Homework 4

References

- Lectures 13-16 (inclusive).

Instructions

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

In [166…

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

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

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

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

Student details

- **First Name:** Ben

- **Last Name:** McAteer
- **Email:** bmcateer@purdue.edu

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

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

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

In [167…

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

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

In [168…

```
data =  np.loadtxt('stress_strain.txt')
```

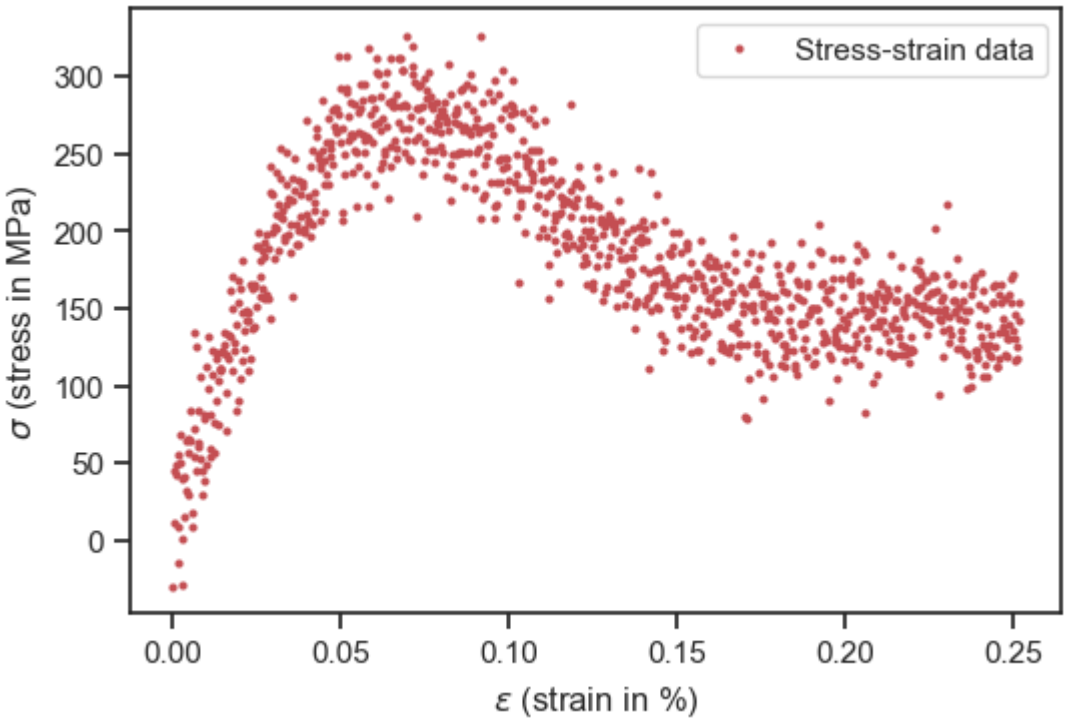
The dataset was generated using a molecular dynamics simulation of a plastic material (thanks to Professor Alejandro Strachan for sharing the data!). Specifically, Strachan's group did the following:

- 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:

In [169…

```
# Strain
x = data[:, 0]
# Stress in MPa
y = data[:, 1]

plt.figure()
plt.plot(
    x,
    y,
    'ro',
    markersize=2,
    label='Stress-strain data'
)
```

```
plt.xlabel('$\epsilon$ (strain in %)')
plt.ylabel('$\sigma$ (stress in MPa)')
plt.legend(loc='best');
```



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

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.

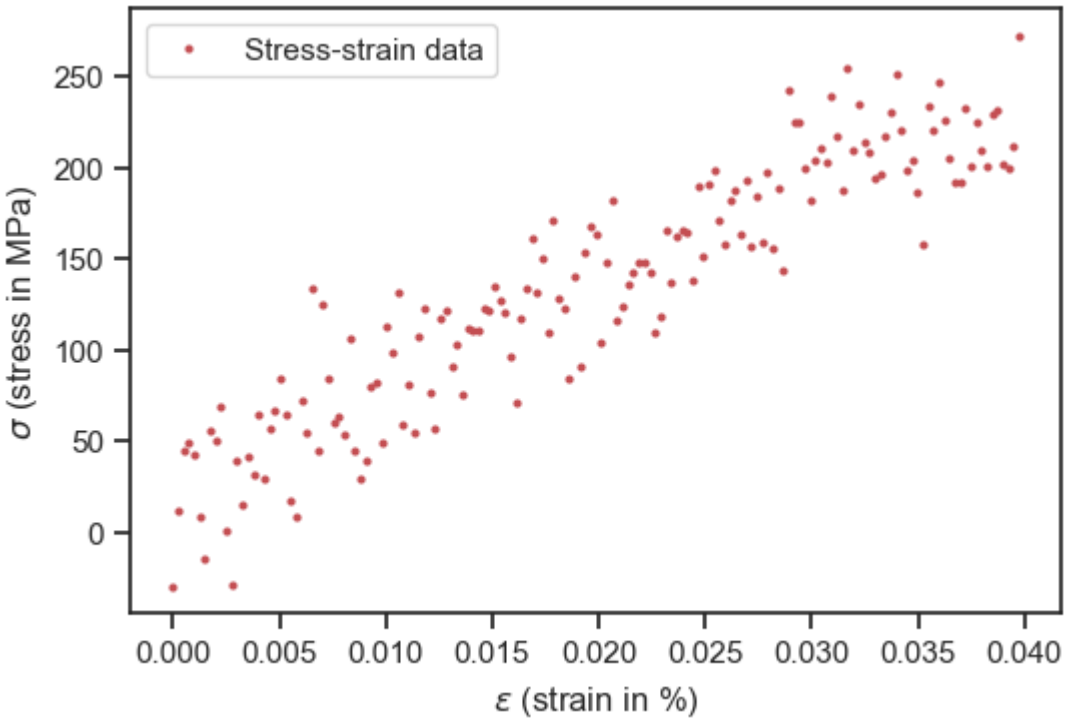
Subpart A.I

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

In [170…

```
# The point at which the stress-strain curve stops being linear
```

```
epsilon_l = 0.04
# Relevant data (this is nice way to get the linear part of the stresses and straints)
x_rel = x[x < 0.04]
y_rel = y[x < 0.04]

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



Split your data into training and validation.

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

In [171…

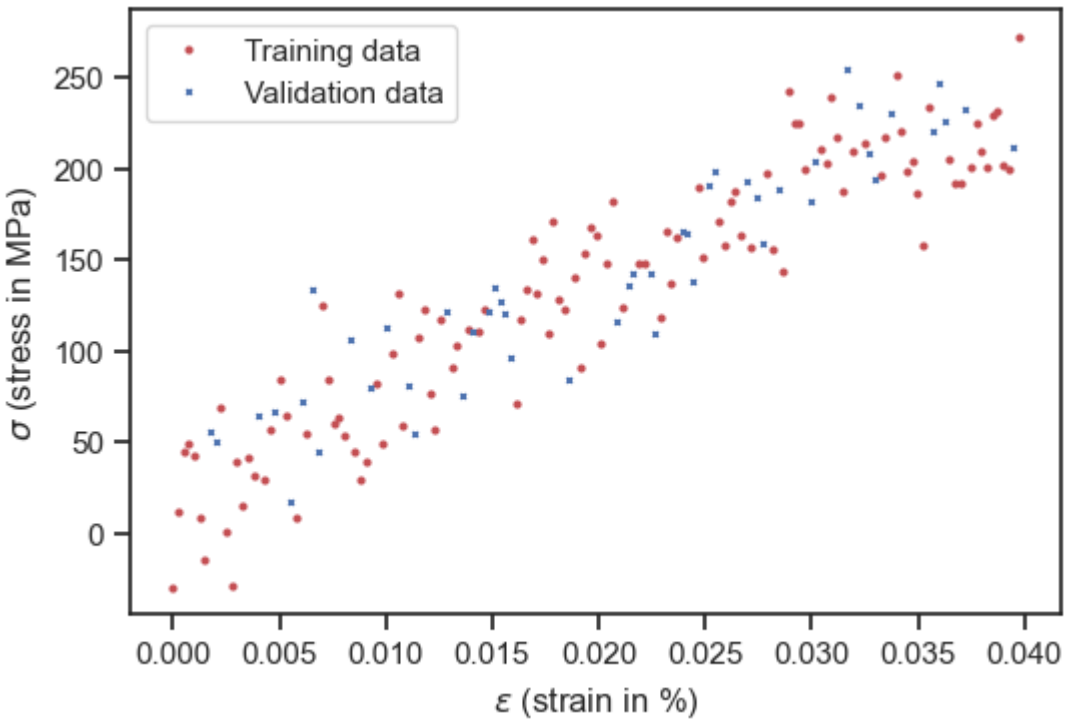
```
from sklearn.model_selection import train_test_split#,cross_val_score,GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn import linear_model
from sklearn.metrics import classification_report,accuracy_score, f1_score

