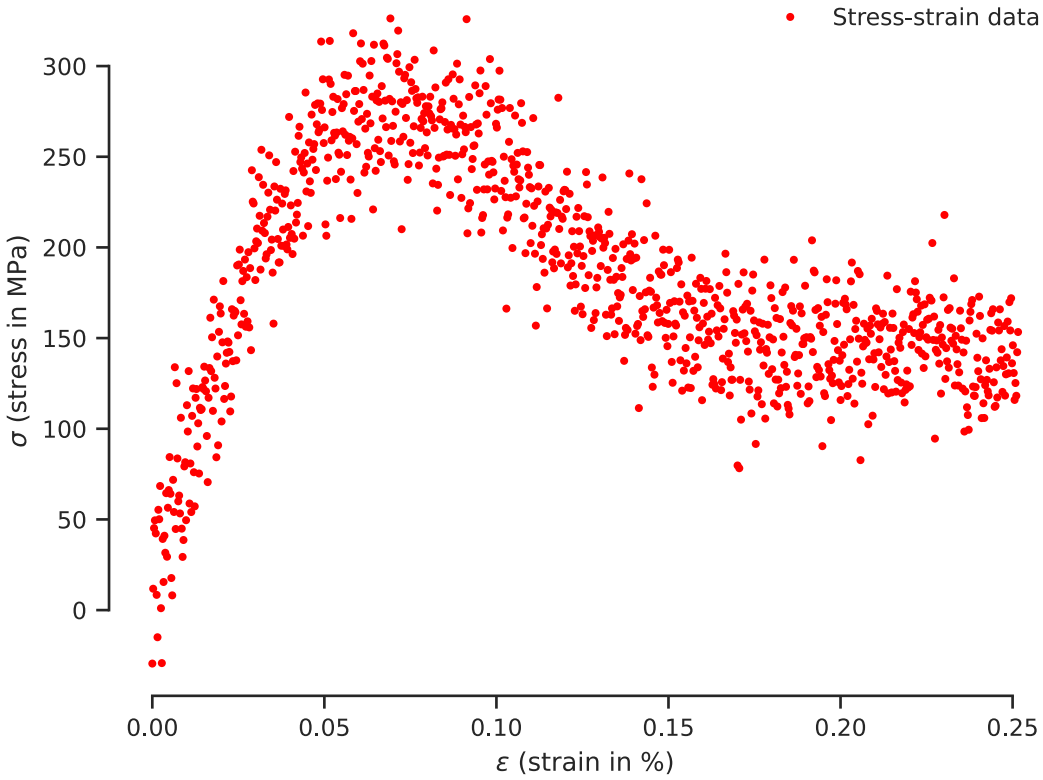
[61]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
 ↪lecturebook/data/stress_strain.txt"
download(url)
```

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

[62]:
```
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 by 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:

[63]:
```
# 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.

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:

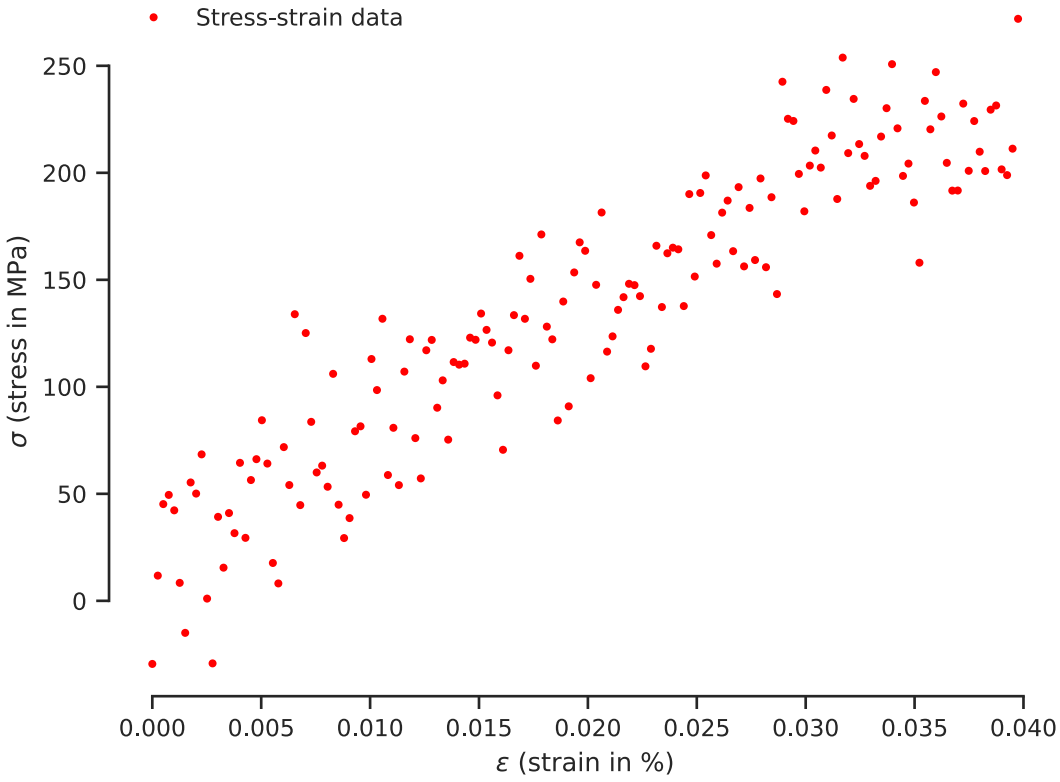
$$\sigma(\epsilon) = E\epsilon.$$

The constant E is known as the *Young modulus* of the material. Assume that you measure ϵ without 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.

[64]:
```
# 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", frameon=False)
sns.despine(trim=True)
```



Split your data into training and validation.

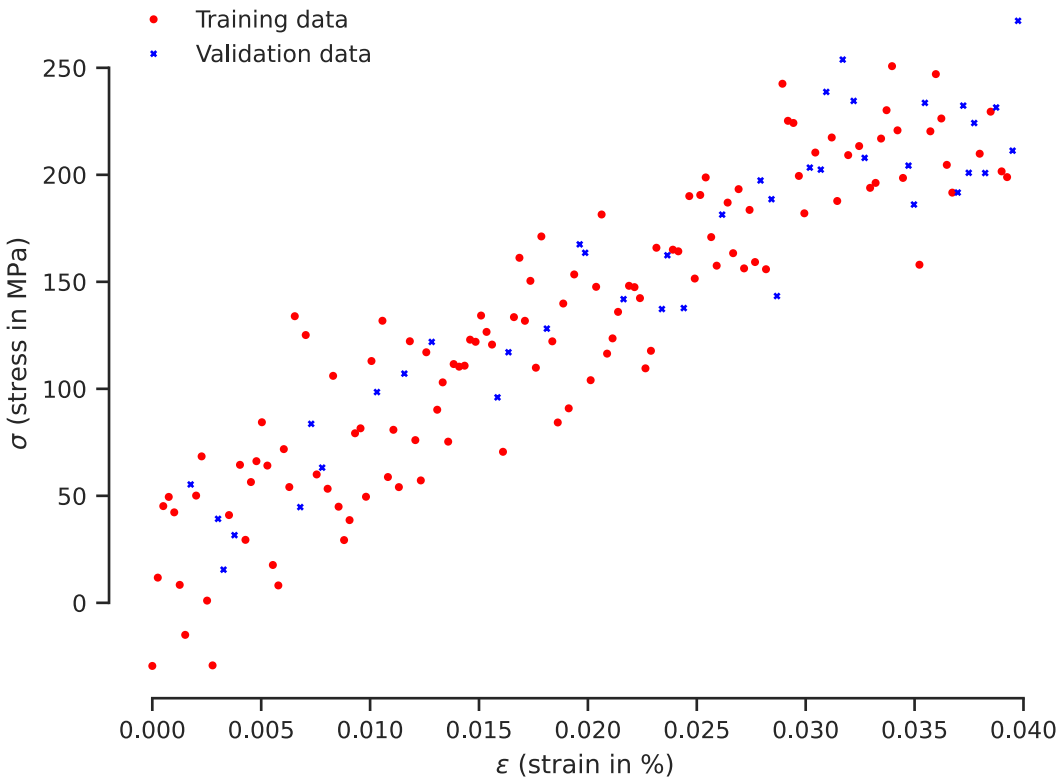
Hint: You may use sklearn.model_selection.train_test_split if you wish.

[65]:
```
# Split the data into training and validation datasets
from sklearn.model_selection import train_test_split
```

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

Use the following to visualize your split:

[66]:
```
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)
```



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.

We could use a 1st degree polynomial to fit the data to a line with an intercept, but including an intercept doesn't make as much sense for this case since we know that the true intercept is at (0, 0) since zero stress corresponds to zero strain. Including an intercept would give us lower errors, but this is really just overfitting the data. Therefore, we can eliminate the first column of ones from that matrix and simply use the x-values in a single column to fit to a line through the origin with

a slope value as our single weight.

[67]:
```
from sklearn.linear_model import BayesianRidge

Phi_train = x_train.reshape(-1, 1)
model = BayesianRidge(fit_intercept=False).fit(Phi_train, y_train)

print(f"Noise Variance (sigma) = {np.sqrt(1 / model.alpha_)}")
print(f"Precision of the weights (alpha) = {model.lambda_}")
print("For the Gamma-distributed prior, the following hyperparameters are:")
print(
    f"    alpha_1 (shape parameter over the alpha parameter) = {model.
 ↪get_params()['alpha_1']}"
)
print(
    f"    alpha_2 (rate parameter over the alpha parameter) = {model.
 ↪get_params()['alpha_2']}"
)
print(
    f"    lambda_1 (shape parameter over the lambda parameter) = {model.
 ↪get_params()['lambda_1']}"
)
print(
    f"    lambda_1 (rate parameter over the lambda parameter) = {model.
 ↪get_params()['lambda_2']}"
)
```

```
Noise Variance (sigma) = 30.348584839554704
Precision of the weights (alpha) = 2.3669673459305963e-08
For the Gamma-distributed prior, the following hyperparameters are:
    alpha_1 (shape parameter over the alpha parameter) = 1e-06
    alpha_2 (rate parameter over the alpha parameter) = 1e-06
    lambda_1 (shape parameter over the lambda parameter) = 1e-06
    lambda_1 (rate parameter over the lambda parameter) = 1e-06
```

2.1.3 Subpart A.III

Calculate the mean square error of the validation data.

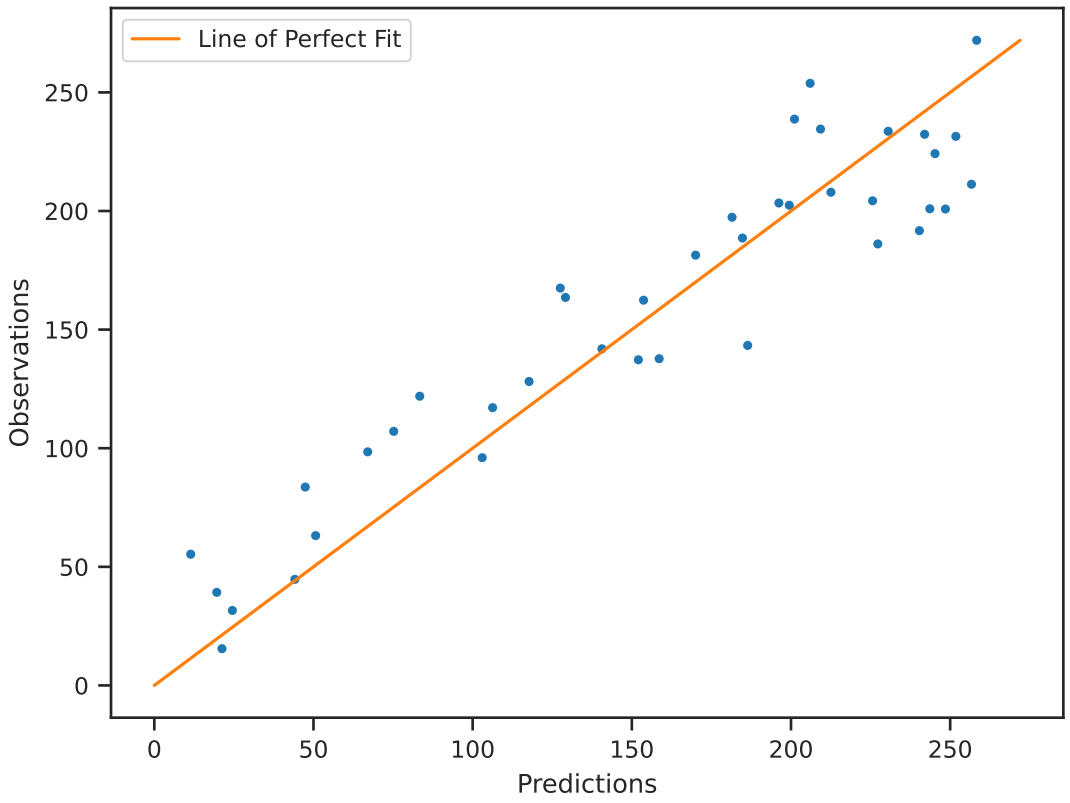
[68]:
```
from sklearn.metrics import mean_squared_error

Phi_valid = x_valid.reshape(-1, 1)
y_pred, y_pred_std = model.predict(Phi_valid, return_std=True)
print(
    f"The mean squared error of the validation data is␣
 ↪{mean_squared_error(y_valid, y_pred):0.4f} MPa^2"
)
```

