homework_05

October 15, 2020

1 Homework 5

Due: 10/22/2020 on gradescope

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 can either:
  - Type the answer using the built-in latex capabilities. In this case, simply export the notebook as a pdf and upload it on gradescope; or
  - You can print the notebook (after you are done with all the code), write your answers by hand, scan, turn your response to a single pdf, and upload on gradescope.
- The total homework points are 100. Please note that the problems are not weighed equally.

Note: Please match all the pages corresponding to each of the questions when you submit on gradescope.

1.3 Student details

- **First Name:**
- **Last Name:**
- **Email:**

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import seaborn as sns
sns.set_context('paper')
sns.set_style('white')
import scipy.stats as st
# A helper function for downloading files
import requests
import os
```

```
def download(url, local_filename=None):
    """
    Downloads the file in the ``url`` and saves it in the current working␣
 ↪directory.
    """
    data = requests.get(url)
    if local_filename is None:
        local_filename = os.path.basename(url)
    with open(local_filename, 'wb') as fd:
        fd.write(data.content)
```

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, putting it in your Google drive, mounting the drive, and changing to the directory of the file (see Problem 0 in Homework; or
- Downloading the file to the working directory of this notebook with this code:

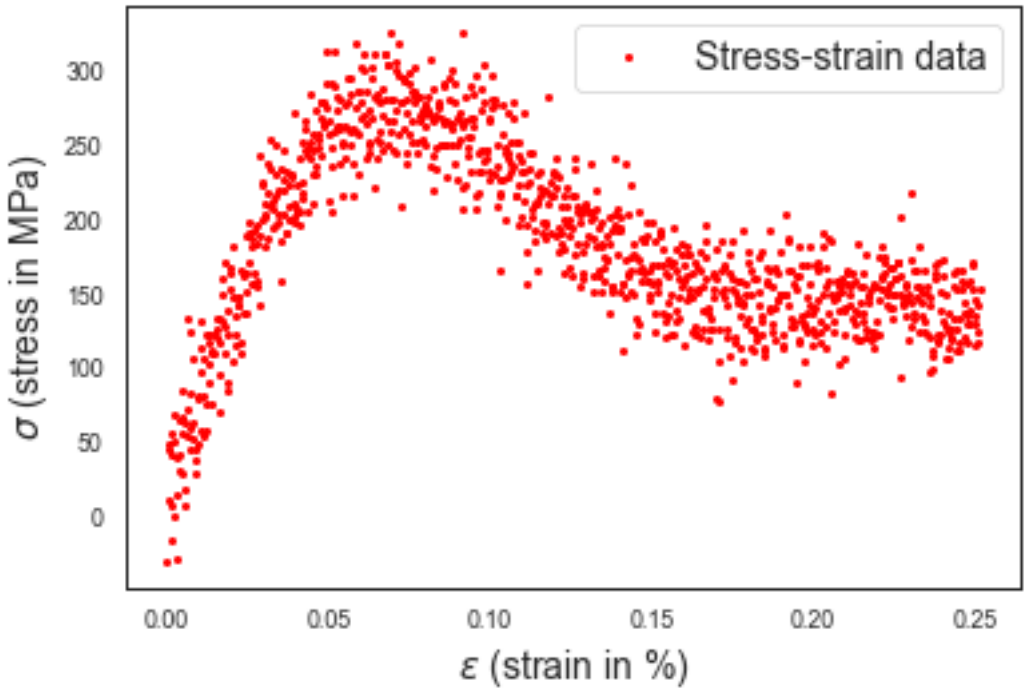
[2]:
```
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
 ↪master/homework/stress_strain.txt'
download(url)
```

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

[3]:
```
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:

[4]:
```
x = data[:, 0] # Strain
y = data[:, 1] # Stress in MPa
plt.figure()
plt.plot(x, y, 'ro', markersize=2, label = 'Stress-strain data')
plt.xlabel('$\epsilon$ (strain in %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```



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:

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

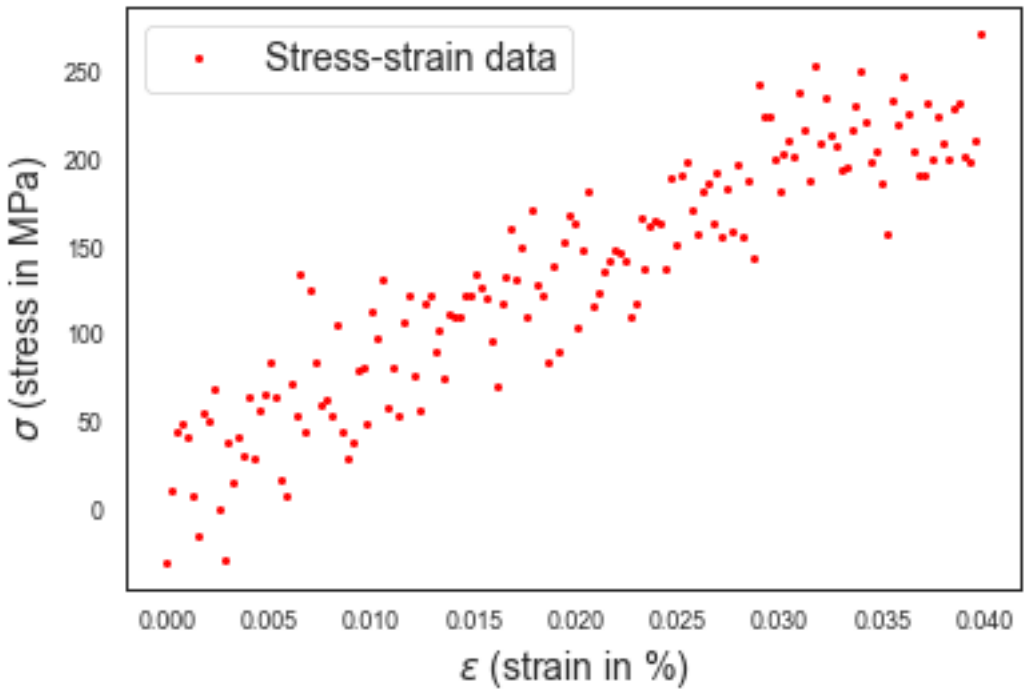
The constant E is known as the *Young modulus* of the material. Assume that you measure ϵ without any noise, but your measured σ is noisy.

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.

[5]:
```
# 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 %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```

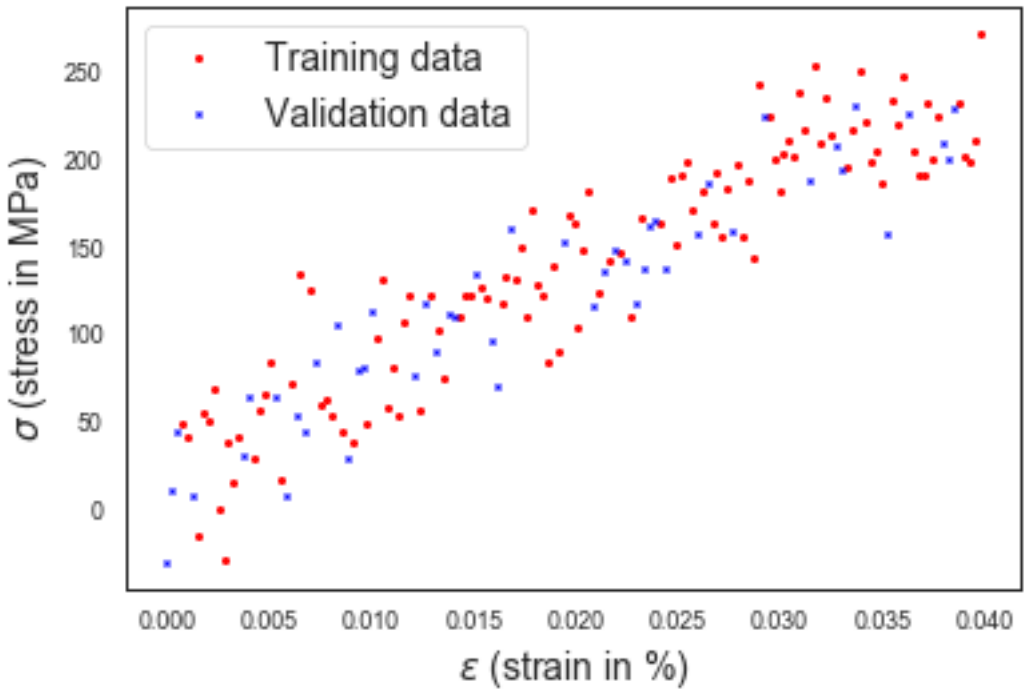


Split your data into training and validation:

[6]:
```
# Split the data into training and validation datasets
# Hint: Consult the hands-on activities of the lectures
# Figure out how many observations you have
num_obs = x_rel.shape[0]
# Select what percentage you want to put in the training data
train_percentage = 0.7
# Figure out how many training points you are going to use:
num_train = int(num_obs * train_percentage)
# Figure out how many validation points you are going to use:
num_valid = num_obs - num_train
print('num_train = {0:d}, num_valid = {1:d}'.format(num_train, num_valid))
```

```
# Before splitting the data, randomly permute rows
idxs = np.arange(x_rel.shape[0])
permuted_idxs = np.random.permutation(idxs)
# Now we split them:
# Get the x's and the y's for regression
x_train = x_rel[permuted_idxs[:num_train]]
y_train = y_rel[permuted_idxs[:num_train]]
# Get the x's and the y's for validation
x_valid = x_rel[permuted_idxs[num_train:]]
y_valid = y_rel[permuted_idxs[num_train:]]
# Let's also plot them to make sure we don't have garbage
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 %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```

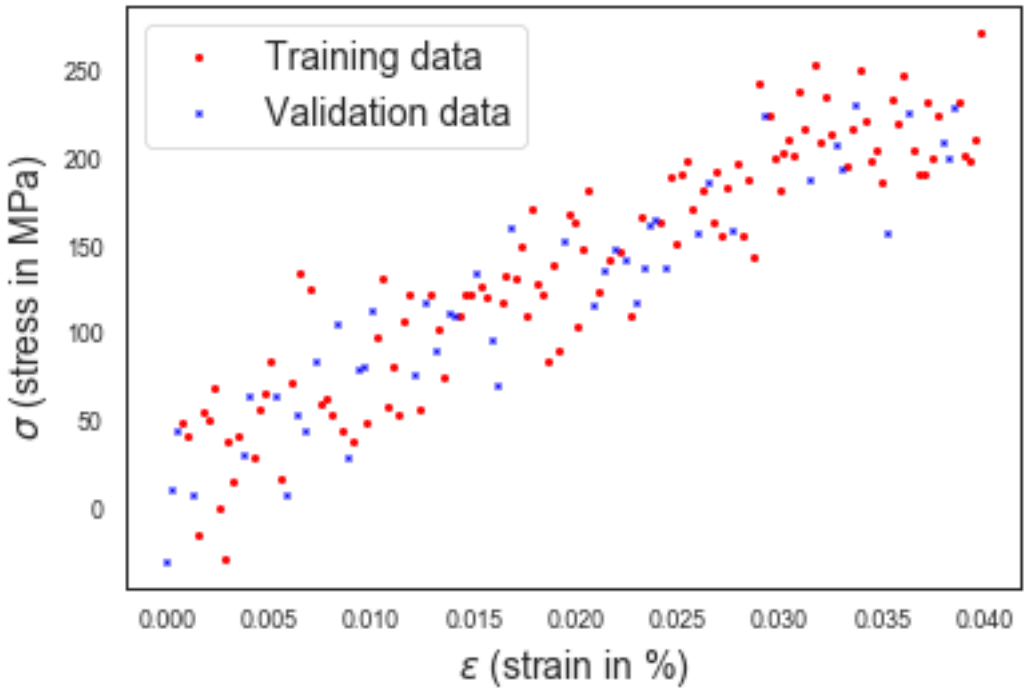
```
num_train = 111, num_valid = 48
```



Use the following to visualize your split:

[7]:
```
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 %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```



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.