x_train, x_valid, y_train, y_valid = train_test_split(x_rel, y_rel,train_size = .70, te
```

Use the following to visualize your split:

In [172…

```
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');
```



Subpart A.II

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

In [173…

```
#Subpart A.II

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

    Arguments:
    x      -- A 2D array with only one column.
    degree -- An integer greater than zero.
    """
    assert isinstance(x, np.ndarray), 'x is not a numpy array.'
    assert x.ndim == 2, 'You must make x a 2D array.'
    assert x.shape[1] == 1, 'x must be a column.'
```

```
    cols = []
    for i in range(degree+1):
        cols.append(x ** i)
    return np.hstack(cols)

def get_fourier_design_matrix(x, L, num_terms):
    """Fourier expansion with ``num_terms`` cosines and sines.

    Arguments:
    x          --  A 2D array with only one column.
    L          --  The "length" of the domain.
    num_terms  --  How many Fourier terms do you want.
                   This is not the number of basis
                   functions you get. The number of basis functions
                   is 1 + num_terms / 2. The first one is a constant.
    """
    assert isinstance(x, np.ndarray), 'x is not a numpy array.'
    assert x.ndim == 2, 'You must make x a 2D array.'
    assert x.shape[1] == 1, 'x must be a column.'
    N = x.shape[0]
    cols = [np.ones((N, 1))]
    for i in range(int(num_terms / 2)):
        cols.append(np.cos(2 * (i+1) * np.pi / L * x))
        cols.append(np.sin(2 * (i+1) * np.pi / L * x))
    return np.hstack(cols)

def get_rbf_design_matrix(x, x_centers, ell):
    """Radial basis functions design matrix.

    Arguments:
    x        --  The input points on which you want to evaluate the
                 design matrix.
    x_center --  The centers of the radial basis functions.
    ell      --  The lengthscale of the radial basis function.
    """
    assert isinstance(x, np.ndarray), 'x is not a numpy array.'
    assert x.ndim == 2, 'You must make x a 2D array.'
    assert x.shape[1] == 1, 'x must be a column.'
    N = x.shape[0]
    cols = [np.ones((N, 1))]
    for i in range(x_centers.shape[0]):
        cols.append(np.exp(-(x - x_centers[i]) ** 2 / ell))
    return np.hstack(cols)

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

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

```
                the design matrix.

    Keyword Arguments:
    phi_func_args -- Any arguments passed to the
                     function that calculates the
                     design matrix.
    num_samples   -- The number of samples to take.
    y_true        -- The true response for plotting.
    nugget        -- A small number to add the covariance
                     if it is not positive definite
                     (numerically).
    """
    Phi_xx = phi_func(
        xx[:, None],
        *phi_func_args
    )
    m = model.coef_
    S = model.sigma_
    w_post = st.multivariate_normal(
        mean=m,
        cov=S + nugget * np.eye(S.shape[0])
    )
    fig, ax = plt.subplots()
    for _ in range(num_samples):
        w_sample = w_post.rvs()
        yy_sample = Phi_xx @ w_sample
        ax.plot(xx, yy_sample, 'r', lw=0.5)
    ax.plot([], [], "r", lw=0.5, label="Posterior samples")
    ax.plot(x, y, 'kx', label='Observed data')
    ax.plot(xx, yy_true, label='True response surface')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    plt.legend(loc="best");

from sklearn.linear_model import ARDRegression
from sklearn.linear_model import BayesianRidge
# Parameters
degree = 3

# Design matrix
Phi = np.hstack([np.zeros((len(x_train),0)), x_train.reshape(len(x_train),1)])

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

Phi_valid = np.hstack([np.zeros((len(x_valid),0)), x_valid.reshape(len(x_valid),1)])

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



alpha = np.sqrt(model.lambda_) # or alpha = np.sqrt(1/model.lambda_)
print('Subpart A.II: ')
print(f'alpha = {alpha}')
sigma = np.sqrt(1 / model.alpha_)
```

```
print(f'sigma = {sigma}')
# Noisevariance = sigma ** 2
# print(f'Noisevariance = {Noisevariance}')
m = model.coef_
S = model.sigma_
# m, S = find_m_and_S(Phi, y_train, sigma, alpha)
print(f"Posterior mean w: {m}")
print(f"Posterior covariance w:")
print(S)
```

```
Subpart A.II:
alpha = 0.0001569448113955057
sigma = 29.8598429260212
Posterior mean w: [6370.483]
Posterior covariance w:
[[15157.209]]
```

Subpart A.III

Calculate the mean square error of the validation data.

In [174…

```
#part A.III
# SM = np.linalg.lstsq(Phi_valid, y_valid,rcond = None) #need to reshape still

# y_validpred= (Phi_valid[:,0])  @ SM;

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

```
MSE = 844.0214103715383
```

Subpart A.IV

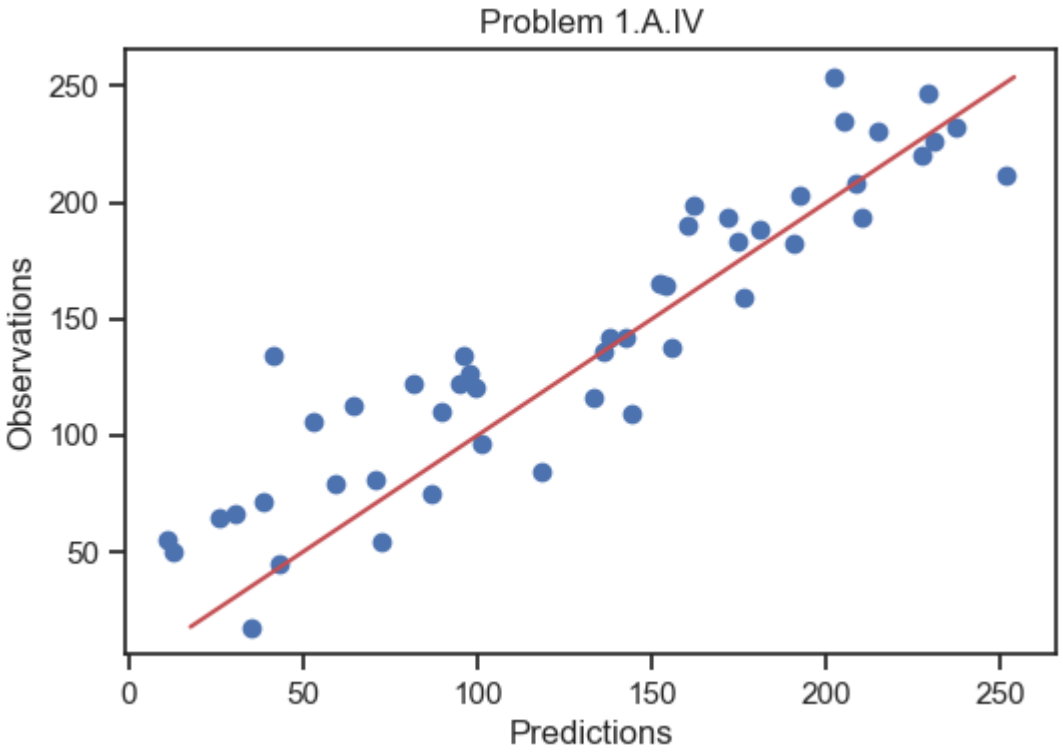
Make the observations vs predictions plot for the validation data.

In [175…