```
The mean squared error of the validation data is 740.6339 MPa^2
```

2.1.4 Subpart A.IV

Make the observations vs predictions plot for the validation data.

[69]:
```
fig, ax = plt.subplots()
ax.plot(y_pred, y_valid, ".")
ax.plot([0, y_valid.max()], [0, y_valid.max()], label="Line of Perfect Fit")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations")

ax.legend();
```

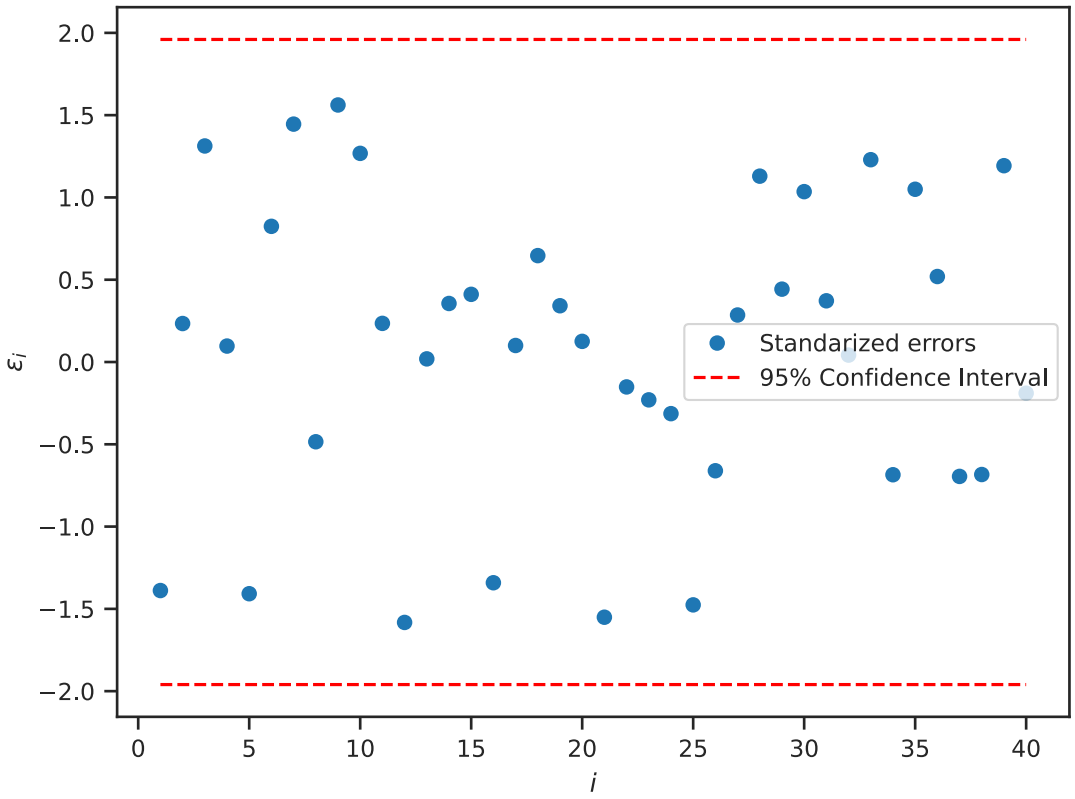


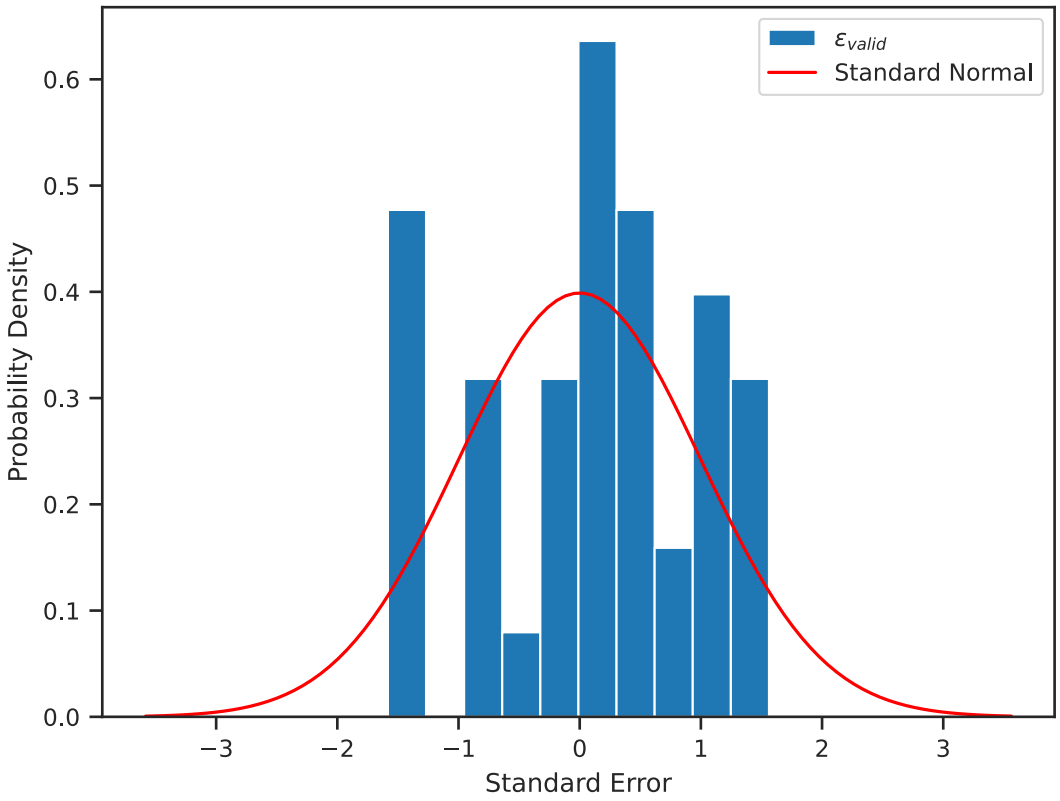
2.1.5 Subpart A.V

Compute and plot the standardized errors for the validation data.

[70]:
```
standardized_errors = (y_valid - y_pred) / y_pred_std

indices = np.arange(1, standardized_errors.size + 1)

fig, ax = plt.subplots()
ax.plot(indices, standardized_errors, "o", label="Standarized errors")
```

```
ax.plot(indices, 1.96 * np.ones(standardized_errors.shape[0]), "r--")
ax.plot(
    indices,
    -1.96 * np.ones(standardized_errors.shape[0]),
    "r--",
    label="95% Confidence Interval",
)
ax.set_xlabel("$i$")
ax.set_ylabel("$\epsilon_i$")
ax.legend()

fig, ax = plt.subplots()
ax.hist(standardized_errors, density=True, label=r"$\epsilon_{valid}$")
err_sample_points = np.linspace(
    standardized_errors.min() - 2, standardized_errors.max() + 2, 100
)
ax.plot(
    err_sample_points,
    st.distributions.norm.pdf(err_sample_points),
    "r",
    label="Standard Normal",
)
ax.set_xlabel("Standard Error")
ax.set_ylabel("Probability Density")
ax.legend();
```

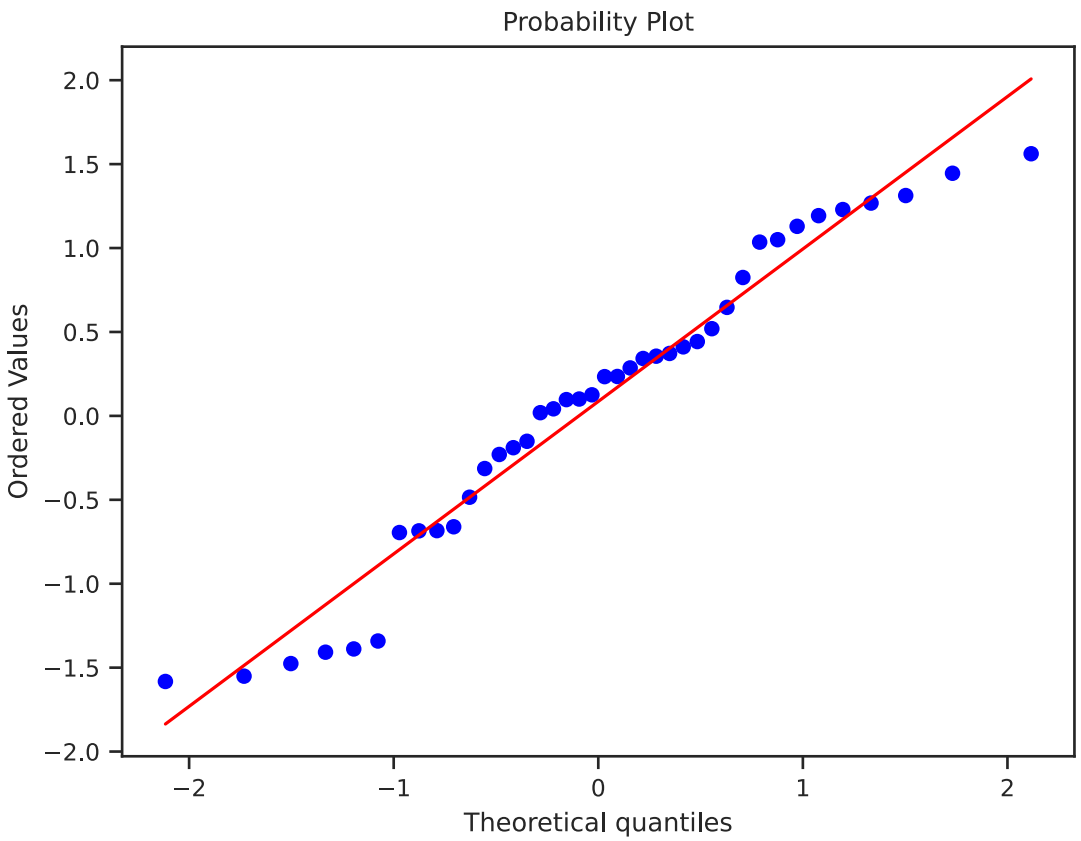

2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
ε_i
0
5
10
15
20
25
30
35
40
i
Standarized errors
95% Confidence Interval



2.1.6 Subpart A.VI

Make the quantile-quantile plot of the standardized errors.

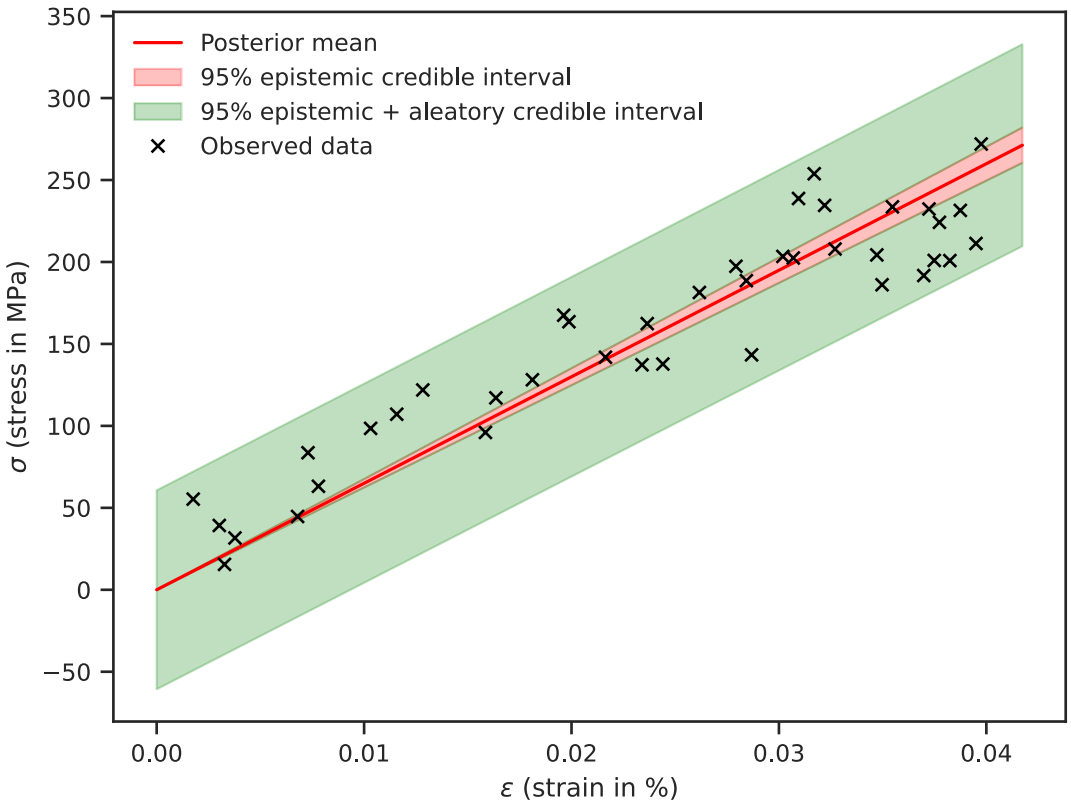
[71]:
```
fig, ax = plt.subplots()