[8]:
```
# We just need the design matrix.
# We know that the constant should be zero because of the physics of the␣
 ↪problem.
# So, let's force it to be zero by NOT including the constant term in our model
Phi_train = x_train.reshape((num_train, 1))
# Performing Bayesian linear regression using scikit learn
from sklearn.linear_model import BayesianRidge
model = BayesianRidge(fit_intercept=False).fit(Phi_train, y_train)
# Get the parameters as in the hands-on activity:
sigma = np.sqrt(1.0 / model.alpha_)
print('sigma = {0:1.2f}'.format(sigma))
alpha = np.sqrt(1.0 / model.lambda_)
print('alpha = {0:1.2f}'.format(alpha))
```

```
sigma = 30.45
alpha = 6506.70
```

2.1.3 Subpart A.III

Calculate the mean square error of the validation data.

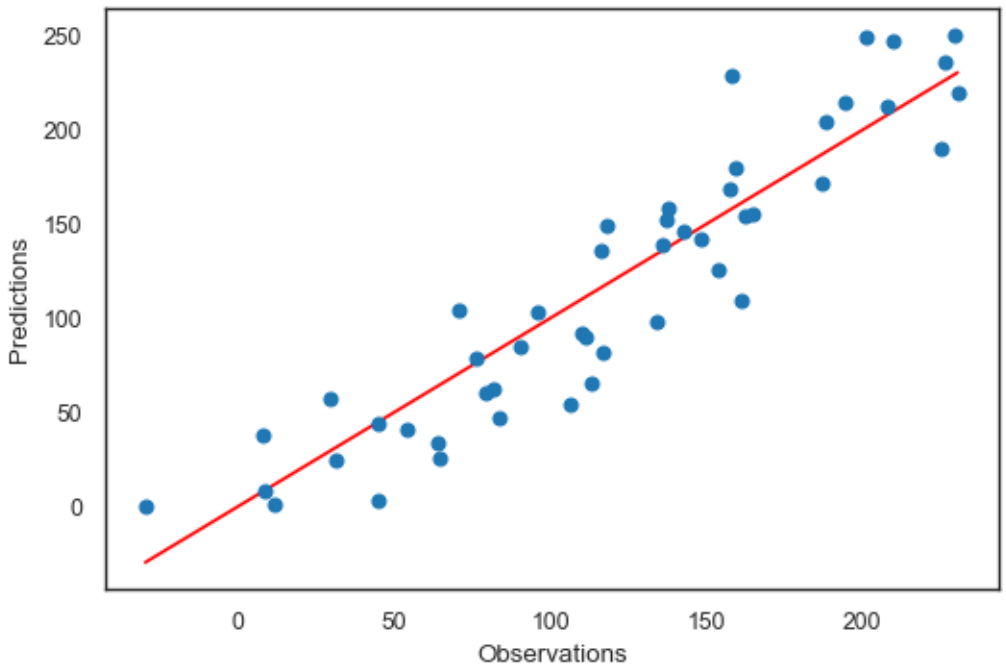
[9]:
```
# Predict on the validation data
y_valid_p_mean, y_valid_p_std = model.predict(x_valid[:, None], return_std=True)
MSE = np.mean((y_valid_p_mean - y_valid) ** 2)
print('MSE = {0:1.2f}'.format(MSE))
```

```
MSE = 757.64
```

2.1.4 Subpart A.IV

Make the observations vs predictions plot for the validation data.

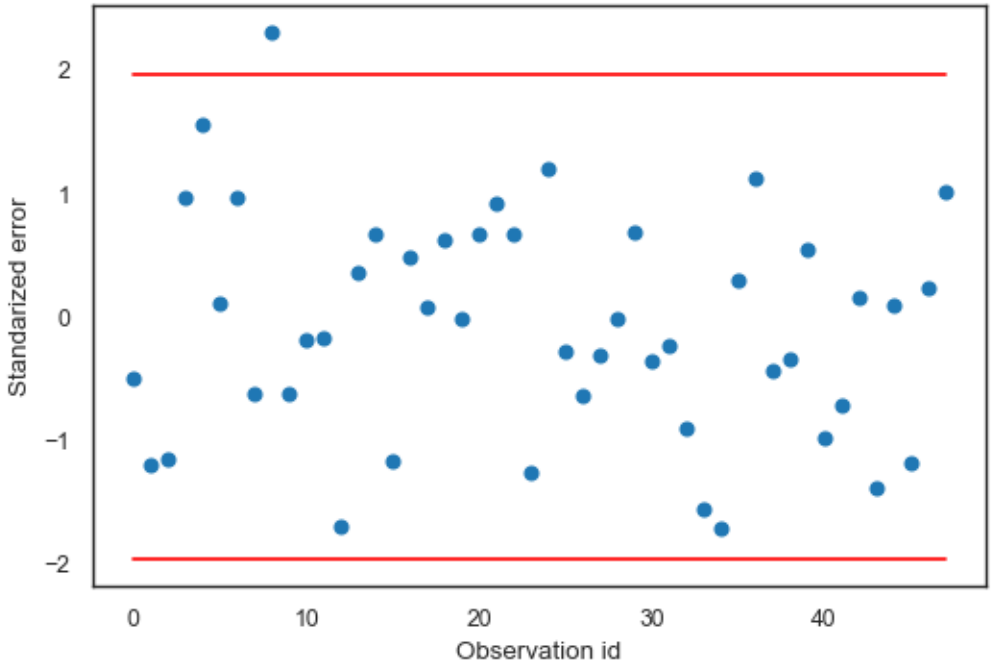
[10]:
```
fig, ax = plt.subplots(dpi=100)
yys = np.linspace(y_valid.min(), y_valid.max())
ax.plot(yys, yys, 'r')
ax.plot(y_valid, y_valid_p_mean, 'o')
ax.set_xlabel('Observations')
ax.set_ylabel('Predictions');
```



2.1.5 Subpart A.V

Compute and plot the standarized errors for the validation data.

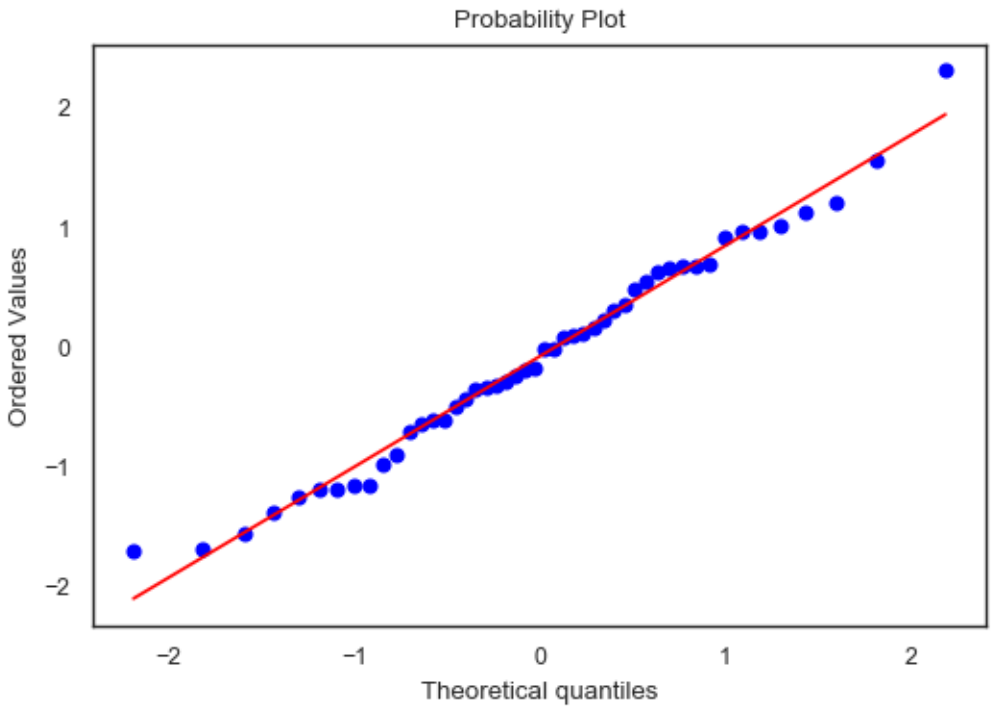
[11]:
```
std_errs = (y_valid_p_mean - y_valid) / y_valid_p_std
fig, ax = plt.subplots(dpi=100)
ax.plot(std_errs, 'o')
obs_ids = np.arange(std_errs.shape[0])
ax.plot(obs_ids, 1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.plot(obs_ids, -1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.set_xlabel('Observation id')
ax.set_ylabel('Standarized error');
```



2.1.6 Subpart A.VI

Make the quantile-quantile plot of the standarized errors.