```
#part A.IV
fig, ax = plt.subplots()
ax.plot(y_predict, y_valid, 'o')
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
ax.plot(yys, yys, 'r-')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations');
plt.title('Problem 1.A.IV')
```

Out[175…

```
Text(0.5, 1.0, 'Problem 1.A.IV')
```

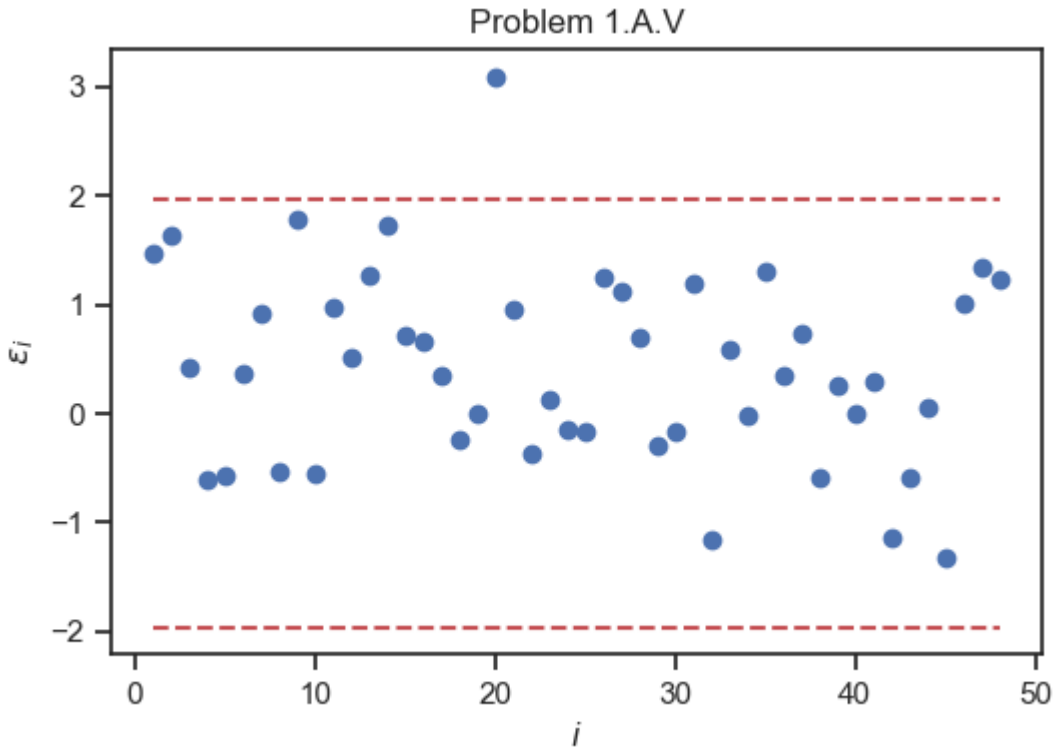


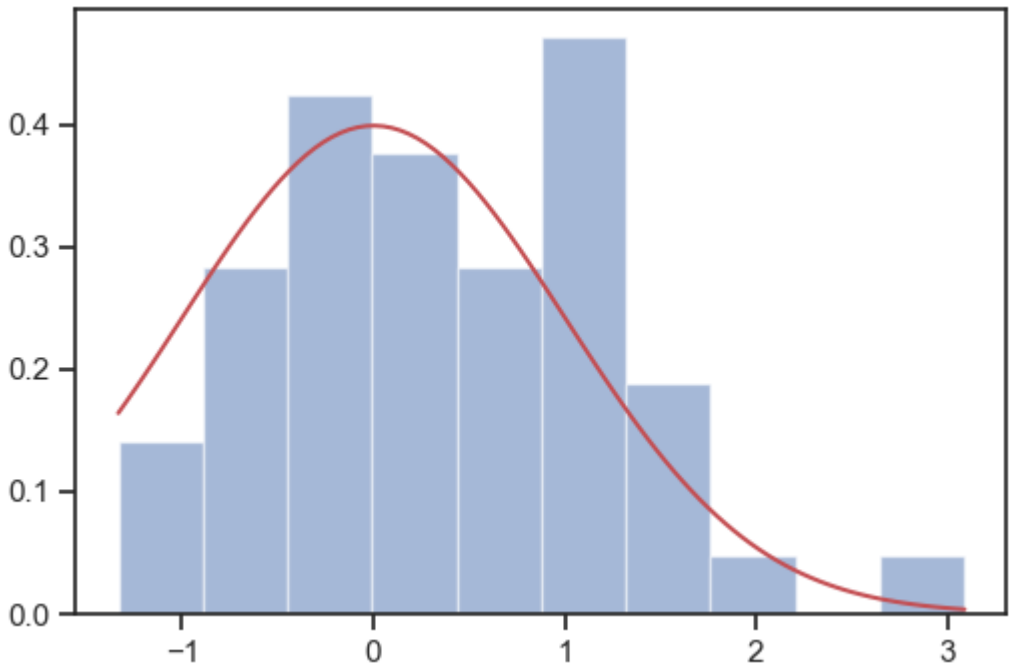
Subpart A.V

Compute and plot the standarized errors for the validation data.

In [176…

```
#part V

eps = (y_valid - y_predict) / y_std
idx = np.arange(1,eps.shape[0] + 1)

fig, ax = plt.subplots()
ax.plot(idx, eps, 'o', label='Standarized errors')
ax.plot(idx, 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(idx, -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$');
plt.title('Problem 1.A.V')



fig, ax = plt.subplots()
ax.hist(eps, alpha=0.5, density=True)
ee = np.linspace(eps.min(), eps.max(), 100)
ax.plot(ee, st.norm.pdf(ee), 'r');
```





Subpart A.VI

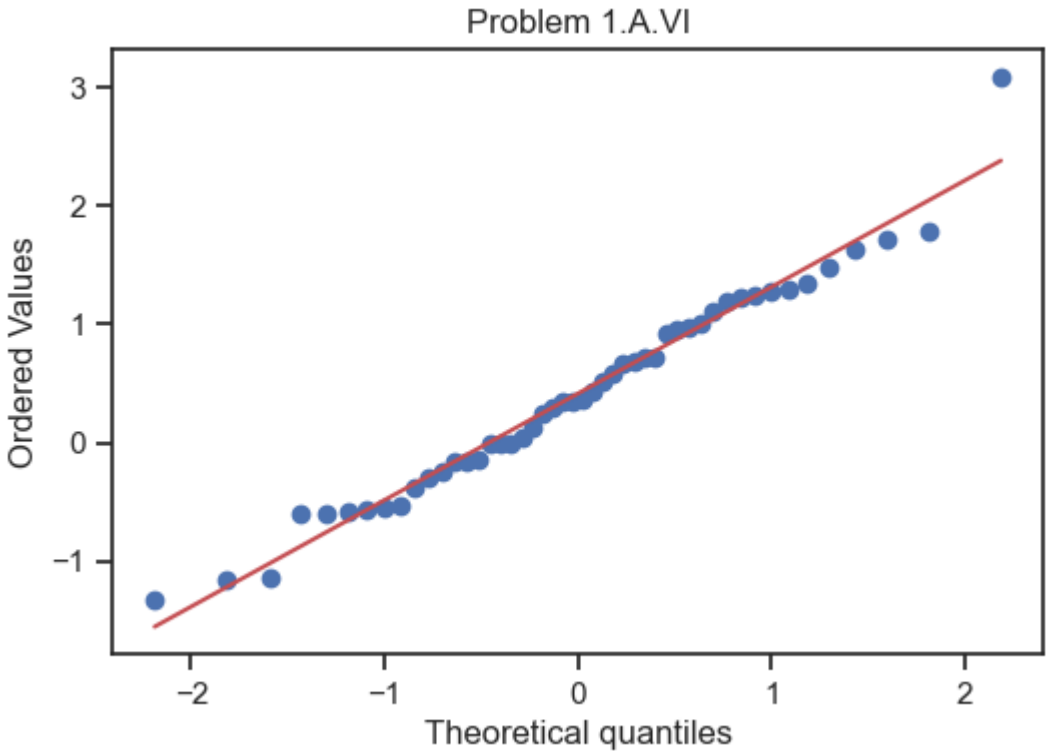
Make the quantile-quantile plot of the standarized errors.

In [177…