st.probplot(standardized_errors, dist=st.norm, plot=ax);
```



2.1.7 Subpart A.VII

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

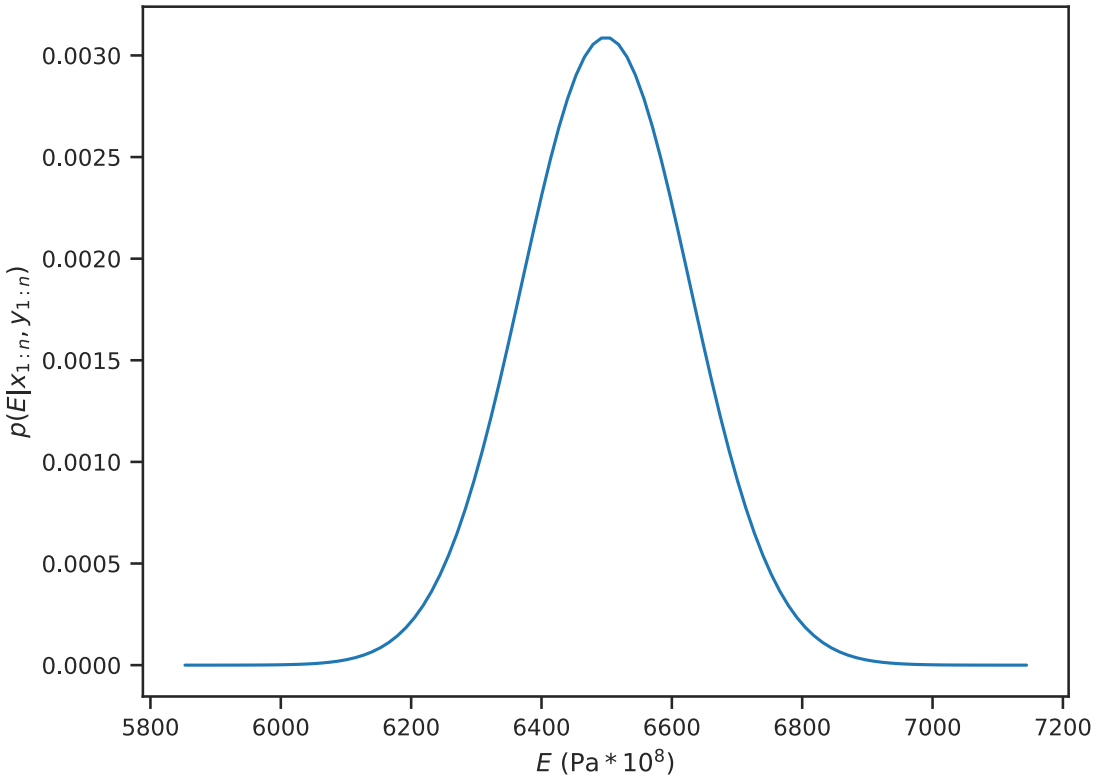
[72]:
```
plot_posterior_predictive(
    model,
    np.linspace(0, x_valid.max() * 1.05, 100),
    x_valid,
    y_valid,
    lambda x: x,
    xlabel="$\epsilon$ (strain in %)",
    ylabel="$\sigma$ (stress in MPa)",
)
```



2.1.8 Subpart A. VIII

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

[73]:
```
E_mean_post = model.coef_.item()
std_post = np.sqrt(model.sigma_.item())
x_sample_points = np.linspace(
    E_mean_post - 5 * std_post, E_mean_post + 5 * std_post, 100
)

fig, ax = plt.subplots()
ax.plot(x_sample_points, st.norm.pdf(x_sample_points, loc=E_mean_post,␣
 ↪scale=std_post))
ax.set_xlabel("$E\ (\mathsf{Pa}*10^8)$")
ax.set_ylabel("$p(E | x_{1:n}, y_{1:n})$");
```

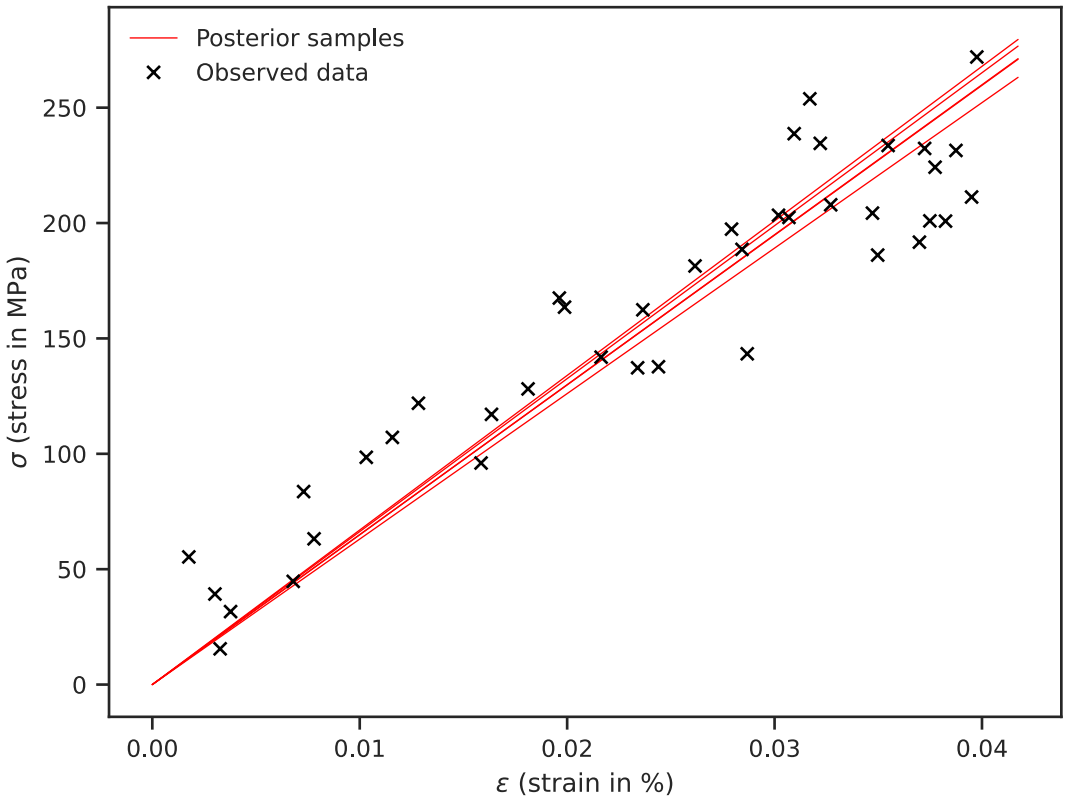


Note that the plots in the problem definition define the strain on the x-axis as a *percentage*. This was taken at face value, such that the y-units of MPa are divided by 0.01 (multiplied by 100) to derive the units of E, Young's modulus.

2.1.9 Subpart A.IX

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

[74]:
```
plot_posterior_samples(
    model,
    np.linspace(0, x_valid.max() * 1.05, 100),
    x_valid,
    y_valid,
    lambda x: x.reshape(-1, 1),
    num_samples=5,
    xlabel="$\epsilon$ (strain in %)",
    ylabel="$\sigma$ (stress in MPa)",
)
```

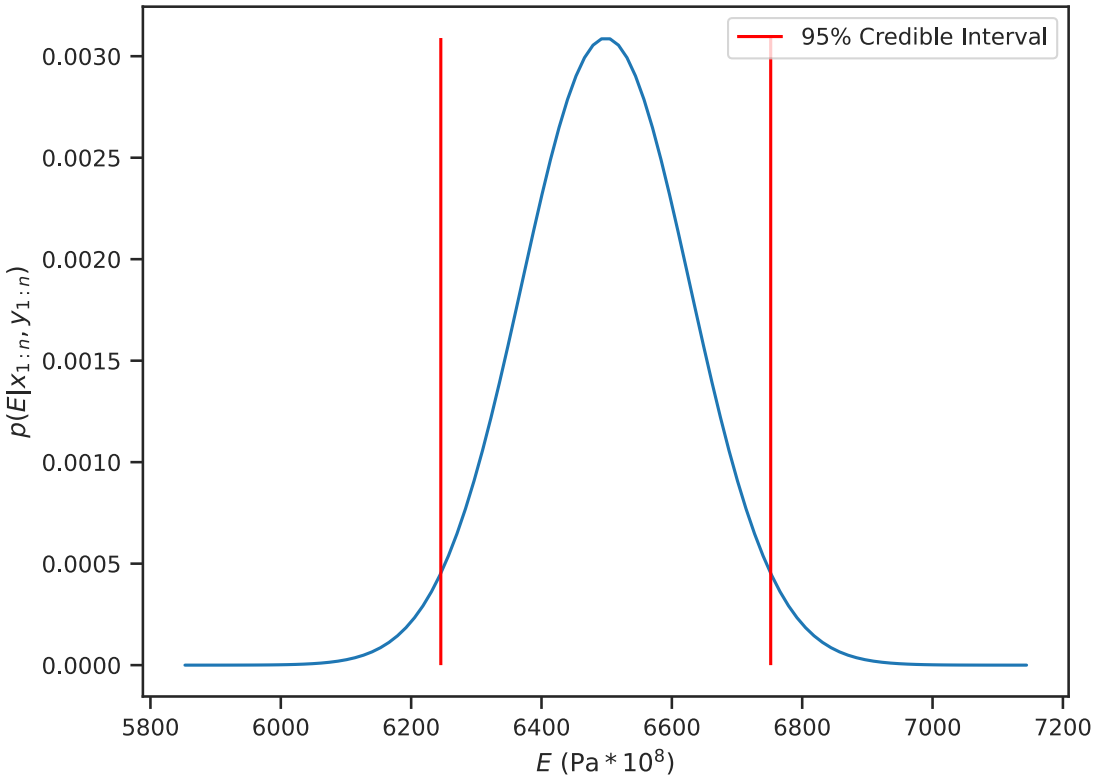


2.1.10 Subpart A.X

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

[75]:
```
lower_95 = st.norm.ppf(0.025, loc=E_mean_post, scale=std_post)
pdf_mean = st.norm.pdf(E_mean_post, loc=E_mean_post, scale=std_post)
upper_95 = st.norm.ppf(0.975, loc=E_mean_post, scale=std_post)

fig, ax = plt.subplots()
ax.plot(x_sample_points, st.norm.pdf(x_sample_points, loc=E_mean_post,␣
 ↪scale=std_post))


ax.vlines([lower_95, upper_95], 0, pdf_mean, "r", label="95% Credible Interval")
ax.set_xlabel("$E\ (\mathsf{Pa}*10^8)$")
ax.set_ylabel("$p(E | x_{1:n}, y_{1:n})$")
ax.legend()

print(f"The 95% interval for E is [{lower_95:.4f}, {upper_95:.4f}] Pa * 10^8\n")
```

```
The 95% interval for E is [6245.5036, 6751.6599] Pa * 10^8
```



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?

[76]:
```
E_est = E_mean_post
print(f"Choose E = {E_est:.4f} Pa*10^8")
```

```
Choose E = 6498.5818 Pa*10^8
```

While we do have uncertainty about the value of E, the best estimate that we have is the mean of the posterior of it conditioned on the data. This is where the probability is highest in the plots above since the posterior is normally distributed about it.

2.2 Part B - Estimate the ultimate strength

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

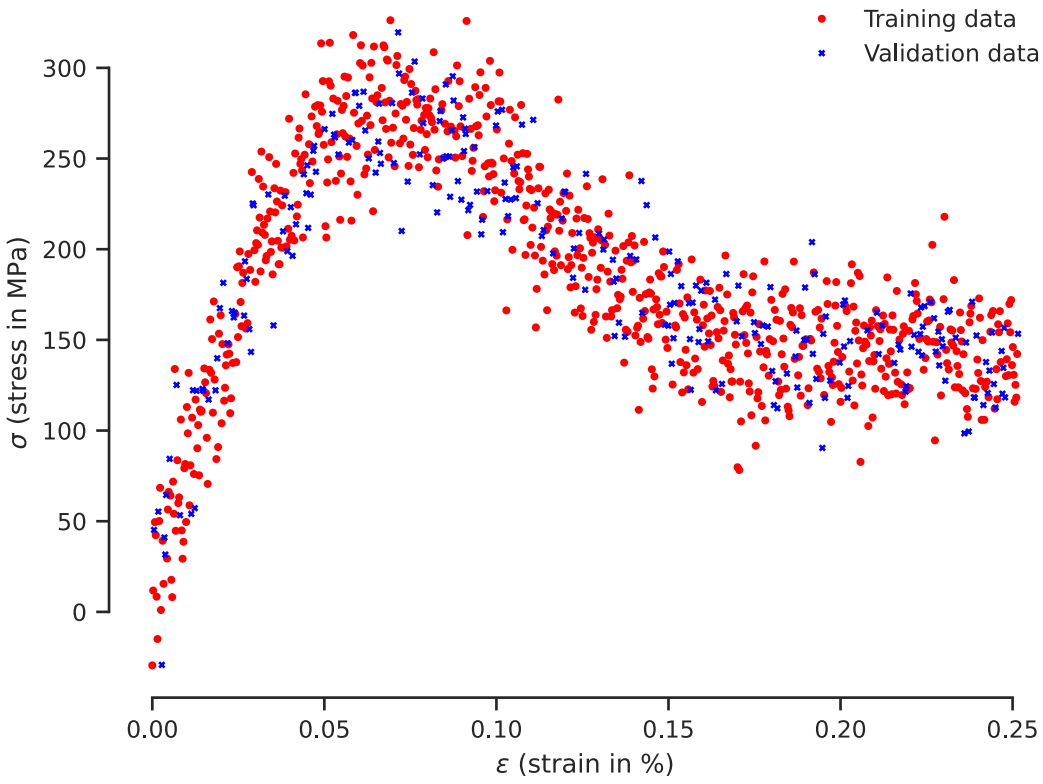
2.2.1 Subpart B.I - Extract training and validation data

Extract training and validation data from the entire dataset.

[77]:
```
x_train, x_valid, y_train, y_valid = train_test_split(x, y)
```

Use the following to visualize your split:

[78]:
```
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)
```



2.2.2 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 long 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:

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

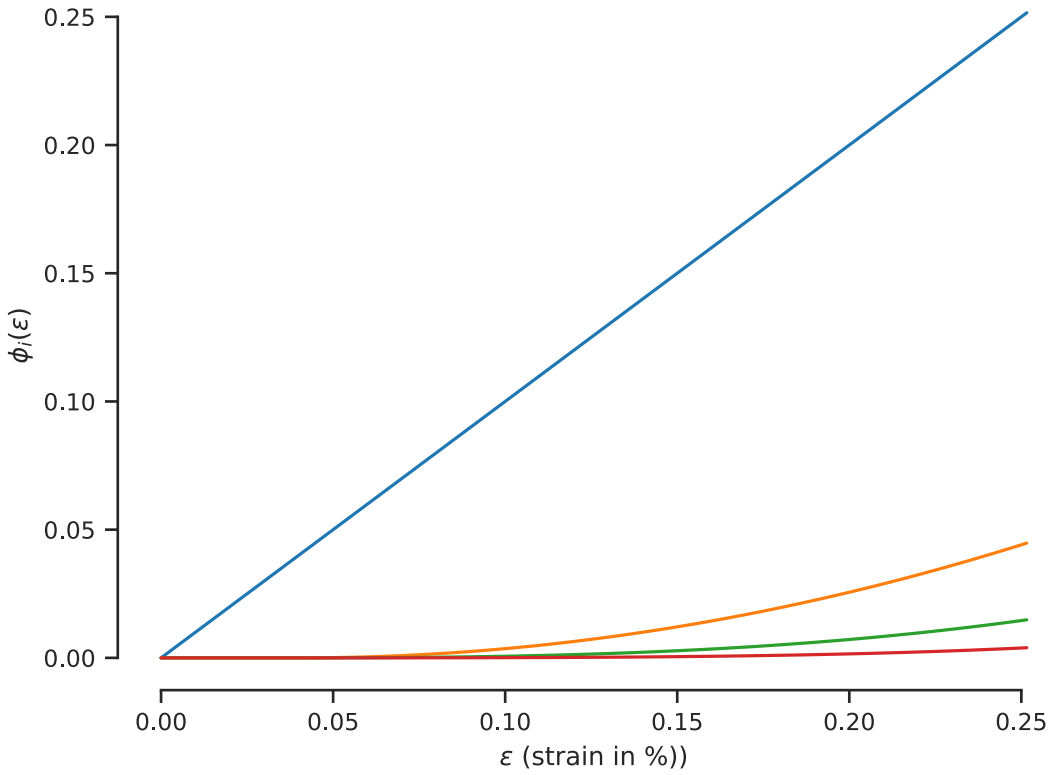
Okay. This is still a generalized linear model. This is nice. Write code for the design matrix:

[79]:
```
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)
    # Ensure that N has only one dimension with length greater than 1
    assert np.sum(np.array(Epsilon.shape) > 1) == 1
    Epsilon = Epsilon.flatten()
    n = Epsilon.shape[0]
    # The design matrix:
    Phi = np.zeros((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] = (
            (i - 1) * epsilon_l**i - i * Epsilon * epsilon_l ** (i - 1) +␣
↪Epsilon**i
        ) * Step
    return Phi
```

Visualize the basis functions here:

[80]:
```
d = 4
standardized_errors = np.linspace(0, x.max(), 100)
Phis = compute_design_matrix(standardized_errors, epsilon_l, d)
fig, ax = plt.subplots(dpi=100)
ax.plot(standardized_errors, Phis)
```