[12]:
```
fig, ax = plt.subplots(dpi=100)
st.probplot(std_errs, dist=st.norm, plot=ax);
```

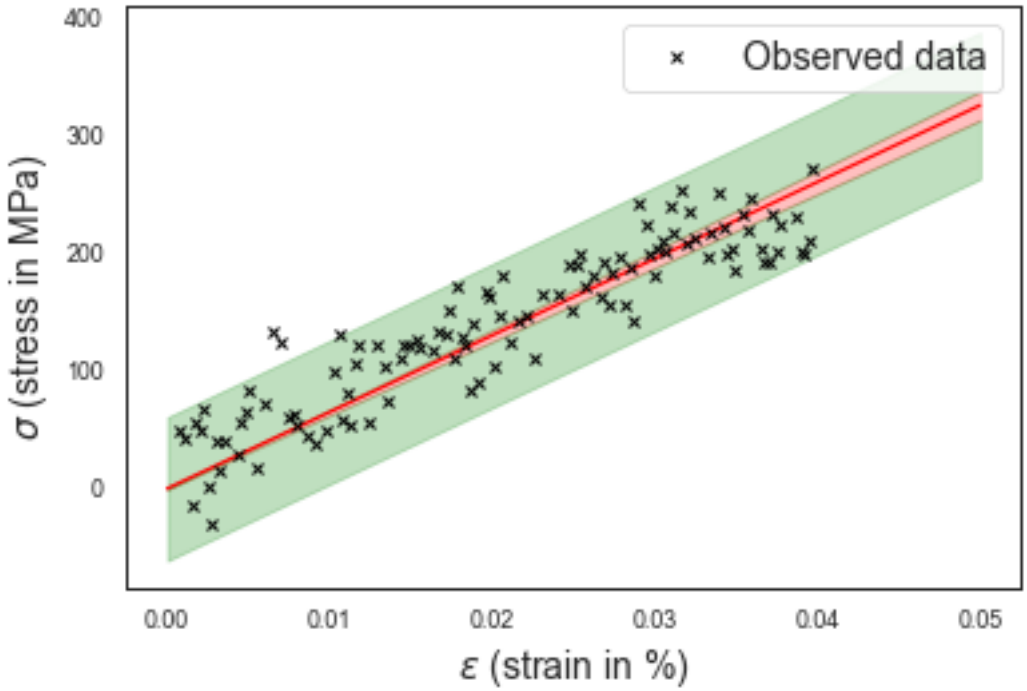


2.1.7 Subpart A.VII

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

[13]:
```
xx = np.linspace(0.0, 0.05, 100)
# Use the model to get the predictive mean and standard deviation:
yy_mean, yy_measured_std = model.predict(xx[:, None], return_std=True)
# Separate the epistemic uncertainty
yy_std = np.sqrt(yy_measured_std ** 2 - sigma**2)
# Plot
fig, ax = plt.subplots()
ax.plot(xx, yy_mean, 'r')
# Epistemic lower bound
yy_le = yy_mean - 2.0 * yy_std
# Epistemic upper bound
yy_ue = yy_mean + 2.0 * yy_std
# Epistemic + aleatory lower bound
yy_lae = yy_mean - 2.0 * yy_measured_std
# Episemic + aleatory upper bound
yy_uae = yy_mean + 2.0 * yy_measured_std
ax.fill_between(xx, yy_le, yy_ue, color='red', alpha=0.25)
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)
```

```
# plot the data again
ax.plot(x_train, y_train, 'kx', label='Observed data')
plt.xlabel('$\epsilon$ (strain in %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```

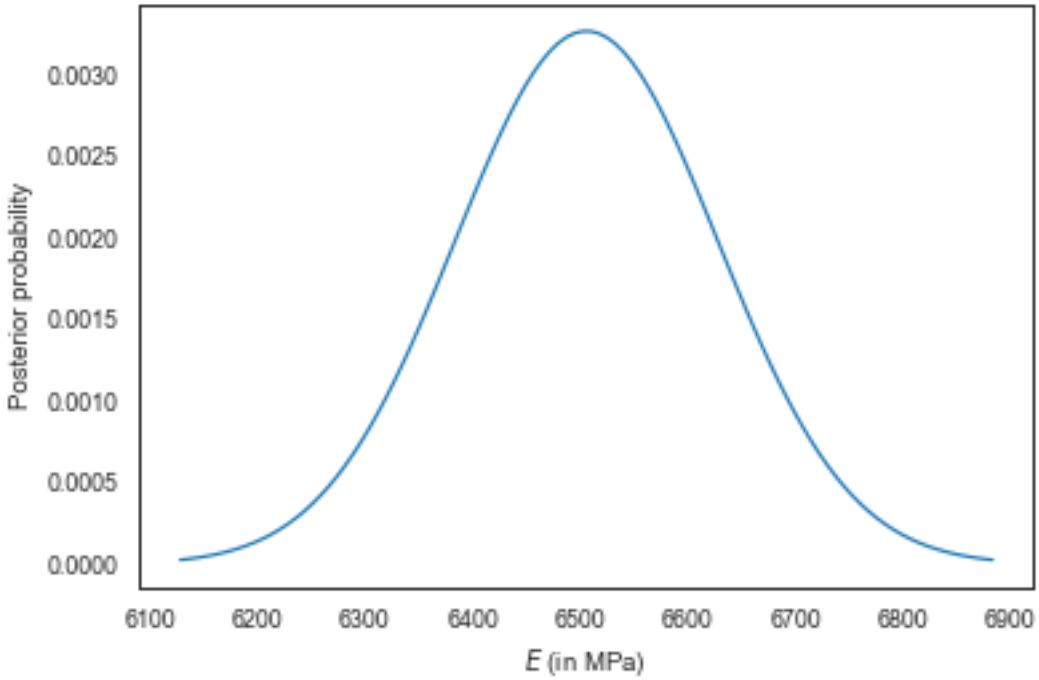


2.1.8 Subpart A. VIII

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

[14]:
```
# The posterior mean of the weights is here
m_norm = model.coef_
print(m_norm)
# The posterior covariance matrix for the weights is here
S_norm = model.sigma_
print(S_norm)
# Create a Normal random variable to represent this
E_post = st.norm(loc=m_norm[0], scale=np.sqrt(S_norm[0, 0]))
# Plot the PDF of E_post
Es = np.linspace(E_post.ppf(0.001), E_post.ppf(0.999), 100)
fig, ax = plt.subplots()
ax.plot(Es, E_post.pdf(Es))
ax.set_xlabel('$E$ (in MPa)')
ax.set_ylabel('Posterior probability');
```

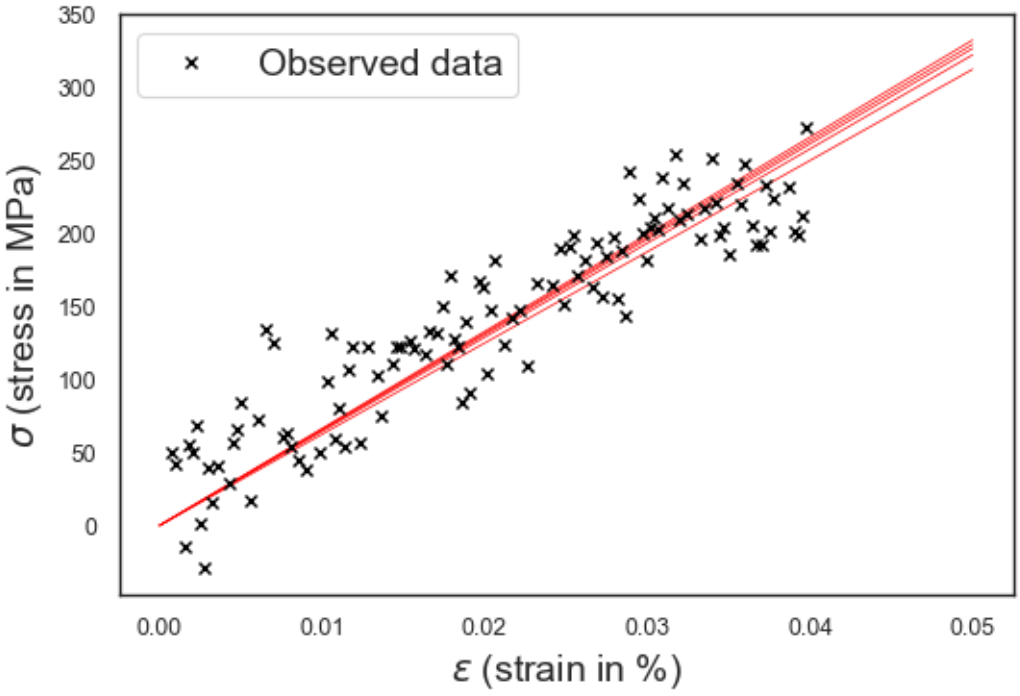
```
[6505.56599629]
[[14901.06725049]]
```



2.1.9 Subpart A.IX

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

[15]:
```
fig, ax = plt.subplots(dpi=100)
for i in range(5):
    E = E_post.rvs()
    yy_sample = E * xx
    ax.plot(xx, yy_sample, 'r', lw=0.5)
# plot the data again
ax.plot(x_train, y_train, 'kx', label='Observed data')
plt.xlabel('$\epsilon$ (strain in %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```



2.1.10 Subpart A.X

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

[16]:
```
E_l = E_post.ppf(0.025)
E_u = E_post.ppf(0.975)
print('E is in between {0:1.2f} and {1:1.2f} MPa with 95% probability!'.
 ↪format(E_l, E_u))
```

```
E is in between 6266.31 and 6744.82 MPa with 95% probability!
```

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?

[17]:
```
# I pick this:
print('E* = {0:1.2f} MPa'.format(E_post.median()))
```

```
E* = 6505.57 MPa
```

This is a bit subjective, but I would pick the median of the distribution (which here happens to be the same as the max of the posterior and the mean).

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

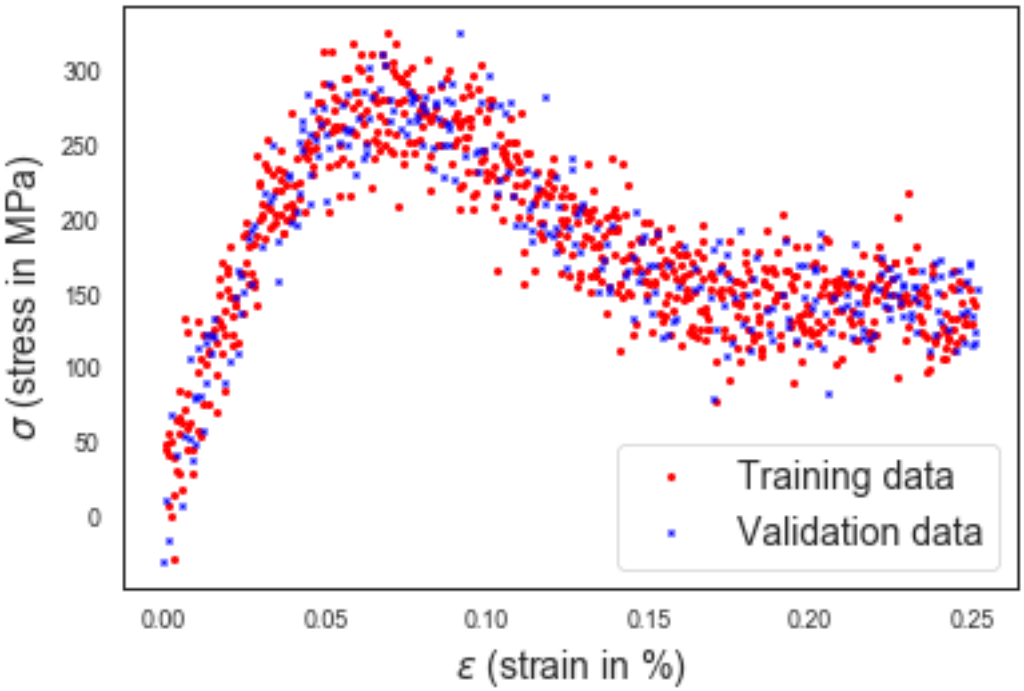
Extract training and validation data from the entire dataset.