```
#part VI
fig, ax = plt.subplots()
st.probplot(eps, dist=st.norm, plot=ax);
plt.title('Problem 1.A.VI')
```

Out[177…

```
Text(0.5, 1.0, 'Problem 1.A.VI')
```



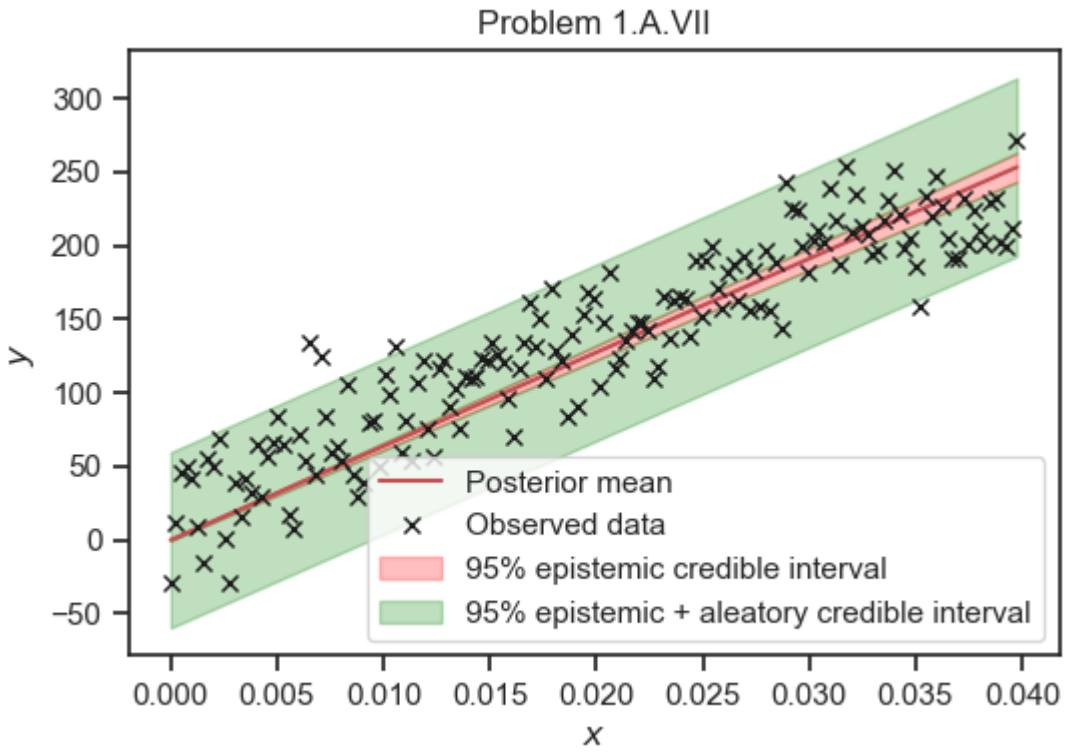
Subpart A.VII

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

In [178…

```
#part VII
"""Plot the posterior predictive separating
aleatory and espitemic uncertainty.

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

Keyword Arguments:
phi_func_args -- Any arguments passed to the
                 function that calculates the
                 design matrix.
y_true        -- The true response for plotting.
"""

x_predict = np.zeros((len(x_rel),1))
for i in range(len(x_rel)):
    x_predict[i] = x_rel[i]

y_relpredict, y_relstd = model.predict(x_predict,return_std = True)

yy_std = np.sqrt(y_relstd ** 2 - sigma**2)
yy_le = y_relpredict - 2.0 * yy_std
yy_ue = y_relpredict + 2.0 * yy_std
yy_lae = y_relpredict - 2.0 * y_relstd
```

```
yy_uae = y_relpredict + 2.0 * y_relstd

fig, ax = plt.subplots()
ax.plot(x_rel, y_relpredict, 'r', label="Posterior mean")
ax.fill_between(
    x_rel,
    yy_le,
    yy_ue,
    color='red',
    alpha=0.25,
    label="95% epistemic credible interval"
)
ax.fill_between(
    x_rel,
    yy_lae,
    yy_le,
    color='green',
    alpha=0.25
)
ax.fill_between(
    x_rel,
    yy_ue,
    yy_uae,
    color='green',
    alpha=0.25,
    label="95% epistemic + aleatory credible interval"
)
ax.plot(x_rel, y_rel, 'kx', label='Observed data')
plt.title('Problem 1.A.VII')
#if y_rel is not None:
#    ax.plot(x_rel, y_rel, "--", label="True response")
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc="best");
```



Subpart A. VIII

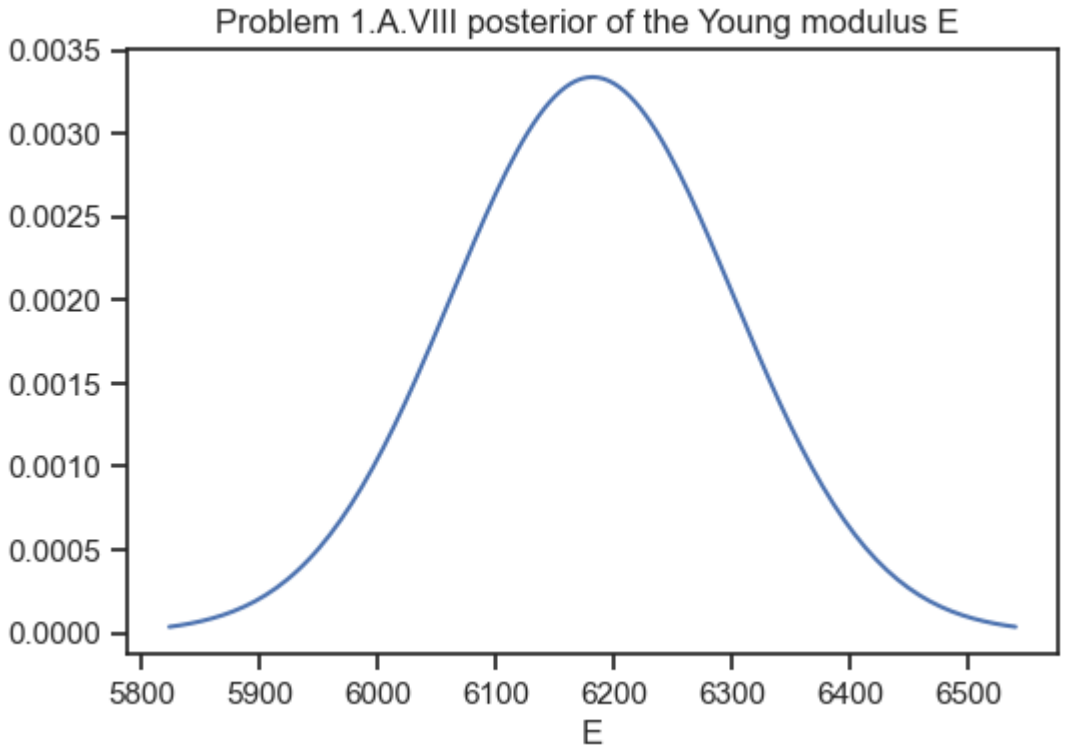
Visualize the posterior of the Young modulus E conditioned on the data.

In [179…