```
ax.set_xlabel("$\epsilon$ (strain in %))")
ax.set_ylabel("$\phi_i(\epsilon)$")
sns.despine(trim=True)
```



2.2.3 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.).

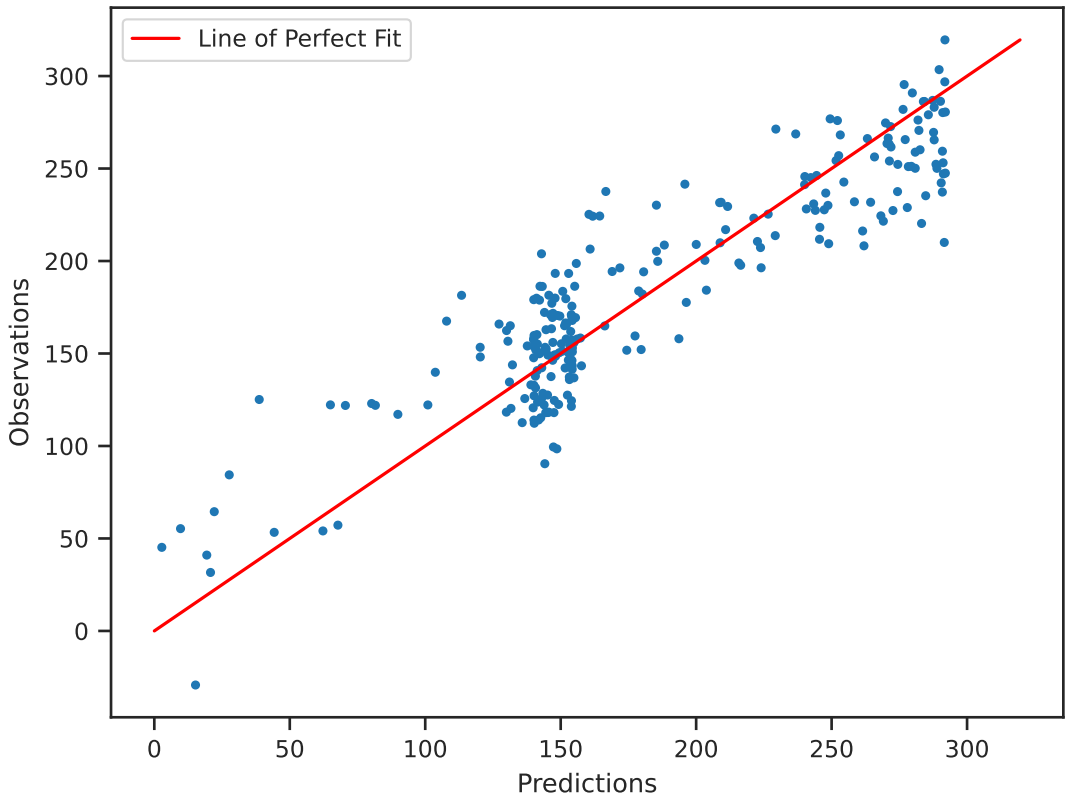
[81]:
```
from sklearn.linear_model import ARDRegression

model = ARDRegression(fit_intercept=False).fit(
    compute_design_matrix(x_train, epsilon_1, d), y_train
)

y_pred, y_pred_std = model.predict(
    compute_design_matrix(x_valid, epsilon_1, d), return_std=True
)
```

Compute MSE

[82]:
```
Phi_valid = compute_design_matrix(x_valid, epsilon_l, d)
y_pred, y_pred_std = model.predict(Phi_valid, return_std=True)
print(
    f"The mean squared error of the validation data is␣
 ↪{mean_squared_error(y_valid, y_pred):0.4f} MPa^2"
)
```

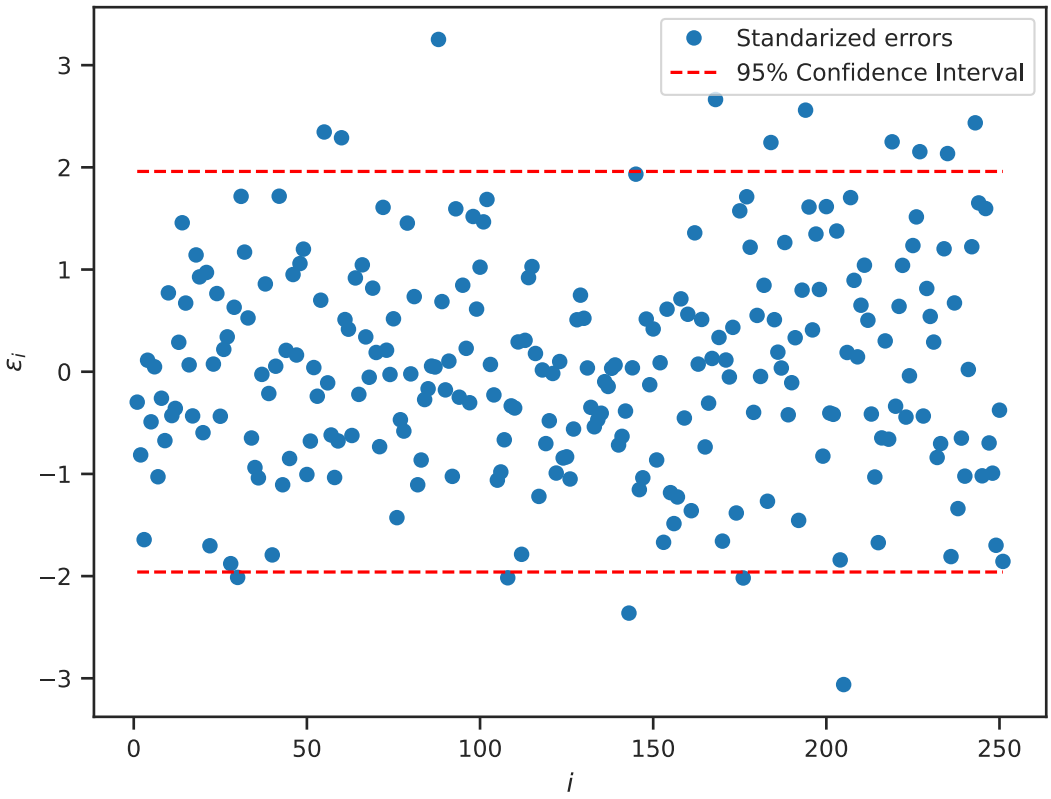
```
The mean squared error of the validation data is 773.9880 MPa^2
```

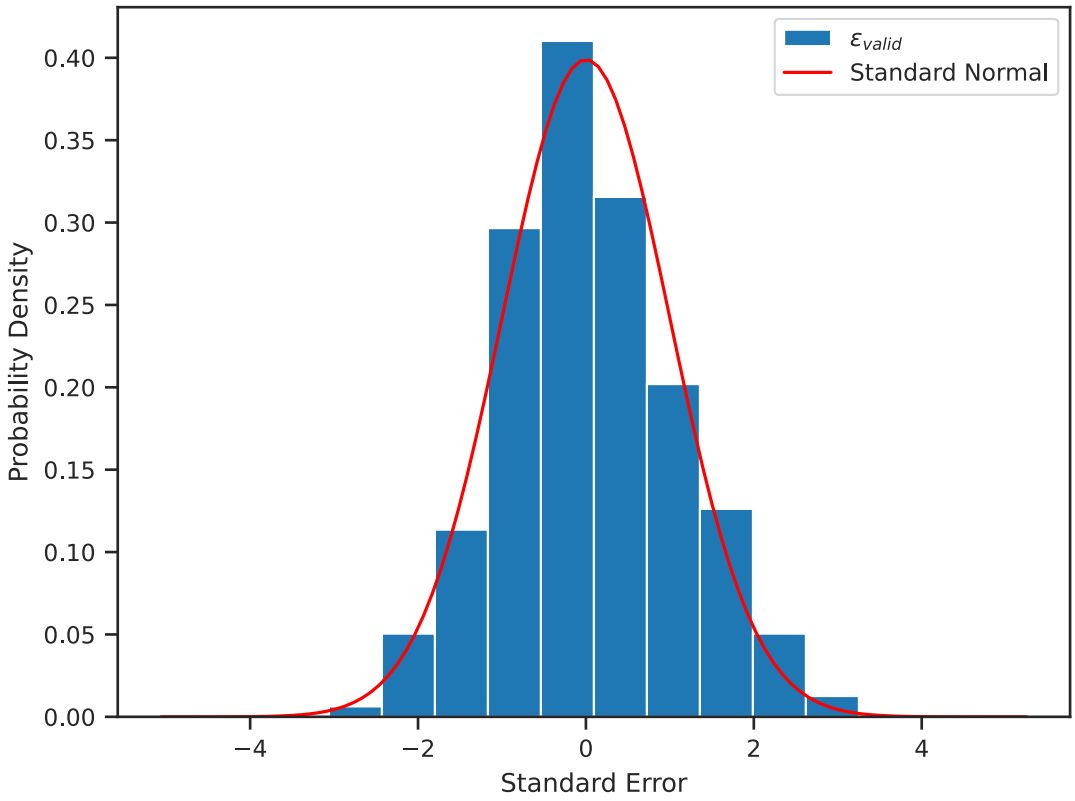
[83]:
```
fig, ax = plt.subplots()
ax.plot(y_pred, y_valid, ".")
ax.plot([0, y_valid.max()], [0, y_valid.max()], "r", label="Line of Perfect␣
 ↪Fit")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations")

ax.legend();
```



Standardized Errors

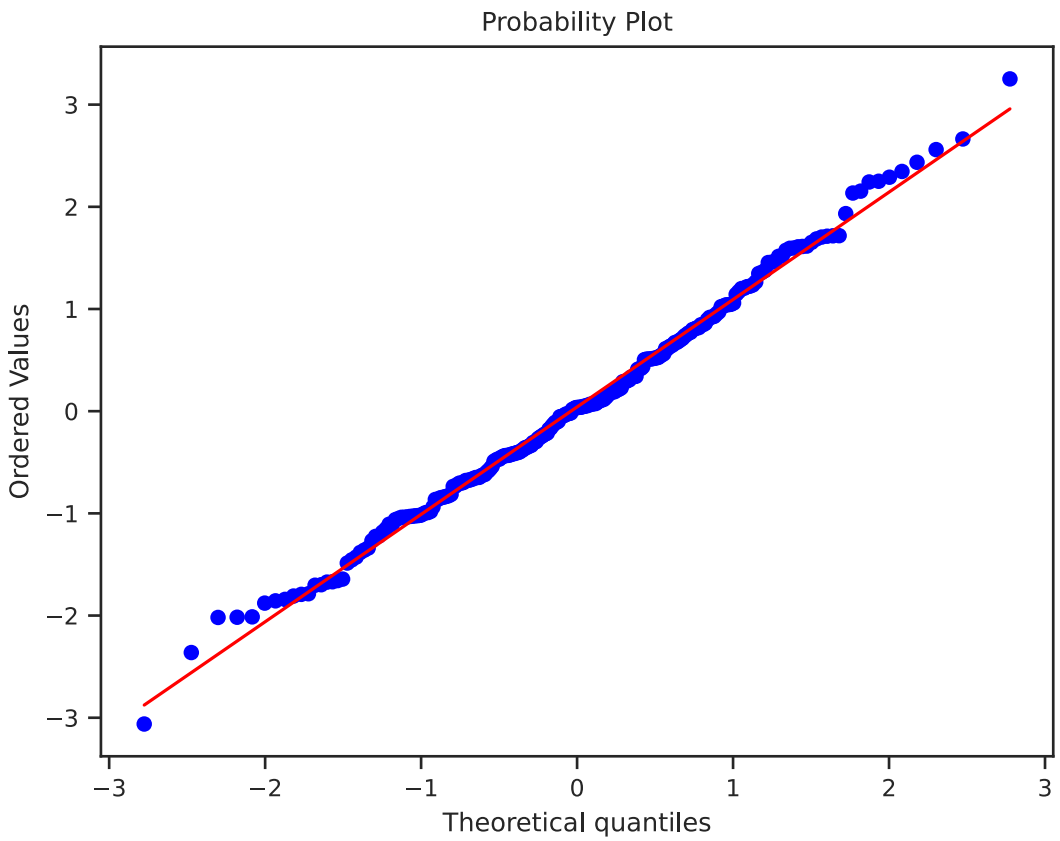
[84]:
```
standardized_errors = (y_valid - y_pred) / y_pred_std