[18]:
```
# Figure out how many observations you have
num_obs = x.shape[0]
# Select what percentage you want to put in the training data
train_percentage = 0.7
# Figure out how many training points you are going to use:
num_train = int(num_obs * train_percentage)
# Figure out how many validation points you are going to use:
num_valid = num_obs - num_train
print('num_train = {0:d}, num_valid = {1:d}'.format(num_train, num_valid))
# Before splitting the data, randomly permute rows
idxs = np.arange(x.shape[0])
permuted_idxs = np.random.permutation(idxs)
# Now we split them:
# Get the x's and the y's for regression
x_train = x[permuted_idxs[:num_train]]
y_train = y[permuted_idxs[:num_train]]
# Get the x's and the y's for validation
x_valid = x[permuted_idxs[num_train:]]
y_valid = y[permuted_idxs[num_train:]]
```

```
num_train = 700, num_valid = 301
```

Use the following to visualize your split:

[19]:
```
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 %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```



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:

$$f(\epsilon) = \begin{cases} h(\epsilon) = E\epsilon, \ \epsilon < \epsilon_l, \\ g(\epsilon) = \sum_{i=0}^{d} w_i \epsilon^i, \epsilon \geq \epsilon_l \end{cases}$$

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

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:

$$h(\epsilon_l) = g(\epsilon_l)$$
$$h'(\epsilon_l) = g'(\epsilon_l).$$

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:

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

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 end result is:

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

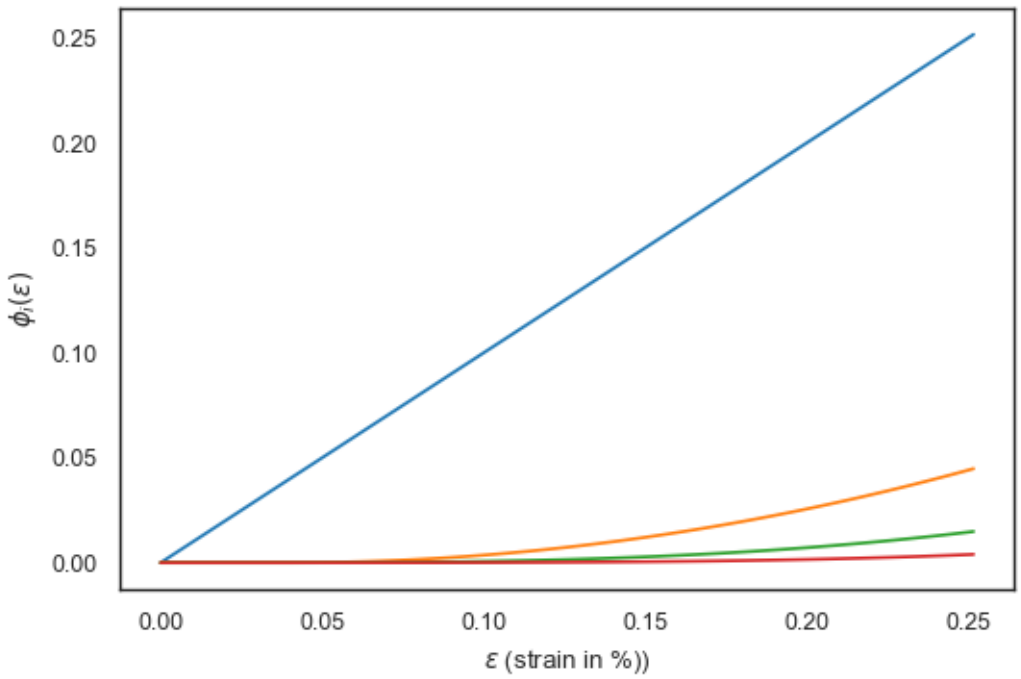
Okay. This is still a generalized linear model. This is nice. Write code for the design matrix:

[20]:
```
# Complete this code to make your model:
def compute_design_matrix(Epsilon, epsilon_l, d):
    """
    Computes 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
    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_l ** (i - 1) * Epsilon
                       + Epsilon ** i) * Step
    return Phi
```

Visualize the basis functions here:

[21]:
```
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.).

[22]:
```
from sklearn.linear_model import ARDRegression

Phi_train = compute_design_matrix(x_train, epsilon_l, d)
model = ARDRegression(fit_intercept=False).fit(Phi_train, y_train)
```

Let's calculate the MSE:

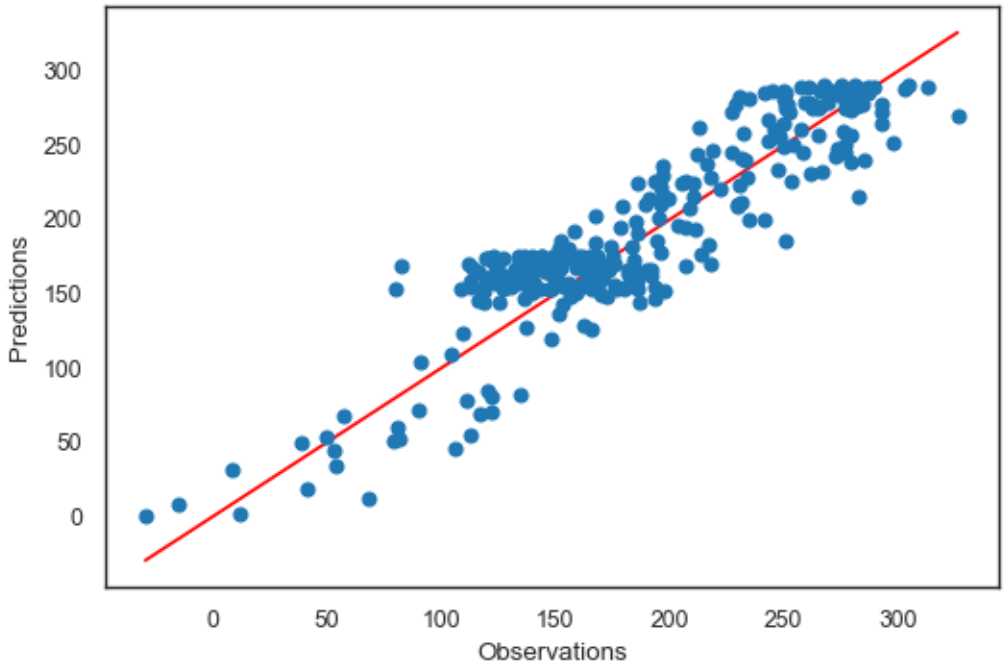
[23]:
```
Phi_valid = compute_design_matrix(x_valid, epsilon_l, d)

y_valid_p_mean, y_valid_p_std = model.predict(Phi_valid, return_std=True)
MSE = np.mean((y_valid_p_mean - y_valid) ** 2)
print('MSE = {0:1.2f}'.format(MSE))
```

```
MSE = 767.57
```

Make the observations vs predictions plot:

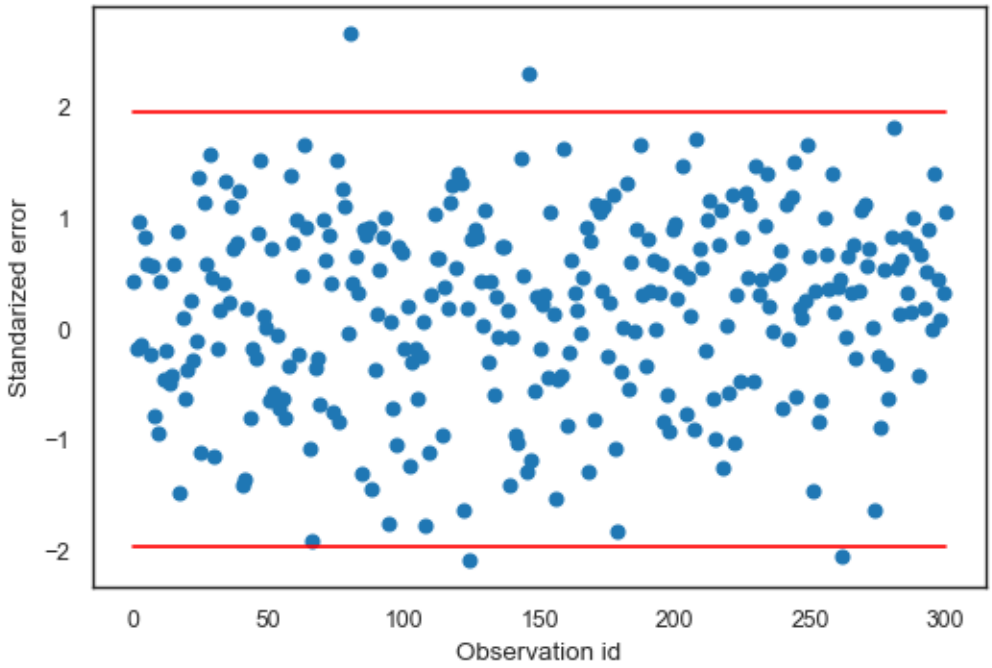
[24]:
```
fig, ax = plt.subplots(dpi=100)
yys = np.linspace(y_valid.min(), y_valid.max())
ax.plot(yys, yys, 'r')
ax.plot(y_valid, y_valid_p_mean, 'o')
```

```
ax.set_xlabel('Observations')
ax.set_ylabel('Predictions');
```



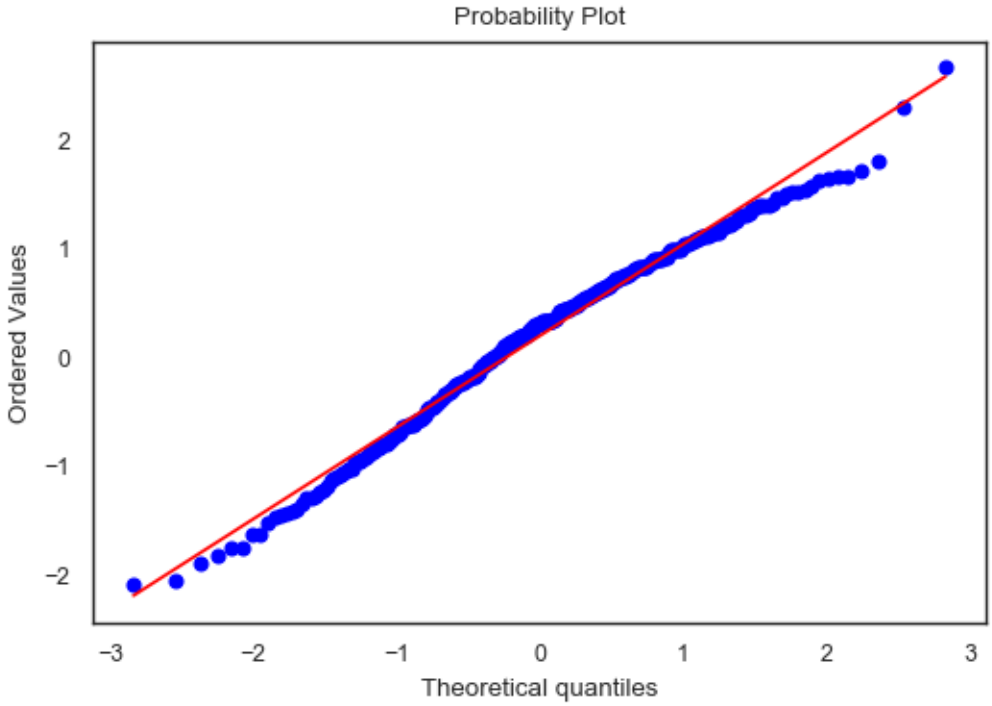
Calculate the plot the standarized errors:

[25]:
```
std_errs = (y_valid_p_mean - y_valid) / y_valid_p_std
fig, ax = plt.subplots(dpi=100)
ax.plot(std_errs, 'o')
obs_ids = np.arange(std_errs.shape[0])
ax.plot(obs_ids, 1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.plot(obs_ids, -1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.set_xlabel('Observation id')
ax.set_ylabel('Standarized error');
```



Make the quantile-quantile plot:

[26]:
```
fig, ax = plt.subplots(dpi=100)
st.probplot(std_errs, dist=st.norm, plot=ax);
```

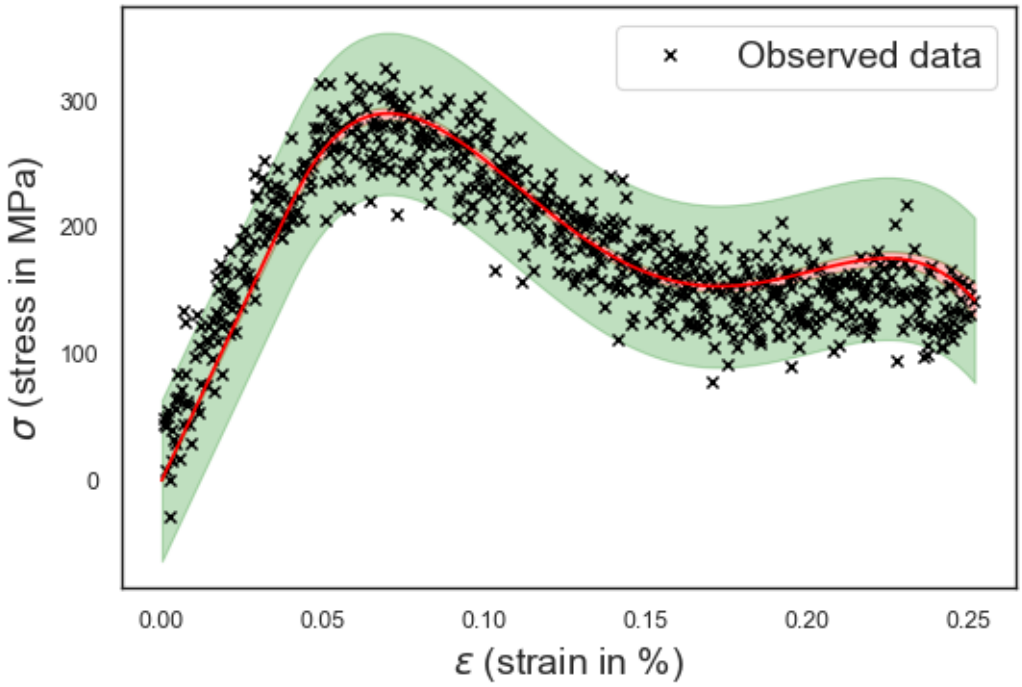


The model fit looks pretty good.

2.2.4 Subpart B.IV

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

[27]:
```
sigma = np.sqrt(1.0 / model.alpha_)
xx = np.linspace(0.0, x.max(), 100)
Phi_xx = compute_design_matrix(xx, epsilon_l, d)
# Use the model to get the predictive mean and standard deviation:
yy_mean, yy_measured_std = model.predict(Phi_xx, return_std=True)
# Separate the epistemic uncertainty
yy_std = np.sqrt(yy_measured_std ** 2 - sigma**2)
# Plot
fig, ax = plt.subplots(dpi=100)
ax.plot(x_train, y_train, 'kx', label='Observed data')
ax.plot(xx, yy_mean, 'r')
# Epistemic lower bound
yy_le = yy_mean - 2.0 * yy_std
# Epistemic upper bound
yy_ue = yy_mean + 2.0 * yy_std
# Epistemic + aleatory lower bound
yy_lae = yy_mean - 2.0 * yy_measured_std
# Episemic + aleatory upper bound
yy_uae = yy_mean + 2.0 * yy_measured_std
ax.fill_between(xx, yy_le, yy_ue, color='red', alpha=0.25)
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)
# plot the data again
plt.xlabel('$\epsilon$ (strain in %)', fontsize=14)
plt.ylabel('$\sigma$ (stress in MPa)', fontsize=14)
plt.legend(loc='best', fontsize = 14);
```



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.

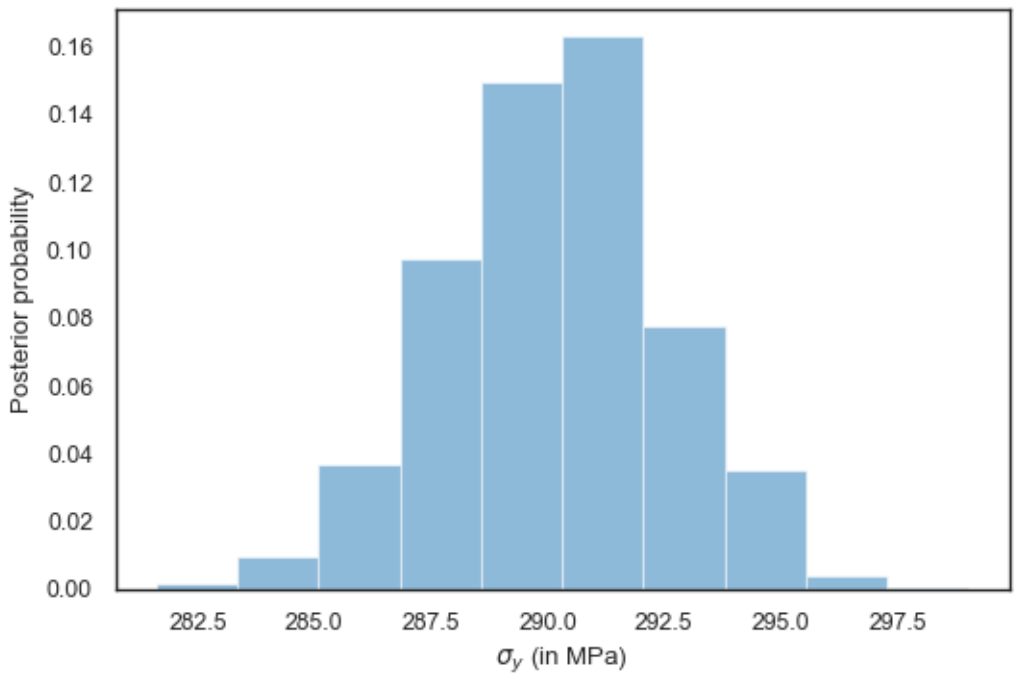
Do the following: - Visualize posterior of the ultimate strength. - Find a 95% credible interval for the ultimate strength. - Pick a value for the ultimate strength.

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.

[28]:
```
# Use Hands-on Activity 15.2
# The posterior mean of the weights is here (this is for the normalized␣
↪features, however)
m_norm = model.coef_
# The posterior covariance matrix for the weights is here (also for the␣
↪normalized features)
S_norm = model.sigma_ + 1e-6 * np.eye(model.sigma_.shape[0])
# The posterior of the weights
w_post = st.multivariate_normal(mean=m_norm, cov=S_norm)
```

```
# The number of samples to take
num_post_samples = 1000
# Storing the ultimate strength of each one of the posterior samples
ultimate_strength_post_samples = np.ndarray((num_post_samples,))
# Start taking samples
for n in range(num_post_samples):
    w_sample = w_post.rvs()
    yy_sample = np.dot(Phi_xx, w_sample)
    ultimate_strength_post_samples[n] = np.max(yy_sample)
# Do the histogram
fig, ax = plt.subplots(dpi=100)
ax.hist(ultimate_strength_post_samples, density=True, alpha=0.5)
ax.set_xlabel('$\sigma_y$ (in MPa)')
ax.set_ylabel('Posterior probability');
```

```
/Users/iliasbilionis/opt/anaconda3/lib/python3.7/site-
packages/scipy/stats/_multivariate.py:660: RuntimeWarning: covariance is not
positive-semidefinite.
  out = random_state.multivariate_normal(mean, cov, size)
```



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:

[29]:
```
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
↪master/homework/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:

[30]:
```
import pandas as pd
data = pd.read_excel('compressor_data.xlsx')
data
```

[30]:

	T_e	DT_sh	T_c	DT_sc	T_amb	f	m_dot	m_dot.1	Capacity	Power	Current	COP	Efficiency
0	-30	11	25	8	35	60	28.8	8.000000	1557	901	4.4	1.73	0.467
1	-30	11	30	8	35	60	23.0	6.388889	1201	881	4.0	1.36	0.425
2	-30	11	35	8	35	60	17.9	4.972222	892	858	3.7	1.04	0.382
3	-25	11	25	8	35	60	46.4	12.888889	2509	1125	5.3	2.23	0.548
4	-25	11	30	8	35	60	40.2	11.166667	2098	1122	5.1	1.87	0.519
..	…	…	…	…	…	..	…	…	…	…	…	…	…
60	10	11	45	8	35	60	245.2	68.111111	12057	2525	11.3	4.78	0.722
61	10	11	50	8	35	60	234.1	65.027778	10939	2740	12.3	3.99	0.719
62	10	11	55	8	35	60	222.2	61.722222	9819	2929	13.1	3.35	0.709
63	10	11	60	8	35	60	209.3	58.138889	8697	3091	13.7	2.81	0.693
64	10	11	65	8	35	60	195.4	54.277778	7575	3223	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.

[31]:
```
# 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
```

[31]:
```
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]])
```

[32]:
```
# Here is how to extract the Capacity
y = data['Capacity'].values
y
```

[32]:
```
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.