```
#part VIII

xx = np.linspace(np.min(x_rel[np.nonzero(x_rel)]), max(x_rel),159);
EMean = np.mean(y_relpredict/xx)
Estd = np.mean(yy_std / xx)

Eposterior = st.norm(EMean,Estd)
Eaxis = np.linspace(EMean-3*Estd,EMean+3*Estd,159)
Epdf = Eposterior.pdf(Eaxis)

plt.figure()
plt.plot(Eaxis,Epdf)
plt.title('Problem 1.A.VIII posterior of the Young modulus E ')
plt.xlabel('E')
plt.ylabel('')
```

Out[179…

```
Text(0, 0.5, '')
```



Subpart A.IX

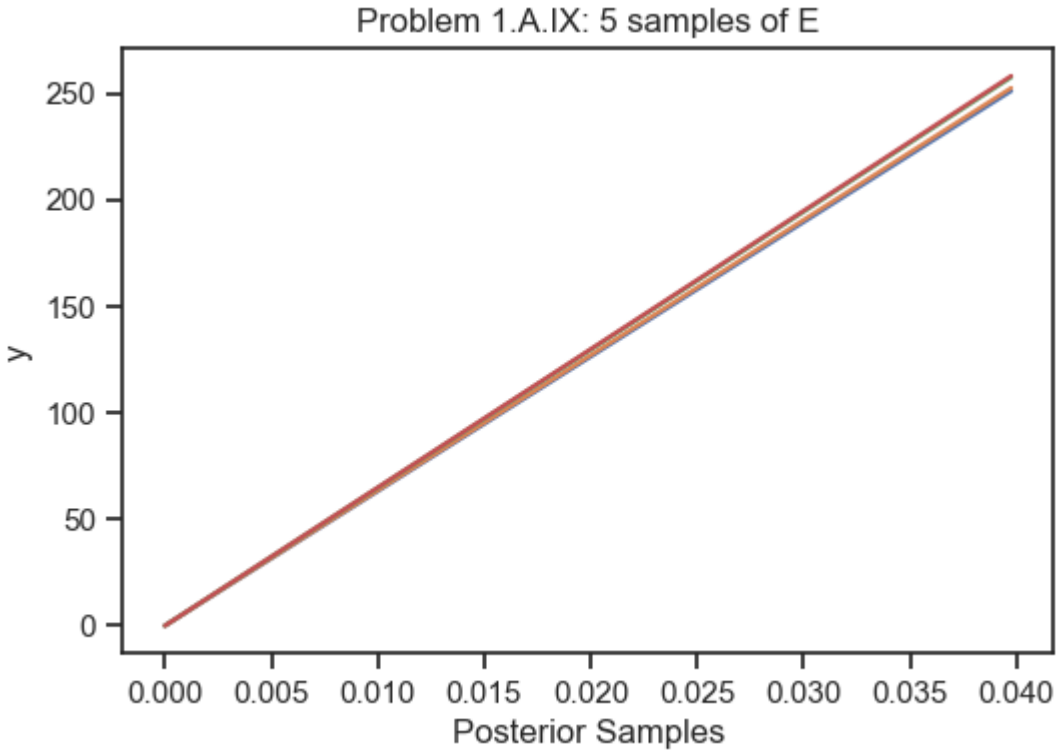
Take five samples of stress-strain curve in the elastic regime and visualize them.

In [180…

```
#part IX
w_post = st.multivariate_normal(mean=m, cov= S *np.eye(S.shape[0]))

#Phi_xx = get_polynomial_design_matrix(xx[:,None],degree)

plt.figure()
```

```
for i in range(0,4):
    waxis = w_post.rvs()
    y_post = x_rel * waxis
    plt.plot(x_rel,y_post)
plt.title('Problem 1.A.IX: 5 samples of E')
plt.ylabel('y')
plt.xlabel('Posterior Samples')
```

Out[180… `Text(0.5, 0, 'Posterior Samples')`



Subpart A.X

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

In [181…

```
#part X
E_low = Eposterior.ppf(0.025)
E_high = Eposterior.ppf(0.975)

print(f'Problem 1.A.X: the 95% credible interval of Youngs Modulus is: [{E_low:0.2f},{E
```

```
Problem 1.A.X: the 95% credible interval of Youngs Modulus is: [5947.89,6416.22]
```

Subpart A.XI

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

In [182…

```
#part XI

print(f'Problem 1.A.XI: I would pick an average Youngs modulus like {EMean:0.2f} to rep
```

```
Problem 1.A.XI: I would pick an average Youngs modulus like 6182.05 to represent the pop
```

```
ulation of plastic. This would ensure that other calculations involving youngs modulus b
est represent the plastic. A higher Youngs modulus could be used to assume the plastic i
s more brittle if stretching plastics is a concern however.
```

See above print statement

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.

Subpart B.I - Extract training and validation data

Extract training and validation data from the entire dataset.

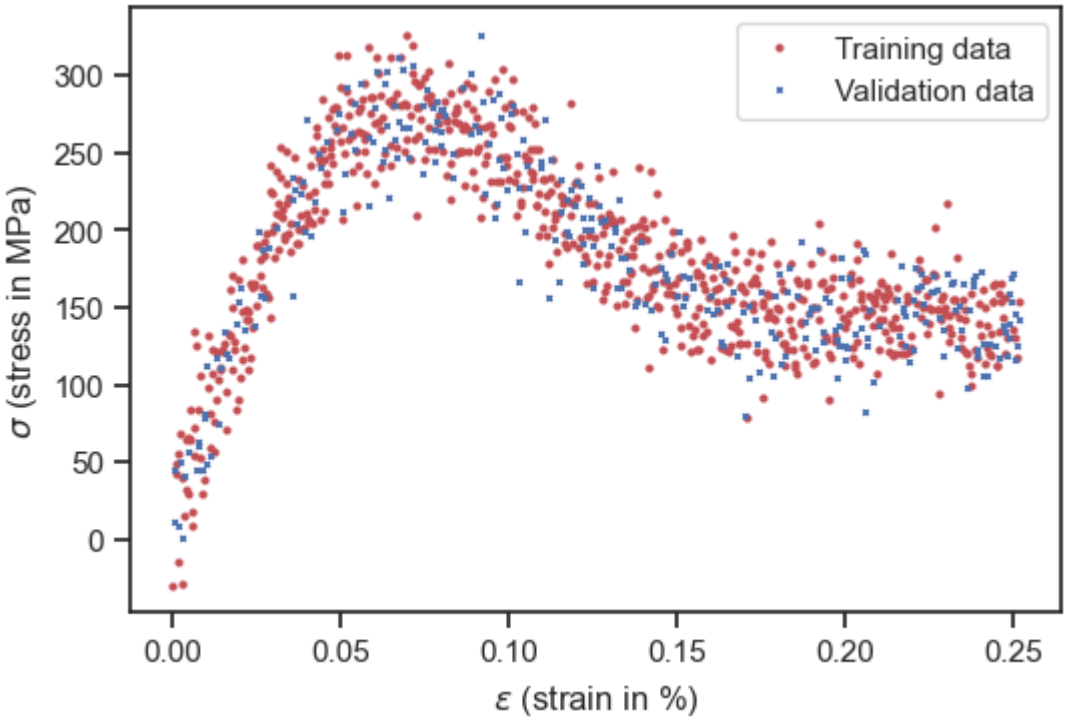
In [183…

```
#B.I
x_train, x_valid, y_train, y_valid = train_test_split(x, y,train_size = .70, test_size=
```

Use the following to visualize your split:

In [184…

```
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');
```



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:

In [185…

```
#Part B.II
def compute_design_matrix(Epsilon, epsilon_l, d):
    """Compute the design matrix for the stress-strain curve problem.