indices = np.arange(1, standardized_errors.size + 1)

fig, ax = plt.subplots()
ax.plot(indices, standardized_errors, "o", label="Standarized errors")
ax.plot(indices, 1.96 * np.ones(standardized_errors.shape[0]), "r--")
ax.plot(
    indices,
    -1.96 * np.ones(standardized_errors.shape[0]),
    "r--",
    label="95% Confidence Interval",
)
ax.set_xlabel("$i$")
ax.set_ylabel("$\epsilon_i$")
ax.legend()

fig, ax = plt.subplots()
ax.hist(standardized_errors, density=True, label=r"$\epsilon_{valid}$")
err_sample_points = np.linspace(
    standardized_errors.min() - 2, standardized_errors.max() + 2, 100
)
ax.plot(
    err_sample_points,
    st.distributions.norm.pdf(err_sample_points),
    "r",
    label="Standard Normal",
)
ax.set_xlabel("Standard Error")
ax.set_ylabel("Probability Density")
ax.legend();
```


Standarized errors
95% Confidence Interval
3
2
1
0
−1
−2
−3
ε_i
0
50
100
150
200
250
i



q-q Plot

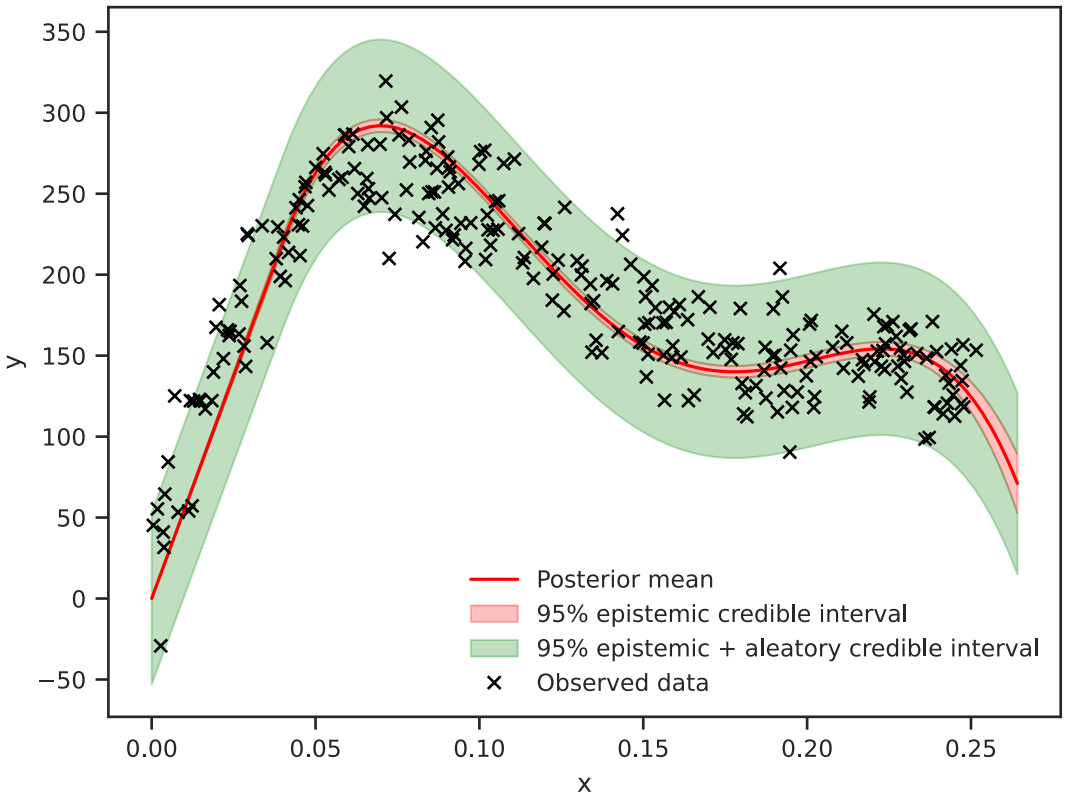
[85]:
```
fig, ax = plt.subplots()
st.probplot(standardized_errors, dist=st.norm, plot=ax);
```



2.2.4 Subpart B.IV

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

[86]:
```
plot_posterior_predictive(
    model,
    np.linspace(0, x_valid.max() * 1.05, 100),
    x_valid,
    y_valid,
    compute_design_matrix,
    phi_func_args=(epsilon_1, d),
)
```



2.2.5 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.

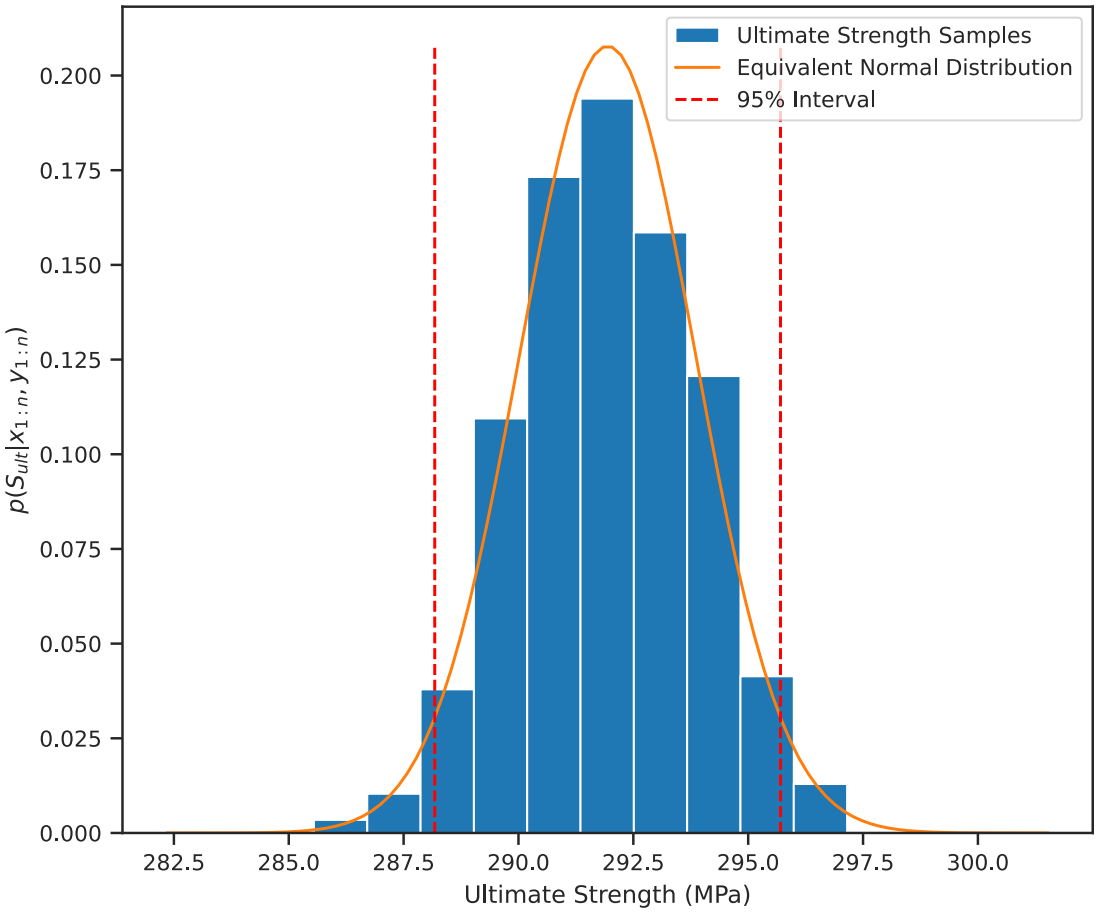
[87]:
```
num_samples = 1000
points_per_sample = 1000

strain_sample_points = np.linspace(0, 0.25, points_per_sample)
S_ultimate = np.zeros(num_samples)
Phi_sample_points = compute_design_matrix(
```

```
    strain_sample_points, epsilon_l=epsilon_l, d=d
)

nugget = 1e-6
m = model.coef_
S = model.sigma_
w_post = st.multivariate_normal(mean=m, cov=S + nugget * np.eye(S.shape[0]))

for i in range(num_samples):
    w_sampled = w_post.rvs()
    stress_sample_points = Phi_sample_points @ w_sampled
    S_ultimate[i] = stress_sample_points.max()
```

[88]:
```
mean_S_ult = S_ultimate.mean()
std_S_ult = S_ultimate.std()
norm_S_ult = st.norm(loc=mean_S_ult, scale=std_S_ult)

lower_95 = norm_S_ult.ppf(0.025)
upper_95 = norm_S_ult.ppf(0.975)
pdf_mean = norm_S_ult.pdf(mean_S_ult)

fig, ax = plt.subplots(figsize=(7, 6))
ax.hist(S_ultimate, density=True, label="Ultimate Strength Samples")
norm_sample_x = np.linspace(mean_S_ult - 5 * std_S_ult, mean_S_ult + 5 *␣
 ↪std_S_ult, 100)
ax.plot(
    norm_sample_x, norm_S_ult.pdf(norm_sample_x), label="Equivalent Normal␣
 ↪Distribution"
)
ax.vlines([lower_95, upper_95], 0, pdf_mean, "r", linestyles="--", label="95%␣
 ↪Interval")
ax.set_xlabel("Ultimate Strength (MPa)")
ax.set_ylabel("$p(S_{ult}|x_{1:n},y_{1:n})$")
ax.legend()

print(
    f"The 95% Credible Interval for the Ultimate Strength is [{lower_95:.4f},␣
 ↪{upper_95:.4f}] MPa"
)
print(
    f"\nThe selected value for the ultimate strength is the mean of the sampled␣
 ↪values, {mean_S_ult:.4f} MPa"
)
```

```
The 95% Credible Interval for the Ultimate Strength is [288.1773, 295.7032] MPa

The selected value for the ultimate strength is the mean of the sampled values,
```

```
291.9402 MPa
```



3 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:

[89]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
 ↪lecturebook/data/compressor_data.xlsx"
download(url)
```

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

[90]:
```
import pandas as pd

data = pd.read_excel("compressor_data.xlsx")
data
```

[90]:
	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 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.

[91]:
```
# 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
```

[91]:
```
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]])
```

[92]:
```
# Here is how to extract the Capacity
y = data["Capacity"].values
y
```

[92]:
```
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 ensure your models make sense. + Use ARD Regression to fit any hyperparameters and the noise.

3.1 Part A - Fit the capacity

3.1.1 Subpart A.I

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

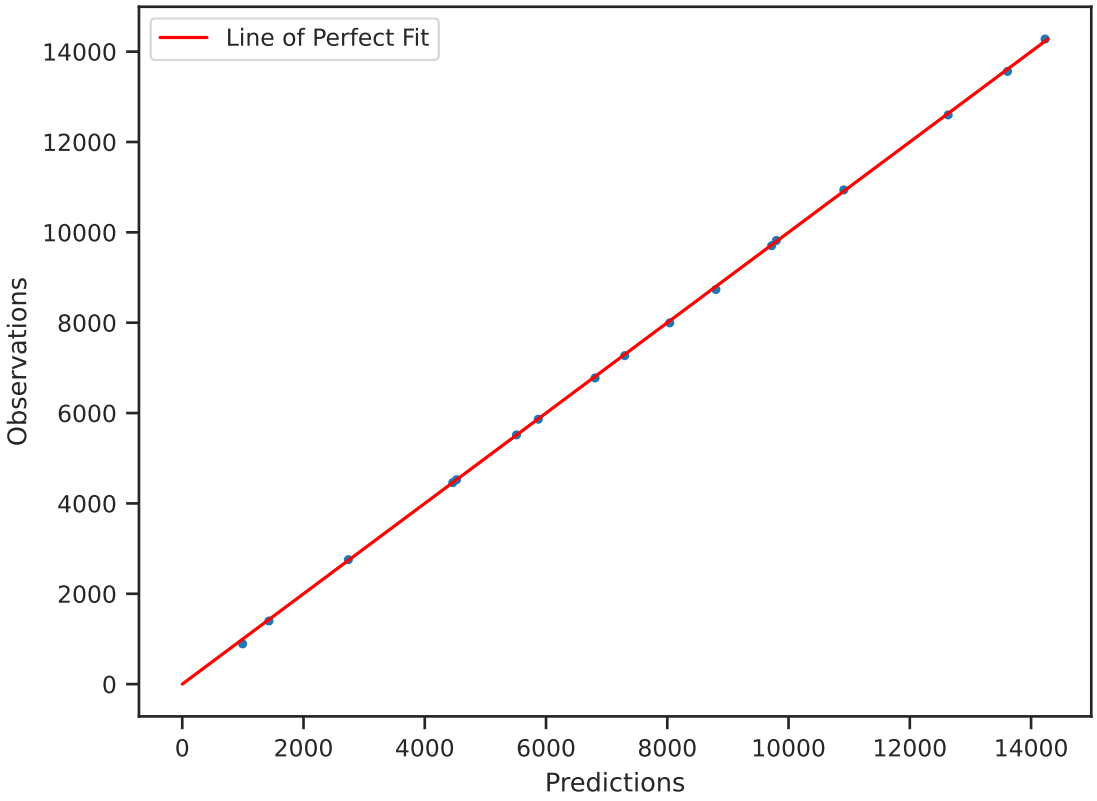
[93]:
```
from sklearn.preprocessing import PolynomialFeatures