3.1 Part A - The Capacity

3.1.1 Subpart A.I - Fit the capacity

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

[33]:
```
# For polynomial features
from sklearn.preprocessing import PolynomialFeatures
# For splitting the data
from sklearn.model_selection import train_test_split
# Split the data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.33)
# Polynomials features
poly = PolynomialFeatures(degree=3)
# Design matrix for train data
Phi_train = poly.fit_transform(X_train)
# Fit with ARD
model = ARDRegression().fit(Phi_train, y_train)
```

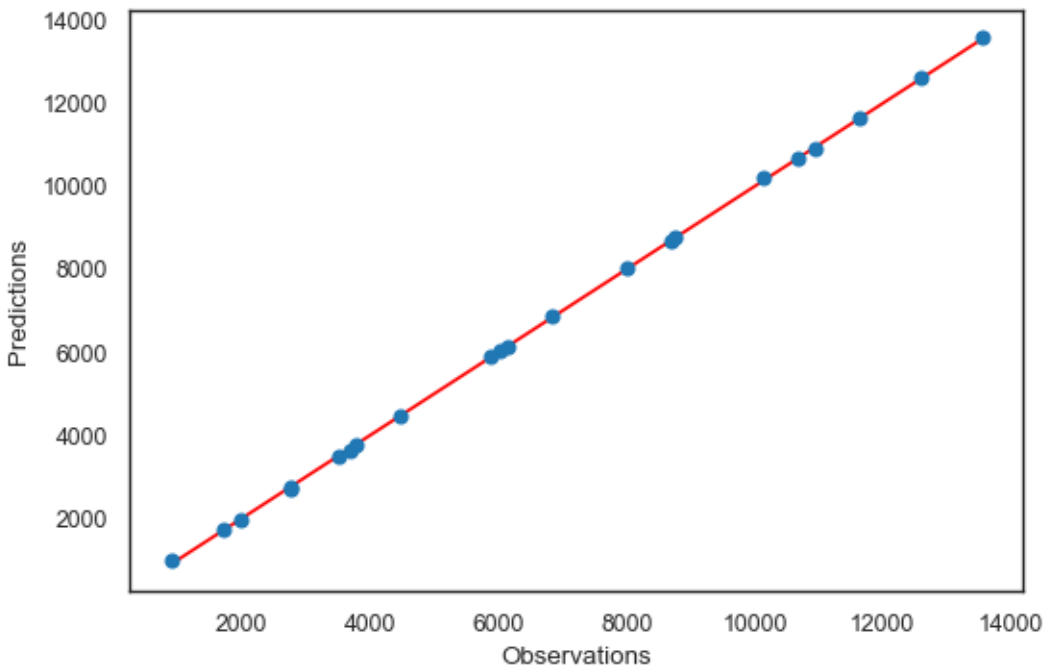
The mean square error:

[34]:
```
Phi_valid = poly.fit_transform(X_valid)

y_valid_p_mean, y_valid_p_std = model.predict(Phi_valid, return_std=True)
MSE = np.mean((y_valid_p_mean - y_valid) ** 2)
print('MSE = {0:1.2f}'.format(MSE))
```

```
MSE = 1289.42
```

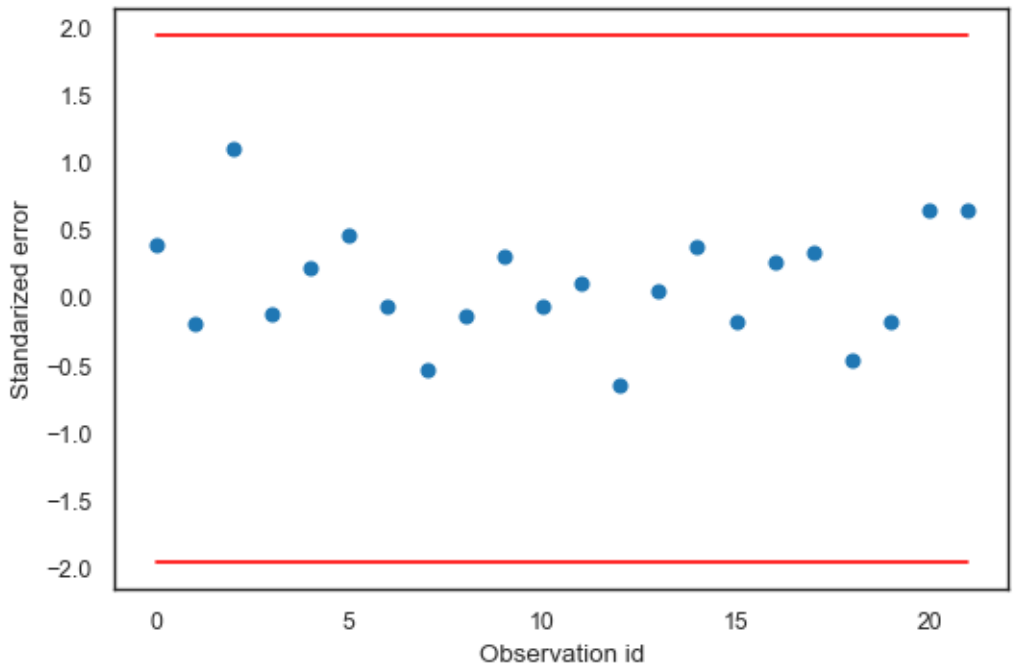
The observations vs predictions plot.

[35]:
```
fig, ax = plt.subplots(dpi=100)
yys = np.linspace(y_valid.min(), y_valid.max())
ax.plot(yys, yys, 'r')
ax.plot(y_valid, y_valid_p_mean, 'o')
ax.set_xlabel('Observations')
ax.set_ylabel('Predictions');
```



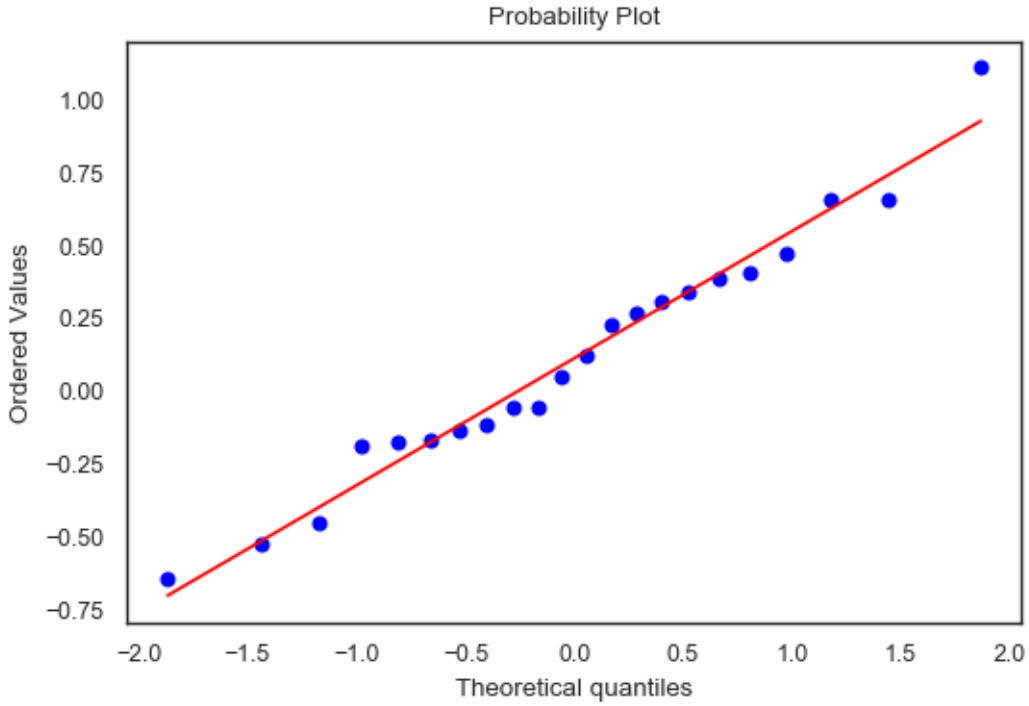
The standarized errors.

[36]:
```
std_errs = (y_valid_p_mean - y_valid) / y_valid_p_std
fig, ax = plt.subplots(dpi=100)
ax.plot(std_errs, 'o')
obs_ids = np.arange(std_errs.shape[0])
ax.plot(obs_ids, 1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.plot(obs_ids, -1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.set_xlabel('Observation id')
ax.set_ylabel('Standarized error');
```



The quantiles of the standarized errors.

[37]:
```
fig, ax = plt.subplots(dpi=100)
st.probplot(std_errs, dist=st.norm, plot=ax);
```



3.1.2 Subpart A.II

What is the noise variance you estimated for the Capacity?

[38]:
```
sigma = np.sqrt(1.0 / model.alpha_)
print('sigma2 = {0:1.2f}'.format(sigma ** 2))
```

```
sigma2 = 1106.59
```

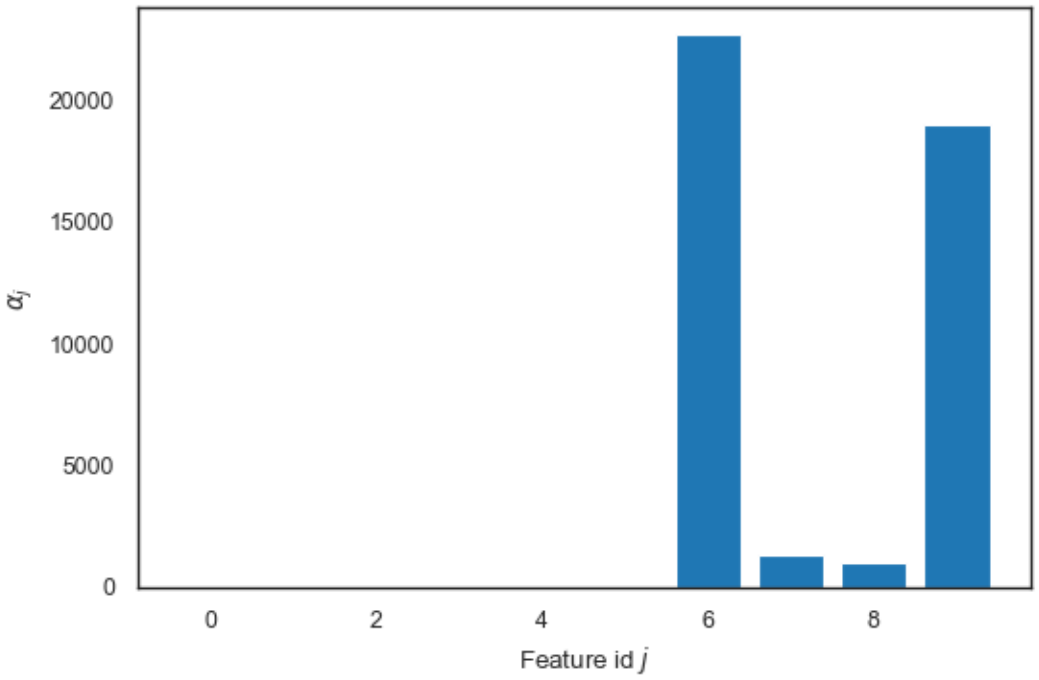
3.1.3 Subpart A.III

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

The feature with the largest α's in the prior are the least imporant. Here they are:

[39]:
```
alpha = model.lambda_
print(alpha)
fig, ax = plt.subplots(dpi=100)
ax.bar(np.arange(Phi_train.shape[1]), alpha)
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$');
```

```
[1.00000000e+00 3.08947974e-06 2.96175464e-05 2.55980764e-02
 1.35995132e-01 2.18607154e+01 2.26573094e+04 1.31045278e+03
 1.03096968e+03 1.89836923e+04]
```



3.1.4 Subpart B.I - Fit the Power

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

[40]:
```
# Here is how to extract the Capacity
y = data['Power'].values
y
```

[40]:
```
array([ 901,  881,  858, 1125, 1122, 1114, 1099, 1075, 1323, 1343, 1356,
       1361, 1354, 1335, 1484, 1534, 1576, 1606, 1624, 1628, 1615, 1600,
       1687, 1764, 1827, 1876, 1909, 1923, 1917, 1663, 1794, 1911, 2014,
       2101, 2169, 2217, 2243, 2246, 1663, 1844, 2010, 2159, 2290, 2400,
       2489, 2554, 2593, 1593, 1830, 2051, 2252, 2434, 2594, 2729, 2839,
       2922, 1442, 1743, 2025, 2286, 2525, 2740, 2929, 3091, 3223])
```

[41]:
```
# For polynomial features
from sklearn.preprocessing import PolynomialFeatures
# For splitting the data
from sklearn.model_selection import train_test_split
# Split the data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.33)
# Polynomials features
poly = PolynomialFeatures(degree=3)
# Design matrix for train data
Phi_train = poly.fit_transform(X_train)
# Fit with ARD
model = ARDRegression().fit(Phi_train, y_train)
```

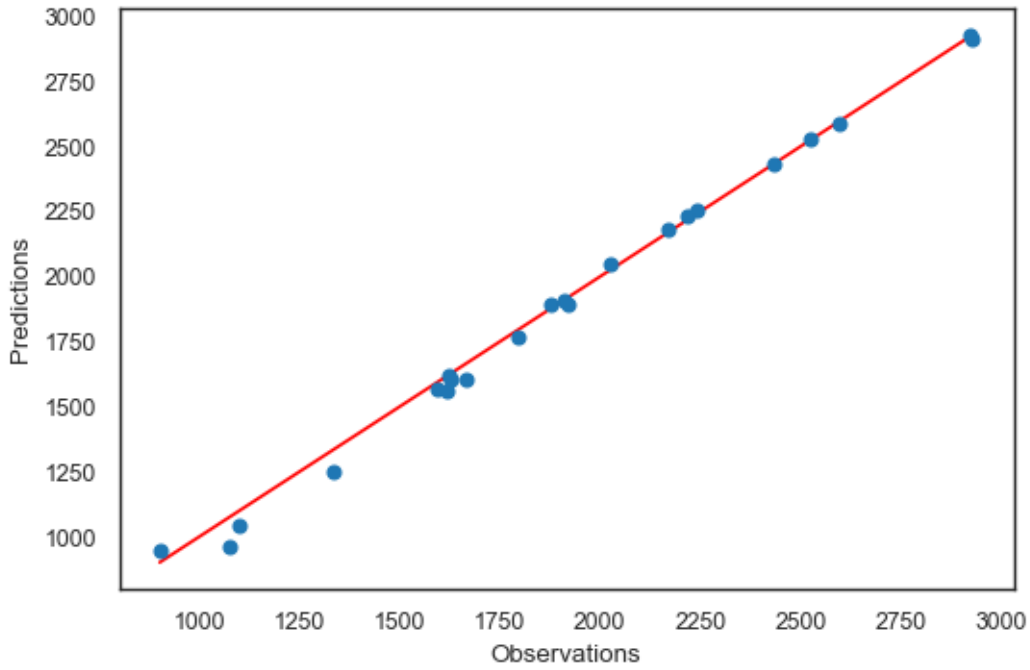
The mean square error:

[42]:
```
Phi_valid = poly.fit_transform(X_valid)

y_valid_p_mean, y_valid_p_std = model.predict(Phi_valid, return_std=True)
MSE = np.mean((y_valid_p_mean - y_valid) ** 2)
print('MSE = {0:1.2f}'.format(MSE))
```

```
MSE = 1671.75
```

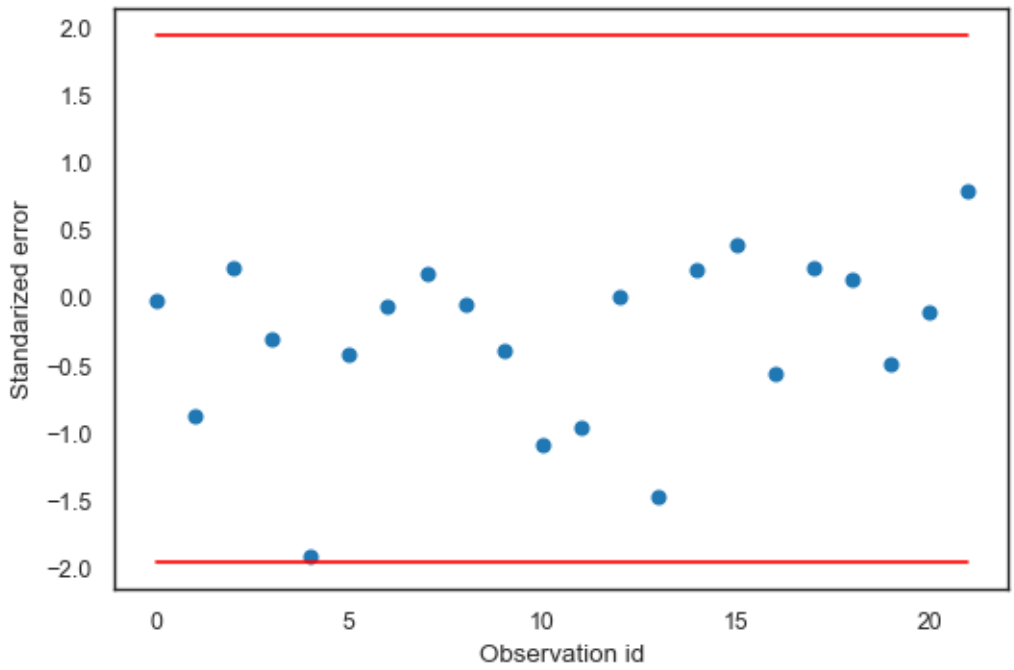
The observations vs predictions plot.

[43]:
```
fig, ax = plt.subplots(dpi=100)
yys = np.linspace(y_valid.min(), y_valid.max())
ax.plot(yys, yys, 'r')
ax.plot(y_valid, y_valid_p_mean, 'o')
ax.set_xlabel('Observations')
ax.set_ylabel('Predictions');
```



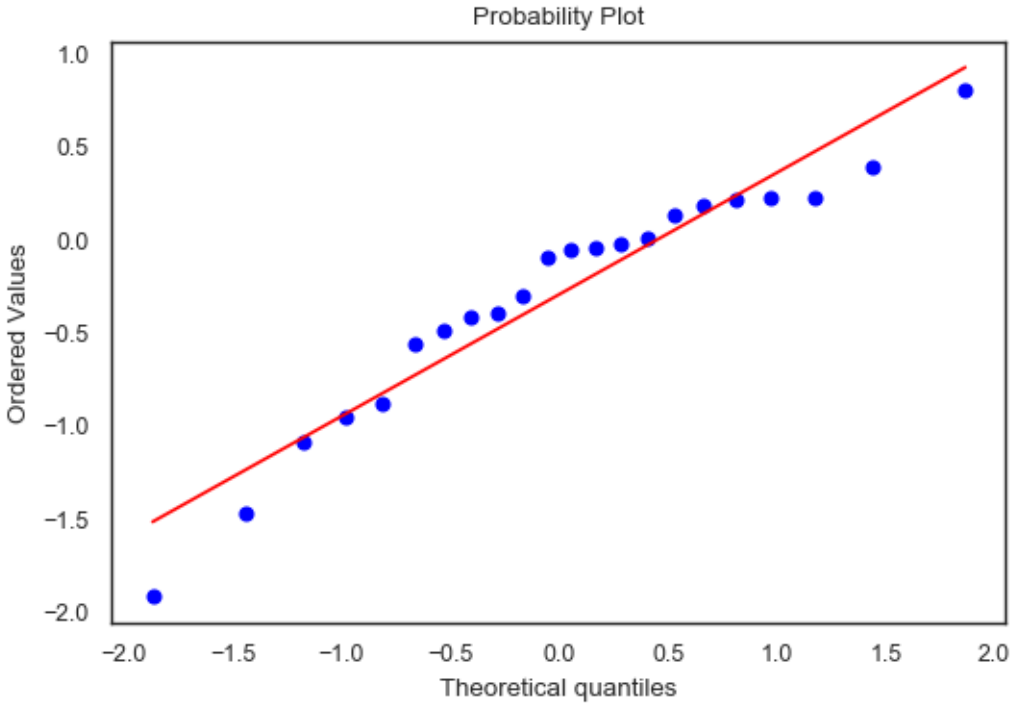
The standarized errors.

[44]:
```
std_errs = (y_valid_p_mean - y_valid) / y_valid_p_std
fig, ax = plt.subplots(dpi=100)
ax.plot(std_errs, 'o')
obs_ids = np.arange(std_errs.shape[0])
ax.plot(obs_ids, 1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.plot(obs_ids, -1.96 * np.ones(obs_ids.shape[0]), 'r')
ax.set_xlabel('Observation id')
ax.set_ylabel('Standarized error');
```



The quantiles of the standarized errors.

[45]:
```
fig, ax = plt.subplots(dpi=100)
st.probplot(std_errs, dist=st.norm, plot=ax);
```



3.1.5 Subpart B.II

What is the noise variance you estimated for the Power?