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

    Returns:
        A design matrix N x d
```

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

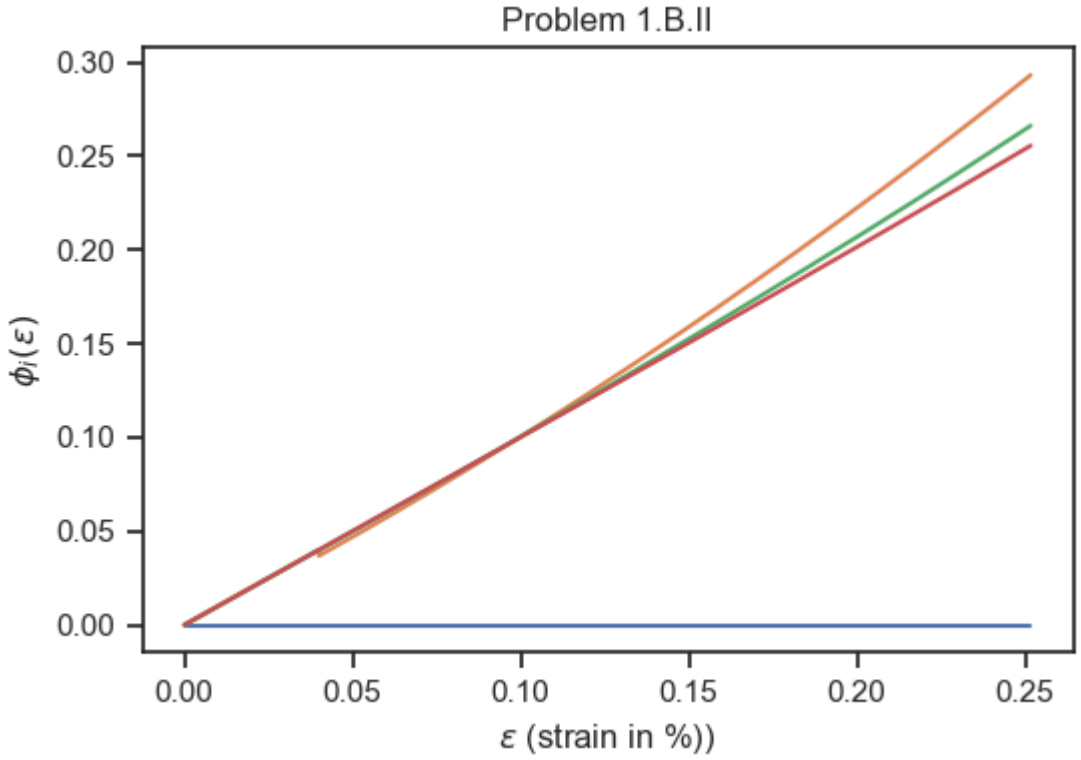
Visualize the basis functions here:

In [186…

```
d = 4
eps = np.linspace(0, x.max(), len(x))
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('$\phi_i(\epsilon)$');
plt.title('Problem 1.B.II')
```

Out[186…

```
Text(0.5, 1.0, 'Problem 1.B.II')
```



Subpart B.III

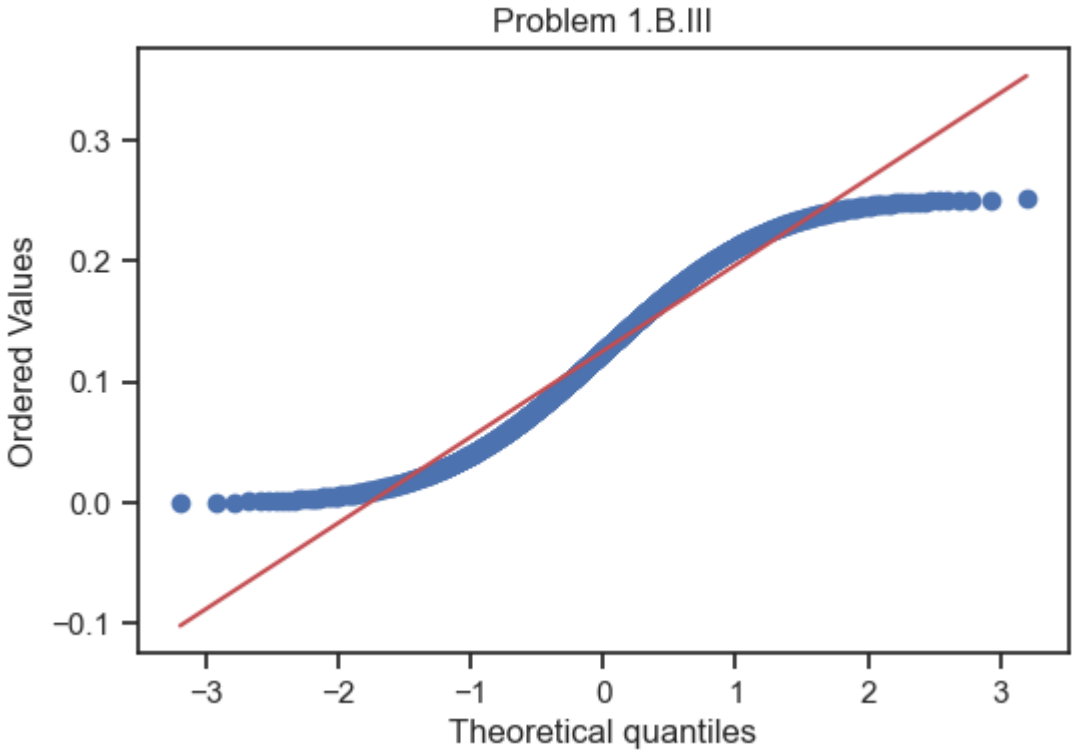
Fit the model using automatic relevance determination and demonstrate that it works well by doing all the things we did above (MSE, observations vs predictions plot, standarized errors, etc.).

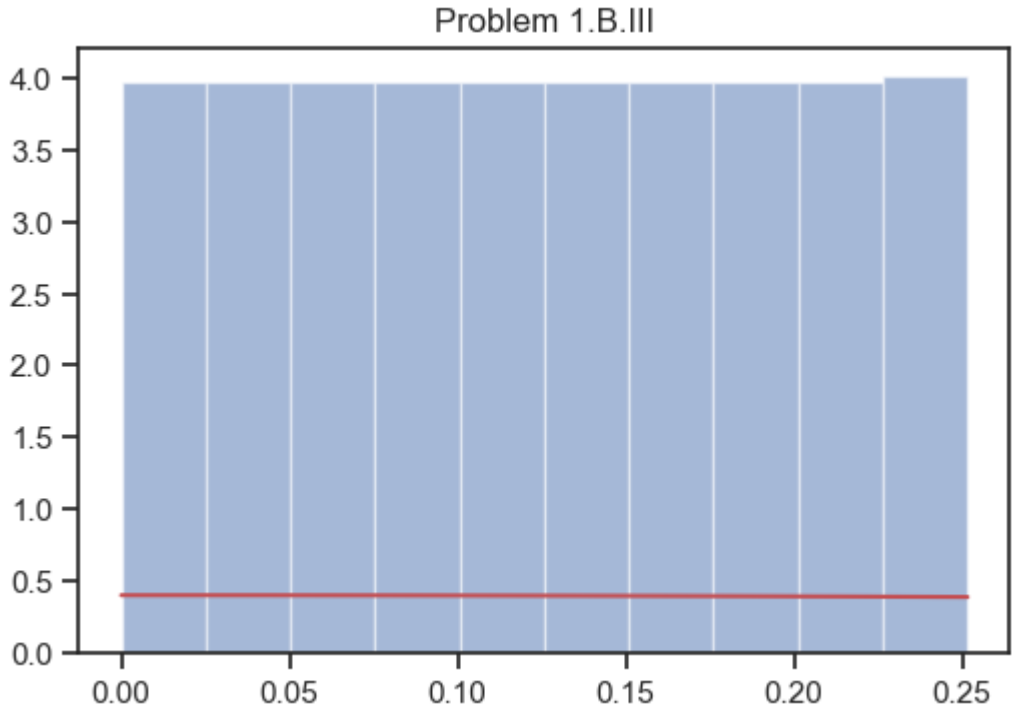
In [187…

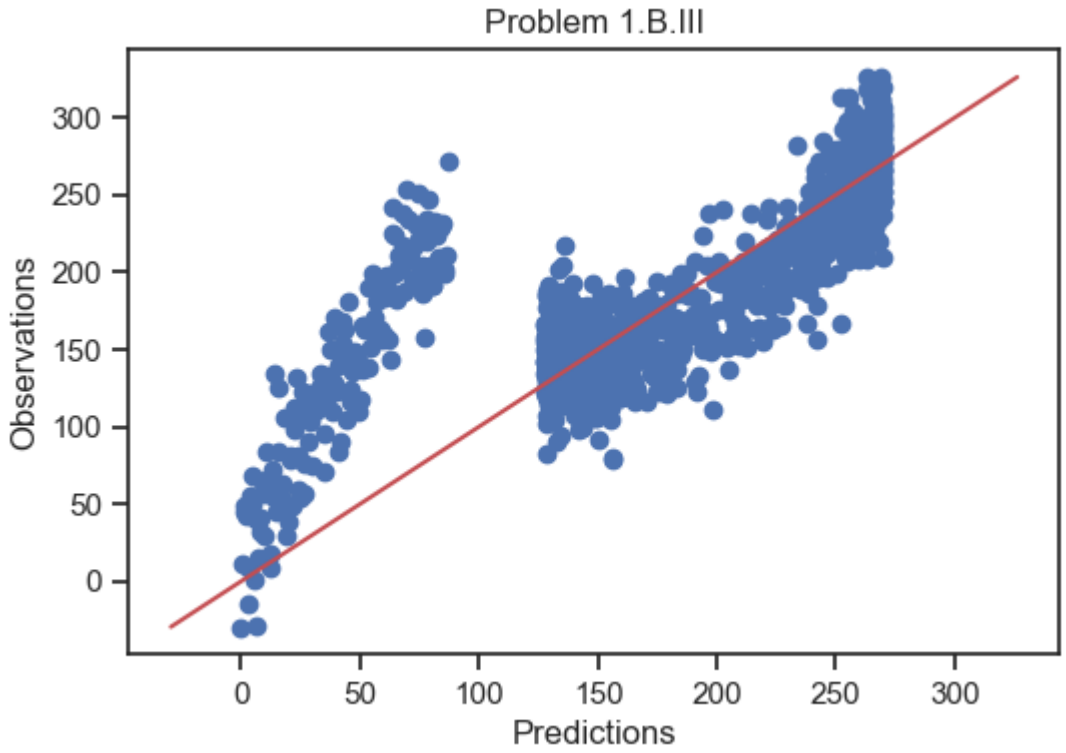
```
#subpart B.III