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

Phi_train = PolynomialFeatures(3).fit_transform(x_train)
model = ARDRegression(fit_intercept=False).fit(Phi_train, y_train)

y_pred, y_pred_std = model.predict(Phi_train, return_std=True)
```

Compute MSE

[94]:
```
Phi_valid = PolynomialFeatures(3).fit_transform(x_valid)
y_pred, y_pred_std = model.predict(Phi_valid, return_std=True)
print(
    f"The mean squared error of the validation data is␣
 ↪{mean_squared_error(y_valid, y_pred):0.4f} W^2"
)
```

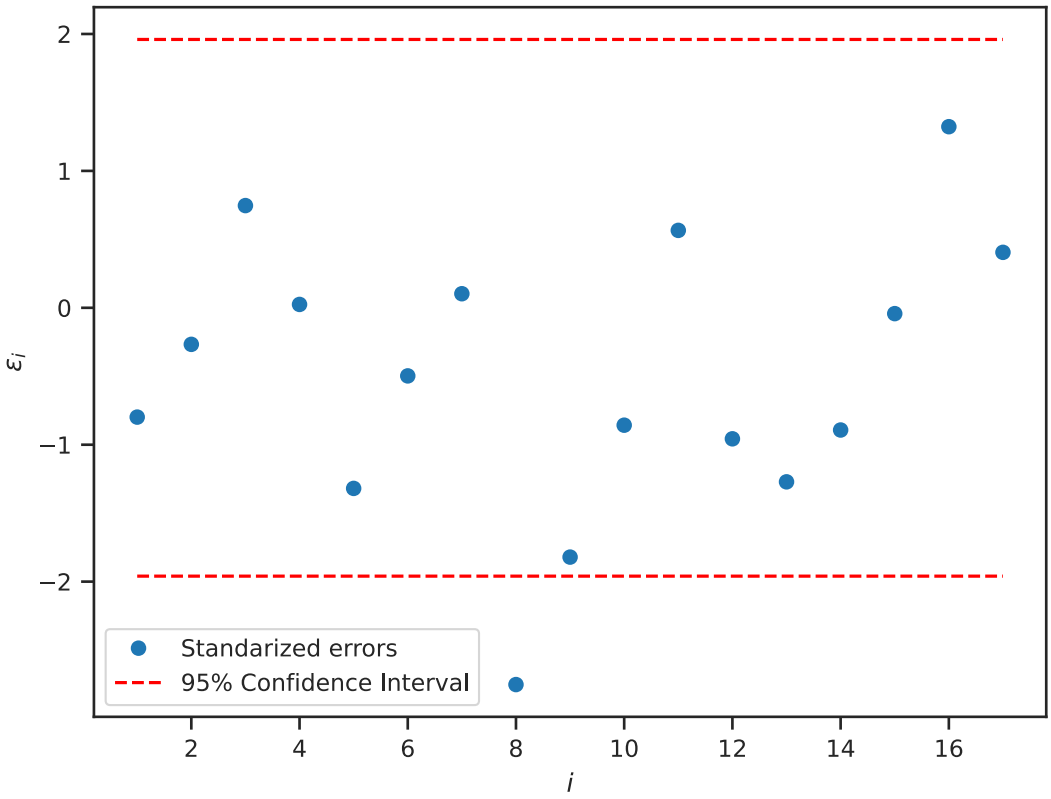
```
The mean squared error of the validation data is 1630.7316 W^2
```

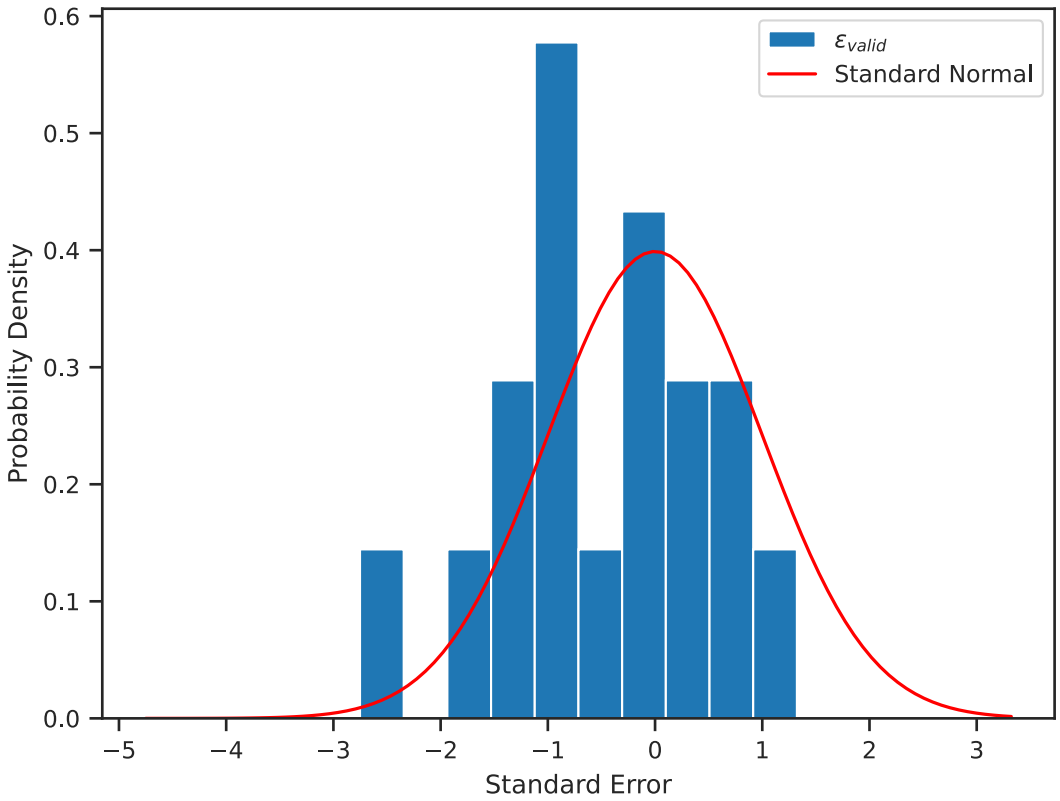
[95]:
```
fig, ax = plt.subplots()
ax.plot(y_pred, y_valid, ".")
ax.plot([0, y_valid.max()], [0, y_valid.max()], "r", label="Line of Perfect␣
 ↪Fit")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations")

ax.legend();
```



Standardized Errors

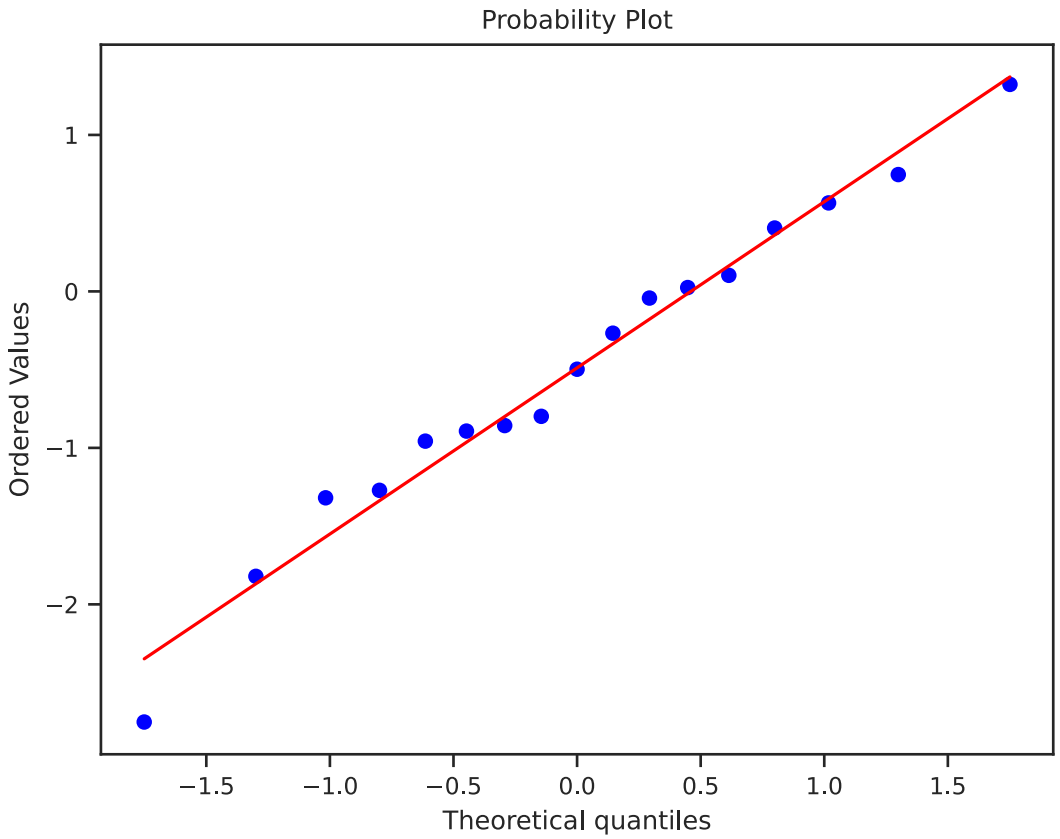
[96]:
```
standardized_errors = (y_valid - y_pred) / y_pred_std

indices = np.arange(1, standardized_errors.size + 1)

fig, ax = plt.subplots()
ax.plot(indices, standardized_errors, "o", label="Standarized errors")
ax.plot(indices, 1.96 * np.ones(standardized_errors.shape[0]), "r--")
ax.plot(
    indices,
    -1.96 * np.ones(standardized_errors.shape[0]),
    "r--",
    label="95% Confidence Interval",
)
ax.set_xlabel("$i$")
ax.set_ylabel("$\epsilon_i$")
ax.legend()

fig, ax = plt.subplots()
ax.hist(standardized_errors, density=True, label=r"$\epsilon_{valid}$")
err_sample_points = np.linspace(
    standardized_errors.min() - 2, standardized_errors.max() + 2, 100
)
ax.plot(
    err_sample_points,
    st.distributions.norm.pdf(err_sample_points),
    "r",
    label="Standard Normal",
)
ax.set_xlabel("Standard Error")
ax.set_ylabel("Probability Density")
ax.legend();
```


2
1
0
−1
−2
ε_i
Standarized errors
95% Confidence Interval
2
4
6
8
10
12
14
16
i



q-q Plot

[97]:
```
fig, ax = plt.subplots()
st.probplot(standardized_errors, dist=st.norm, plot=ax);
```



3.1.2 Subpart A.II

What is the noise variance you estimated for the Capacity?

[98]:
```
noise_variance = np.sqrt(1 / model.alpha_)
print(f"The Capacity's noise variance is {noise_variance:.4f} W^2")
```

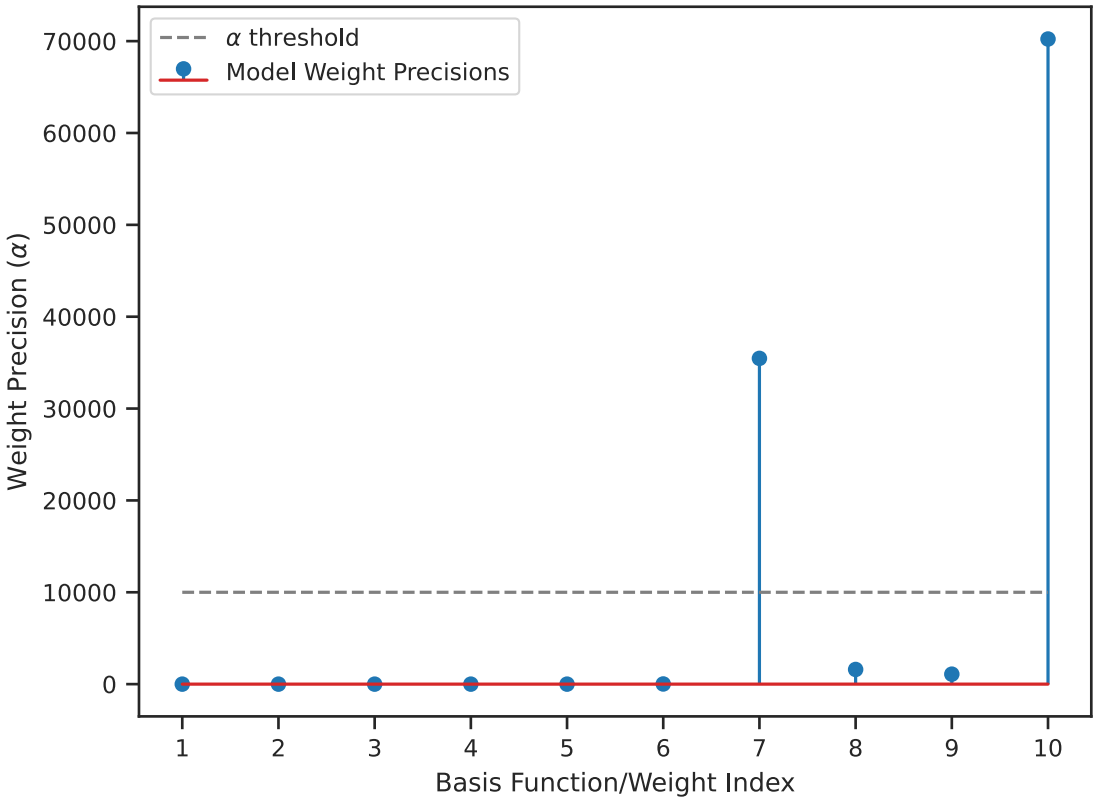
```
The Capacity's noise variance is 32.8834 W^2
```

3.1.3 Subpart A.III

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

[99]:
```
from matplotlib.ticker import MaxNLocator

fig, ax = plt.subplots()
ax.stem(np.arange(1, 11), model.lambda_, label="Model Weight Precisions")
ax.hlines(
    model.threshold_lambda, 1, 10, "gray", linestyles="--", label=r"$\alpha$␣
 ↪threshold"
```