[46]:
```
sigma = np.sqrt(1.0 / model.alpha_)
print('sigma2 = {0:1.2f}'.format(sigma ** 2))
```

```
sigma2 = 554.32
```

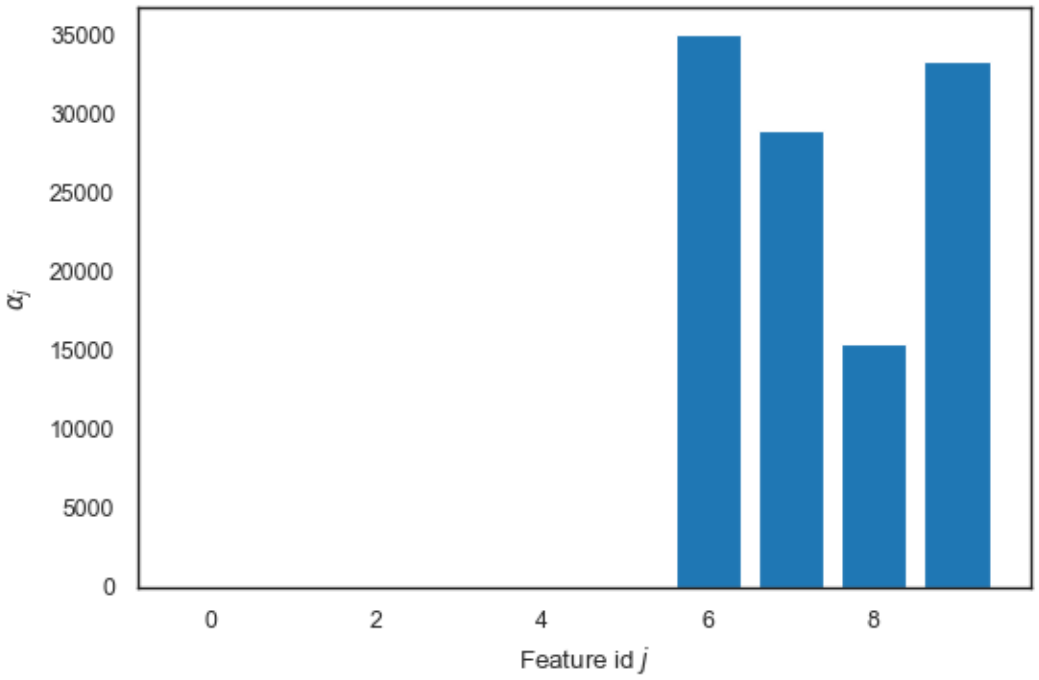
3.1.6 Subpart B.III

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

The feature with the largest α's in the prior are the least imporant. Here they are:

[47]:
```
alpha = model.lambda_
print(alpha)
fig, ax = plt.subplots(dpi=100)
ax.bar(np.arange(Phi_train.shape[1]), alpha)
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$');
```

```
[1.00000000e+00 4.20614137e-04 2.25023293e-04 1.33606180e+00
 3.12202542e-01 4.70041528e+00 3.50434951e+04 2.90136863e+04
 1.55207769e+04 3.33214859e+04]
```



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).

[48]:
```
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
↪master/homework/challenger_data.csv'
download(url)
```

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

[49]:
```
raw_data = pd.read_csv('challenger_data.csv')
raw_data
```

[49]:

	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:

[50]:
```
clean_data_0 = raw_data.dropna()
clean_data_0
```

[50]:
```
           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.

[51]:
```
clean_data = clean_data_0[:-1]
clean_data
```

[51]:

	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:

[52]:
```
x = clean_data['Temperature'].values
x
```

[52]:
```
array([66, 70, 69, 68, 67, 72, 73, 70, 57, 63, 70, 78, 67, 53, 67, 75, 70,
       81, 76, 79, 75, 76, 58])
```

[53]:
```
y = clean_data['Damage Incident'].values.astype(np.int)
y
```

[53]:
```
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.

Answer: This is one of the cases, where we don't have a lot of data. So we are going to use everything for training. To avoid overfitting, we will use the simplest possible model. The model

is:

$$p(y|x, w) = \text{sigm}(w_0 + w_1 x),$$

where w_0 and w_1 are parameters to be determined by data.

[54]:
```
from sklearn.linear_model import LogisticRegression
# A constant feature and the observed variable
X = np.hstack([np.ones((x.shape[0], 1)), x[:, None]])
# Fit the model
model = LogisticRegression(penalty='none', fit_intercept=False).fit(X, y)
```

This is it. Let's take a look at the parameters that that were found:

[55]:
```
model.coef_
```

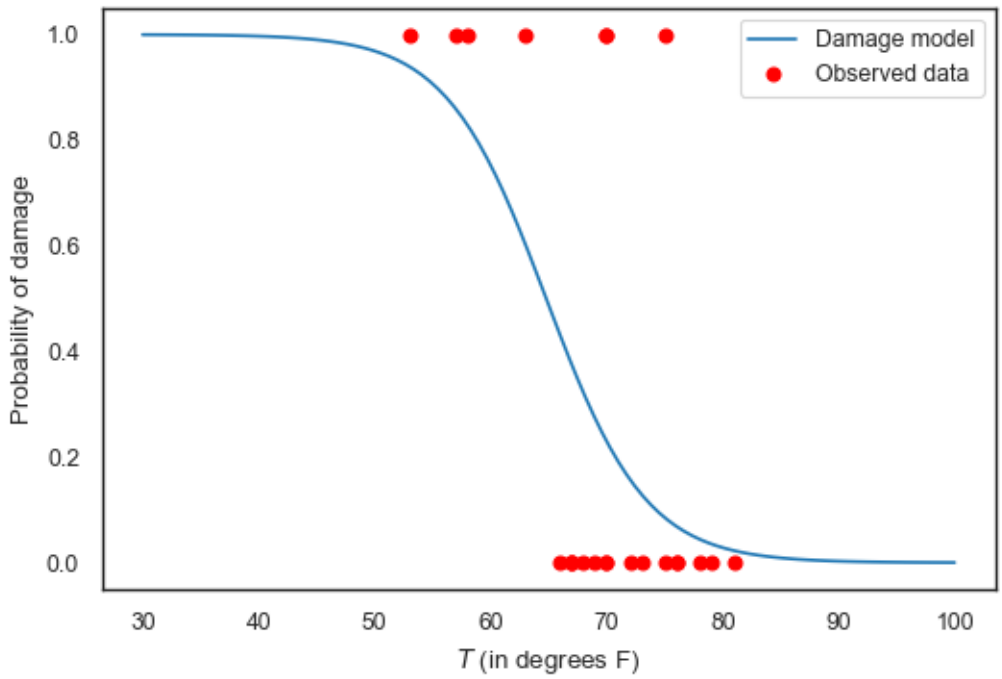
[55]:
```
array([[15.04290143, -0.23216274]])
```

We observe a negative correlation between temperature and damage. Damage becomes more probable as temperature decreases.

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

Plot the probability of damage as a function of temperature.

[56]:
```
fig, ax = plt.subplots(dpi=100)
# Some temperatures
xx = np.linspace(30, 100, 100)
# We need 2D arrays with the first column being one
XX = np.hstack([np.ones((xx.shape[0], 1)), xx[:, None]])
# Evaluate the probability of damage on the temperatures
yy = model.predict_proba(XX)[:, 1]
# Plot the probability of damage
ax.plot(xx, yy, label='Damage model')
ax.plot(x, y, 'ro', label='Observed data')
ax.set_xlabel('$T$ (in degrees F)')
ax.set_ylabel('Probability of damage')
plt.legend(loc='best');
```



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.

Answer:

First, let's start by finding the probability that we have damage when the temperature is 31 degrees F.

[57]:
```
pdam = model.predict_proba([[1, 31]])[0, 1]
print('Prob of damage at 31 degrees F: {0:1.2f}%'.format(pdam * 100))
```

```
Prob of damage at 31 degrees F: 99.96%
```

So, it is kind of ridiculous to launch at 31 degrees F. Nevertheless, let's go over the formal decision analysis and plot our decision function.

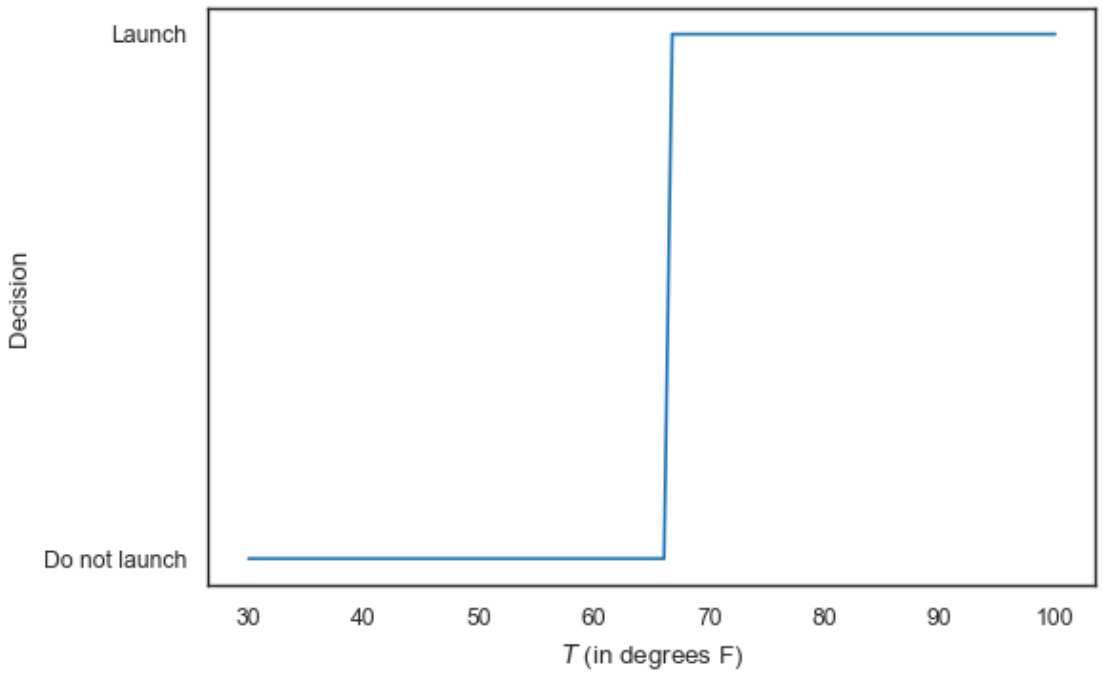
[58]:
```
# Here are some estimates of various possible costs
cost_of_human_life = 7 * 1e6
number_of_crew_members = 7
cost_of_mission = 450 * 1e6
cost_of_destroying_shuttle = 50 * 1e9
political_cost = 20 * 1e9
# Let's use them to put together our cost matrix
```

```
# Cost of not launching when there is no damage
c00 = political_cost
# Cost of not launching when there is damage
c01 = political_cost
# Cost of launching when there is no damage
c10 = cost_of_mission
# Cost of launching when there is damage
c11 = cost_of_mission + cost_of_destroying_shuttle\
      + number_of_crew_members * cost_of_human_life
# Put his in a matrix
cost_matrix = np.array([[c00, c01],
                        [c10, c11]])
print(cost_matrix)
```

```
[[2.0000e+10 2.0000e+10]
 [4.5000e+08 5.0499e+10]]
```

This is of course made up, but probably in the right order of magnitude. The units are dollars.

[59]:
```
# Plot of decision boundary
fig, ax = plt.subplots(dpi=100)
probs = model.predict_proba(XX)
exp_cost = np.einsum('ij,kj->ki', cost_matrix, probs)
decision_idx = np.argmin(exp_cost, axis=1)
ax.plot(xx, decision_idx)
ax.set_yticks([0, 1])
ax.set_yticklabels(['Do not launch', 'Launch'])
ax.set_ylabel('Decision')
ax.set_xlabel('$T$ (in degrees F)');
```



The interesting thing is that unless the political cost is in the tens of billion dollars, then you should probably not launch at all.