modelB = ARDRegression(fit_intercept=False,threshold_lambda=np.inf).fit(Phis,y)



alpha = modelB.lambda_
sigma = np.sqrt(1 / modelB.alpha_)
S = modelB.sigma_


B_y_predict, B_y_std = modelB.predict(Phis,return_std = True)
B_y_MSE  = np.mean((B_y_predict - y)**2)

print('Subpart B.III: ')
print(f'alpha = {alpha}')
print(f'sigma = {sigma}')
#print(f"Posterior mean w: {m}")
print(f"Posterior covariance w:")
print(S)
print(f'MSE = {B_y_MSE:0.2f}')

B_x_predict = np.zeros((len(x),4))
for i in range(len(x)):
    B_x_predict[i,:] = x[i]

Byy_std = np.sqrt(B_y_std ** 2 - sigma**2)

Byy_le = B_y_predict - 2.0 * Byy_std
Byy_ue =B_y_predict + 2.0 * Byy_std
Byy_lae =B_y_predict - 2.0 * B_y_std
Byy_uae = B_y_predict + 2.0 * B_y_std

fig, ax = plt.subplots()
st.probplot(eps, dist=st.norm, plot=ax);
plt.title('Problem 1.B.III')



fig, ax = plt.subplots()
ax.hist(eps, alpha=0.5, density=True)
ee = np.linspace(eps.min(), eps.max(), 100)
ax.plot(ee, st.norm.pdf(ee), 'r');
plt.title('Problem 1.B.III')

fig, ax = plt.subplots()
ax.plot(B_y_predict, y, 'o')
yys = np.linspace(
    y.min(),
    y.max(),
    100)
ax.plot(yys, yys, 'r-')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations');
plt.title('Problem 1.B.III')
```

```
Subpart B.III:
alpha = [1.000e+00 2.798e-10 3.285e-11 7.899e-11]
sigma = 48.021268827255575
Posterior covariance w:
[[ 1.000e+00  0.000e+00  0.000e+00  0.000e+00]
 [ 0.000e+00  1.967e+06 -7.493e+06  5.531e+06]
 [ 0.000e+00 -7.493e+06  2.929e+07 -2.183e+07]
 [ 0.000e+00  5.531e+06 -2.183e+07  1.633e+07]]
MSE = 2299.14
Text(0.5, 1.0, 'Problem 1.B.III')
```

Out[187…







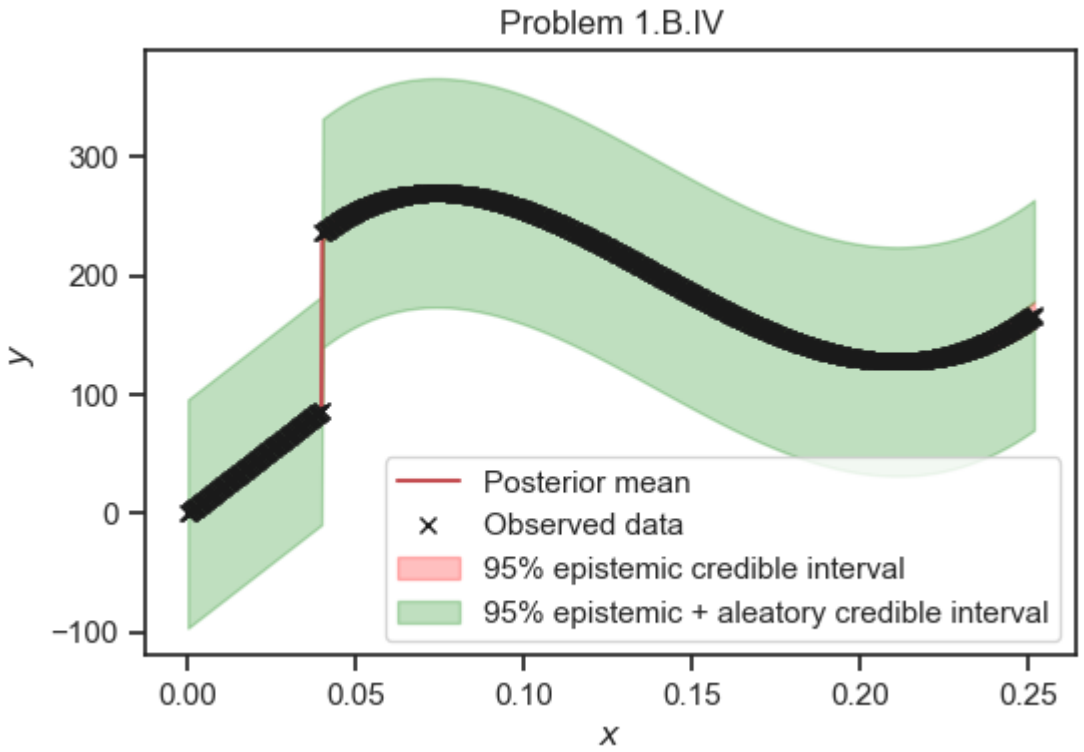
Subpart B.IV

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

In [188…