```
)
ax.set_xlabel("Basis Function/Weight Index")
ax.set_ylabel(r"Weight Precision ($\alpha$)")
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.legend(loc="upper left");
```



The plot above shows the weight precisions (α) associated with the terms of the original polynomial model in the problem statement. Terms/basis functions with high α values are less important to the model. This shows that all the basis functions except for the ones with indices 7 and 10 ($w_7 T_e^2 T_c, w_{10} T_c^3$) are important to the model. In fact, `scikit-learn`'s implementation of ARD Regression will discard these terms since they pass the default α threshold of 10,000.

3.2 Part B - Fit the Power

3.2.1 Subpart B.I

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

[100]:
```
y = data["Power"].values
x_train, x_valid, y_train, y_valid = train_test_split(x, y)
```

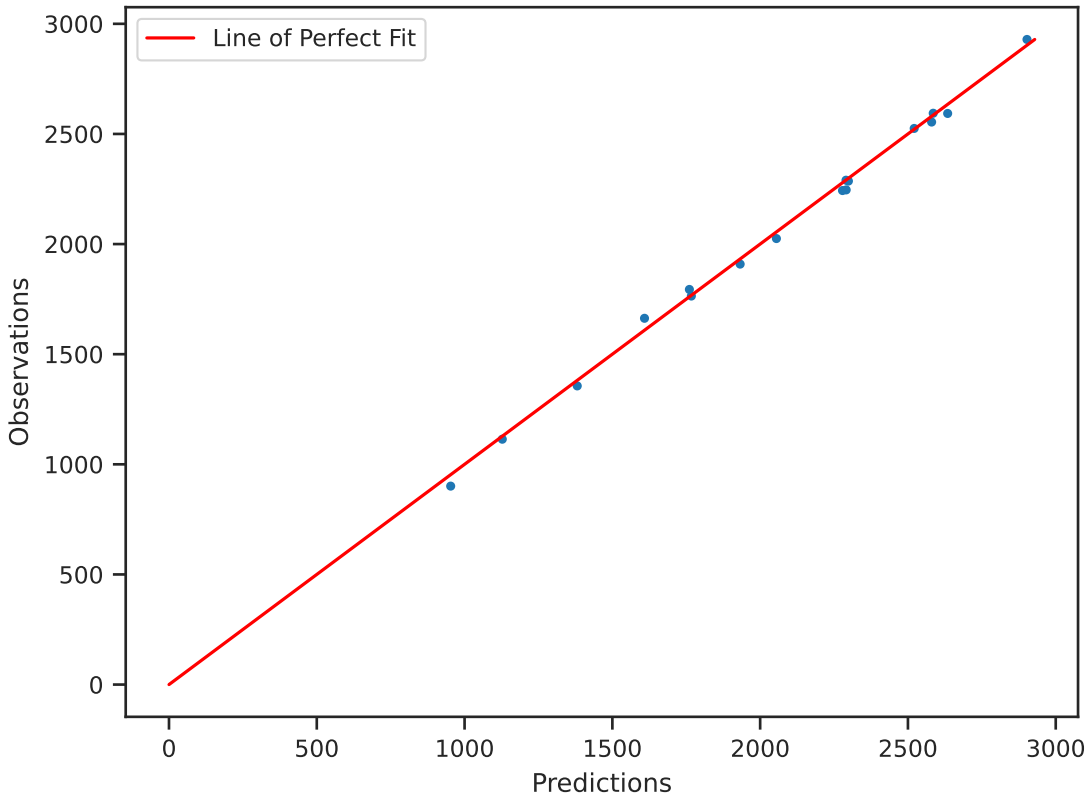
[101]:
```
Phi_train = PolynomialFeatures(3).fit_transform(x_train)
model = ARDRegression(fit_intercept=False).fit(Phi_train, y_train)

Phi_valid = PolynomialFeatures(3).fit_transform(x_valid)
y_pred, y_pred_std = model.predict(Phi_valid, return_std=True)
```

Compute MSE

[102]:
```
Phi_valid = PolynomialFeatures(3).fit_transform(x_valid)
y_pred, y_pred_std = model.predict(Phi_valid, return_std=True)
print(
    f"The mean squared error of the validation data is␣
 ↪{mean_squared_error(y_valid, y_pred):0.4f} W^2"
)
```

```
The mean squared error of the validation data is 921.4311 W^2
```

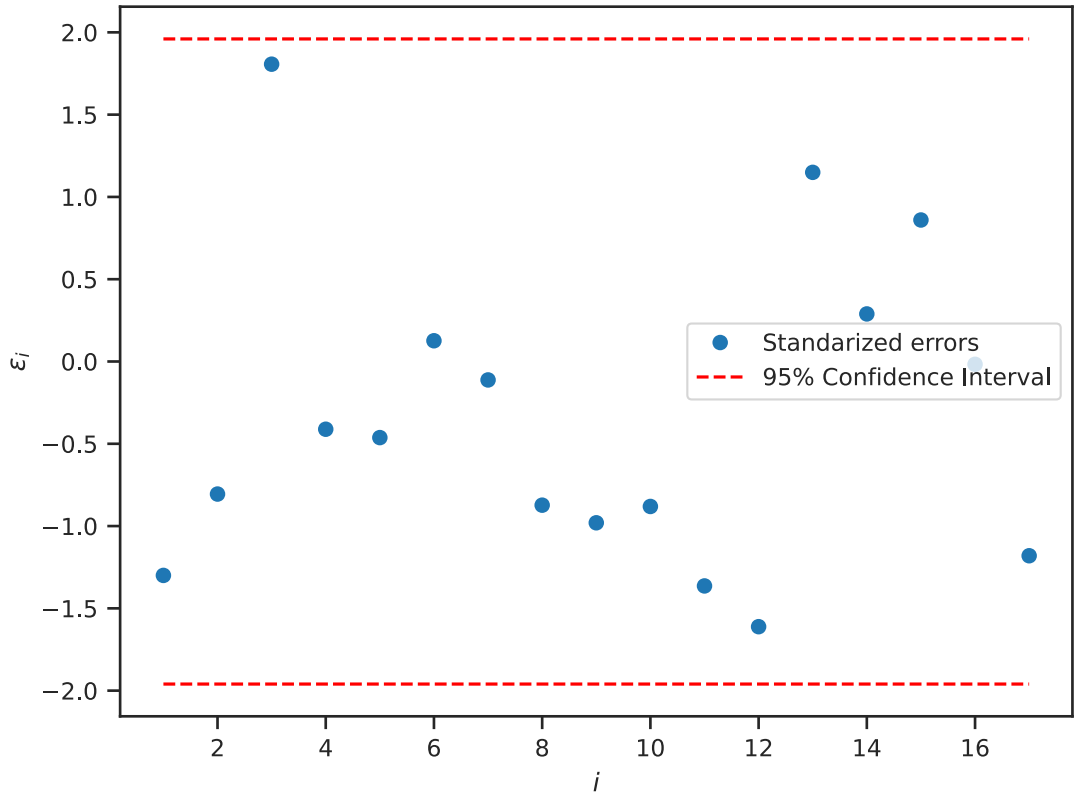
[103]:
```
fig, ax = plt.subplots()
ax.plot(y_pred, y_valid, ".")
ax.plot([0, y_valid.max()], [0, y_valid.max()], "r", label="Line of Perfect␣
 ↪Fit")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations")

ax.legend();
```

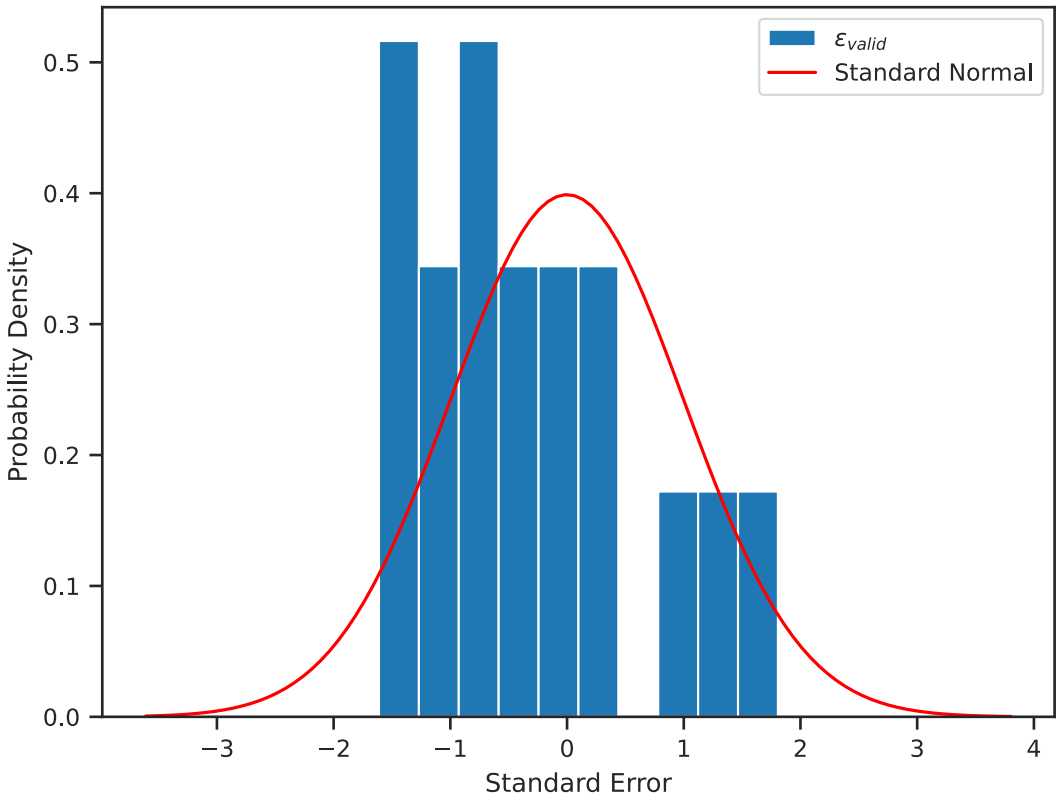


Standardized Errors

[104]:
```
standardized_errors = (y_valid - y_pred) / y_pred_std

indices = np.arange(1, standardized_errors.size + 1)

fig, ax = plt.subplots()
ax.plot(indices, standardized_errors, "o", label="Standarized errors")
ax.plot(indices, 1.96 * np.ones(standardized_errors.shape[0]), "r--")
ax.plot(
    indices,
    -1.96 * np.ones(standardized_errors.shape[0]),
    "r--",
    label="95% Confidence Interval",
)
ax.set_xlabel("$i$")
ax.set_ylabel("$\epsilon_i$")
ax.legend()

fig, ax = plt.subplots()
ax.hist(standardized_errors, density=True, label=r"$\epsilon_{valid}$")
```

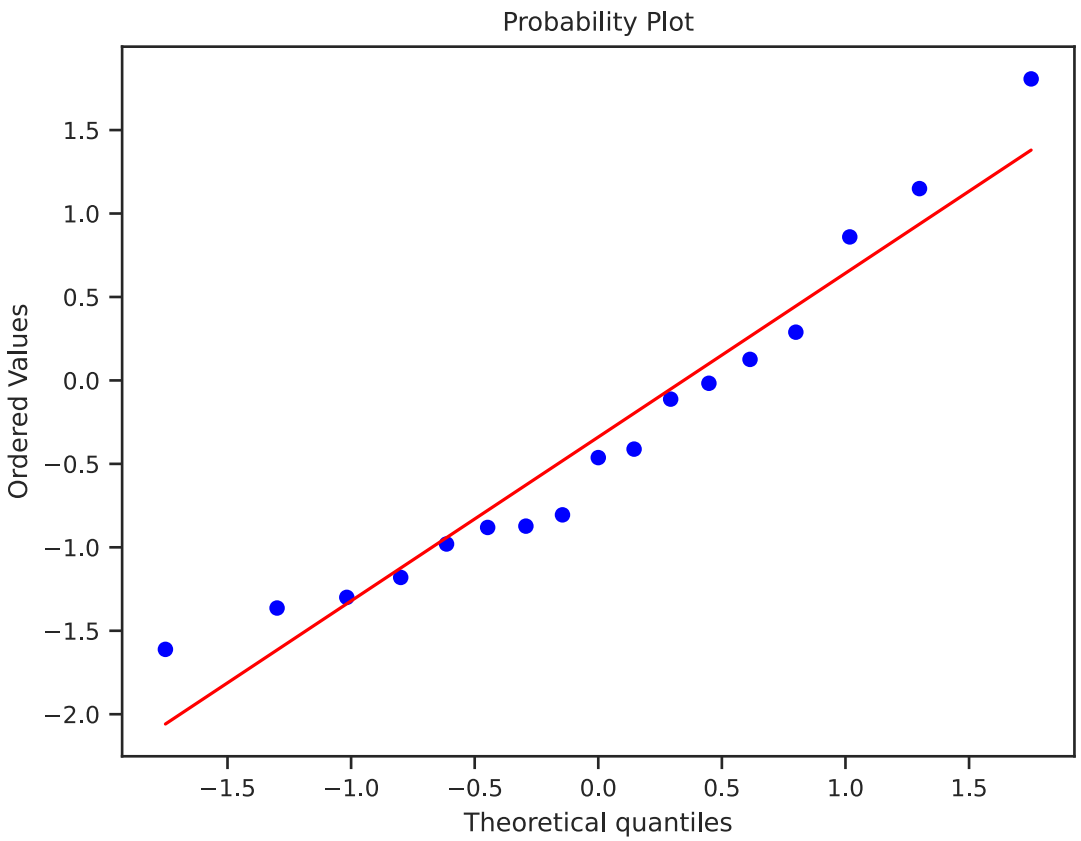
```
err_sample_points = np.linspace(
    standardized_errors.min() - 2, standardized_errors.max() + 2, 100
)
ax.plot(
    err_sample_points,
    st.distributions.norm.pdf(err_sample_points),
    "r",
    label="Standard Normal",
)
ax.set_xlabel("Standard Error")
ax.set_ylabel("Probability Density")
ax.legend();
```





q-q Plot

[105]:
```
fig, ax = plt.subplots()
st.probplot(standardized_errors, dist=st.norm, plot=ax);
```



3.2.2 Subpart B.II

What is the noise variance you estimated for the Power?

[106]:
```
noise_variance = np.sqrt(1 / model.alpha_)
print(f"The Power's noise variance is {noise_variance:.4f} W^2")
```

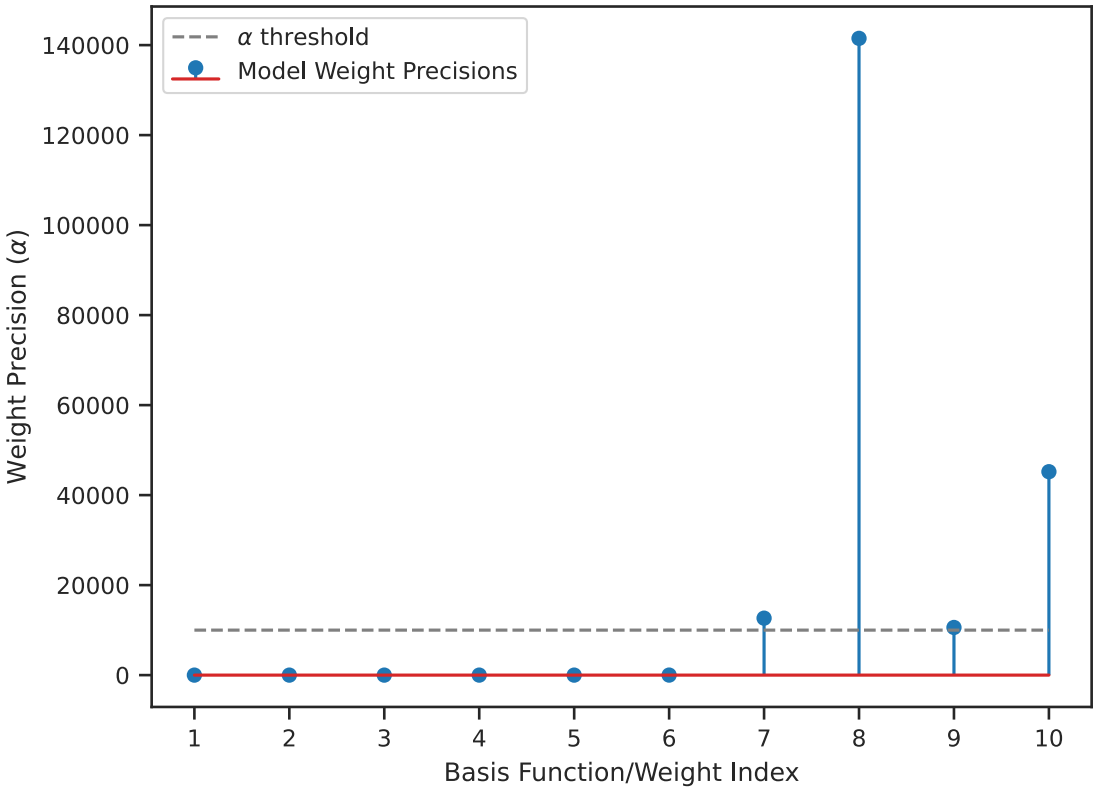
```
The Power's noise variance is 28.1023 W^2
```

3.2.3 Subpart B.III

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

[107]:
```
from matplotlib.ticker import MaxNLocator

fig, ax = plt.subplots()
ax.stem(np.arange(1, 11), model.lambda_, label="Model Weight Precisions")
ax.hlines(
    model.threshold_lambda, 1, 10, "gray", linestyles="--", label=r"$\alpha$␣
 ↪threshold"
```

```
)
ax.set_xlabel("Basis Function/Weight Index")
ax.set_ylabel(r"Weight Precision ($\alpha$)")
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.legend(loc="upper left");
```



This plot shows that all the basis functions except for the ones with indices 7, 8, 9, and 10 ($w_7 T_e^2 T_c, w_8 T_e T_c^2, w_9 T_e^3, w_{10} T_c^3$) are important to the model. `scikit-learn`'s implementation of ARD Regression will discard these terms since they pass the default α threshold of 10,000.

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 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).

[108]:
```
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.

[109]:
```
raw_data = pd.read_csv("challenger_data.csv")
raw_data
```

[109]:

	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:

[110]:
```
clean_data_0 = raw_data.dropna()
clean_data_0
```

[110]:

	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.

[111]:
```
clean_data = clean_data_0[:-1]
clean_data
```

[111]:

	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:

[112]:
```
x = clean_data["Temperature"].values
x
```

[112]:
```
array([66, 70, 69, 68, 67, 72, 73, 70, 57, 63, 70, 78, 67, 53, 67, 75, 70,
       81, 76, 79, 75, 76, 58])
```

[113]:
```
y = clean_data["Damage Incident"].values.astype(np.float64)
y
```

[113]:
```
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

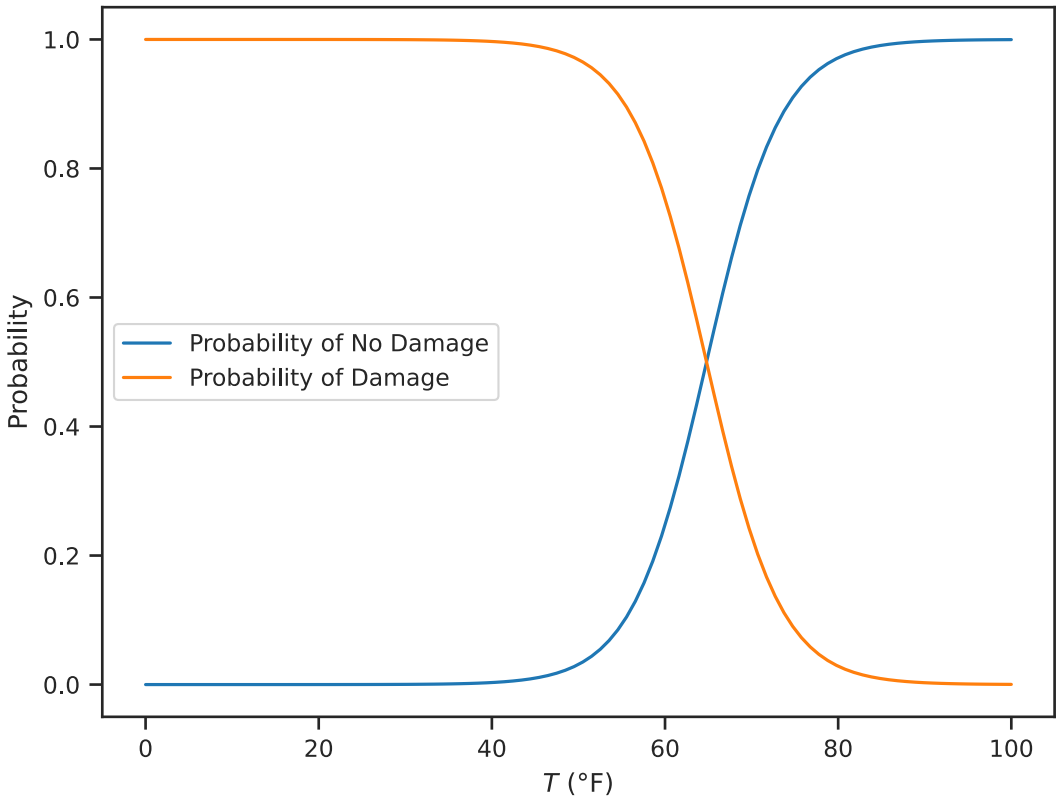
Perform logistic regression between the temperature (x) and the damage label (y). Refrain from validating because there is little data. Just use a simple model so that you don't overfit.

[114]:
```
from sklearn.linear_model import LogisticRegression

Phi_train = lambda x: np.stack([np.ones_like(x), x]).T

model = LogisticRegression(penalty=None, fit_intercept=False).fit(Phi_train(x),␣
↪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.

[115]:
```
T_samples = np.linspace(0, 100, 100)
y_pred = model.predict_proba(Phi_train(T_samples))

fig, ax = plt.subplots()

ax.plot(T_samples, y_pred[:, 0], label="Probability of No Damage")
ax.plot(T_samples, y_pred[:, 1], label="Probability of Damage")
ax.set_xlabel("$T$ ($\degree$F)")
ax.set_ylabel("Probability")
plt.legend();
```



4.3 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.

[116]:
```
p_31F = model.predict_proba(np.array([[1, 31]]))[0]
print(f"The probability of damage at 31 degrees F is {p_31F[1]:.6f}")
```

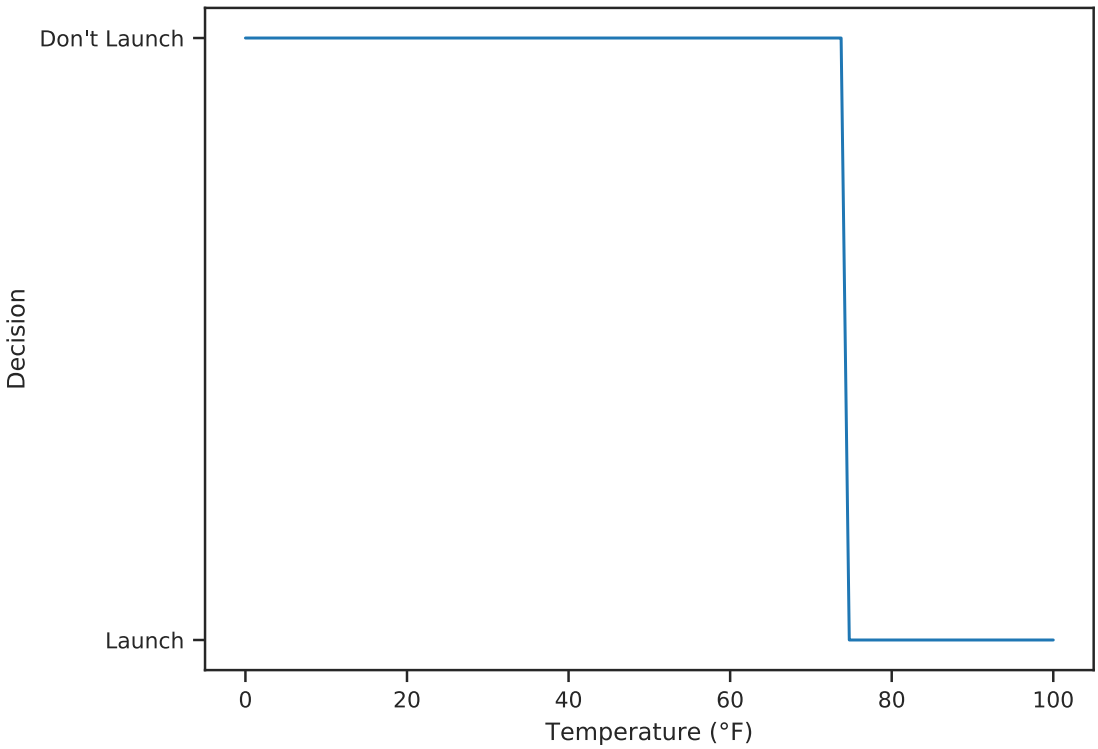
```
The probability of damage at 31 degrees F is 0.999609
```

[117]:
```
cost_matrix = np.array([[0, 10], [1, 0]])

cost_expected = np.array(
    [
        p_31F[0] * cost_matrix[0, 0] + p_31F[1] * cost_matrix[0, 1],
        p_31F[0] * cost_matrix[1, 0] + p_31F[1] * cost_matrix[1, 1],
    ]
)

if cost_expected[0] < cost_expected[1]:
```

```
    least_cost_label = "no damage"
else:
    least_cost_label = "damage"

import textwrap

print(
    textwrap.fill(
        (
            "Given the high probability of Damage at 31F and the cost matrix␣
 ↪that favors incorrectly choosing"
            f" Damage rather than No Damage, the expected loss of picking No␣
 ↪Damage at 31F is {cost_expected[0]:.6f}"
            f" while the expected loss of picking Damage is {cost_expected[1]:.
 ↪6f}. Therefore, to minimize the expected"
            f" loss, we should assume that {least_cost_label} will occur and␣
 ↪{'NOT ' if least_cost_label=='damage' else ''}proceed to launch."
        ),
        80,
    )
)
```

```
Given the high probability of Damage at 31F and the cost matrix that favors
incorrectly choosing Damage rather than No Damage, the expected loss of picking
No Damage at 31F is 9.996088 while the expected loss of picking Damage is
0.000391. Therefore, to minimize the expected loss, we should assume that damage
will occur and NOT proceed to launch.
```

[118]:
```
cost_expected = np.array(
    [
        y_pred[:, 0] * cost_matrix[0, 0] + y_pred[:, 1] * cost_matrix[0, 1],
        y_pred[:, 0] * cost_matrix[1, 0] + y_pred[:, 1] * cost_matrix[1, 1],
    ]
)
optimal_choice = np.argmin(cost_expected, axis=0)

fig, ax = plt.subplots()

ax.plot(T_samples, optimal_choice)
ax.set_yticks([0, 1])
ax.set_yticklabels(["Launch", "Don't Launch"])
ax.set_ylabel("Decision")
ax.set_xlabel("Temperature ($\degree$F)");
```



The decision boundary is around 75 degrees F. This due to the strong preference toward safety that we chose in our cost matrix. While there were not many days that meet this criterion in the sample data, compromising on this choice would endanger the mission's safety, which is unfortunately what we saw happen play out in history. Instead of forcing the launch, the team should have waited for a day with optimal temperature or focused on improving the robustness of the O-rings.