```
#Problem 1.b.IV
fig, ax = plt.subplots()
ax.plot(B_x_predict[:,0], B_y_predict, 'r', label="Posterior mean")
ax.fill_between(
    x,
    Byy_le,
    Byy_ue,
    color='red',
    alpha=0.25,
    label="95% epistemic credible interval"
)
ax.fill_between(
    x,
    Byy_lae,
    Byy_le,
    color='green',
    alpha=0.25
)
ax.fill_between(
    x,
    Byy_ue,
    Byy_uae,
    color='green',
    alpha=0.25,
    label="95% epistemic + aleatory credible interval"
)
ax.plot(x, B_y_predict, 'kx', label='Observed data')
plt.title('Problem 1.B.IV')#
#!if y_rel is not None:
```

```
#    ax.plot(x, y, "--", label="True response")
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc="best");
```



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.

In [189…

```
#part B.V
```

```
Strength = y/x
SortedStrength = np.argsort(Strength)
SortedStrengtharr = Strength[SortedStrength]

MaxStrength = SortedStrengtharr[-25 : ]
print(MaxStrength)
```

```
[ 9547.428  9564.323  9584.003 10336.456 11227.797 11463.613 11643.824
 11894.859 12139.048 12462.576 12473.449 12776.233 12999.054 13847.618
 16020.075 16770.666 17761.736 20473.904 24908.987 30233.133 31422.634
 42005.886 46809.688 65561.082 89798.154]
C:\Users\Ben\AppData\Local\Temp/ipykernel_19272/1243126251.py:3: RuntimeWarning: divide
by zero encountered in true_divide
  Strength = y/x
```

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:

In [190…

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

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

In [191…

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

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.

In [192…

```
# 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


T_e = data['T_e'].values
T_c = data['T_c'].values
x
```

Out[192…

```
array([[-30,  25],
       [-30,  30],
       [-30,  35],
       [-25,  25],
       [-25,  30],
       [-25,  35],
       [-25,  40],
       [-25,  45],
```

```
       [-20,  25],
       [-20,  30],
       [-20,  35],
       [-20,  40],
       [-20,  45],
       [-20,  50],
       [-15,  25],
       [-15,  30],
       [-15,  35],
       [-15,  40],
       [-15,  45],
       [-15,  50],
       [-15,  55],
       [-10,  25],
       [-10,  30],
       [-10,  35],
       [-10,  40],
       [-10,  45],
       [-10,  50],
       [-10,  55],
       [-10,  60],
       [ -5,  25],
       [ -5,  30],
       [ -5,  35],
       [ -5,  40],
       [ -5,  45],
       [ -5,  50],
       [ -5,  55],
       [ -5,  60],
       [ -5,  65],
       [  0,  25],
       [  0,  30],
       [  0,  35],
       [  0,  40],
       [  0,  45],
       [  0,  50],
       [  0,  55],
       [  0,  60],
       [  0,  65],
       [  5,  25],
       [  5,  30],
       [  5,  35],
       [  5,  40],
       [  5,  45],
       [  5,  50],
       [  5,  55],
       [  5,  60],
       [  5,  65],
       [ 10,  25],
       [ 10,  30],
       [ 10,  35],
       [ 10,  40],
       [ 10,  45],
       [ 10,  50],
       [ 10,  55],
       [ 10,  60],
       [ 10,  65]], dtype=int64)
```

In [193…

```
# Here is how to extract the Capacity
```

```
y = data['Capacity'].values
y
```

Out[193…
```
array([ 1557,  1201,   892,  2509,  2098,  1726,  1398,  1112,  3684,
        3206,  2762,  2354,  1981,  1647,  5100,  4547,  4019,  3520,
        3050,  2612,  2206,  6777,  6137,  5516,  4915,  4338,  3784,
        3256,  2755,  8734,  7996,  7271,  6559,  5863,  5184,  4524,
        3883,  3264, 10989, 10144,  9304,  8471,  7646,  6831,  6027,
        5237,  4461, 13562, 12599, 11633, 10668,  9704,  8743,  7786,
        6835,  5891, 16472, 15380, 14279, 13171, 12057, 10939,  9819,
        8697,  7575], dtype=int64)
```

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

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

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.

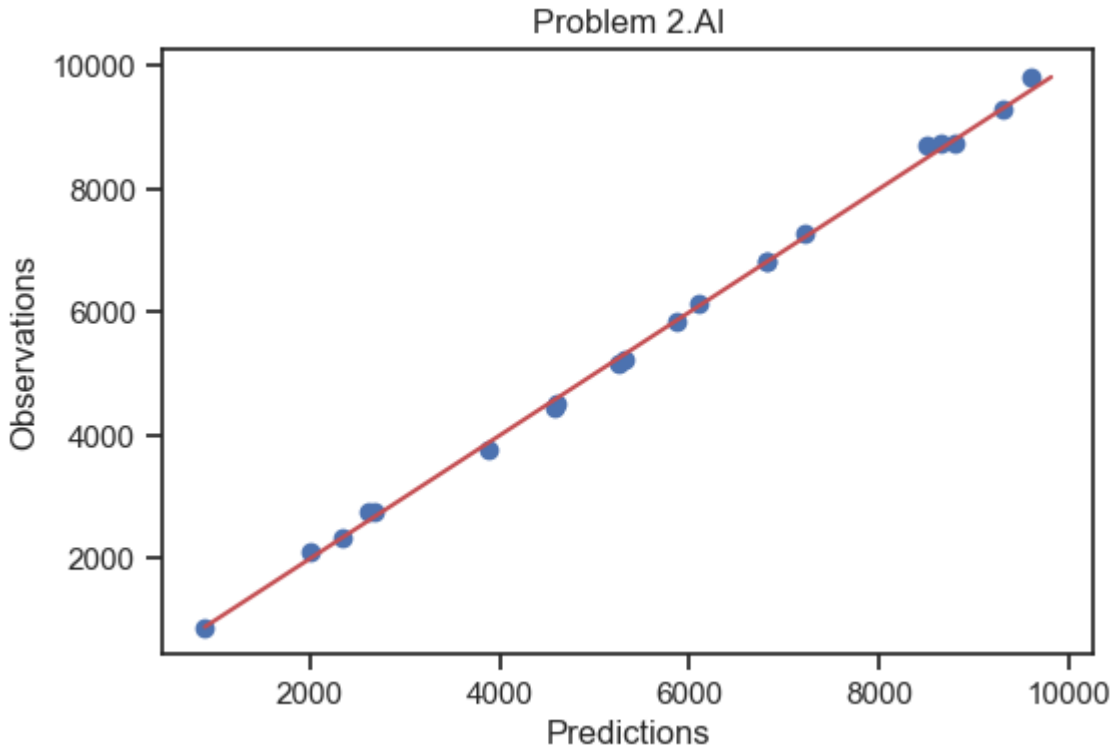
Subpart A.I - Fit the capacity

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

In [194…
```
from sklearn.linear_model import ARDRegression
from sklearn.linear_model import BayesianRidge
from sklearn.model_selection import train_test_split#,cross_val_score,GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn import linear_model
from sklearn.metrics import classification_report,accuracy_score, f1_score
from sklearn.preprocessing import PolynomialFeatures

#part 2.A.I
x_train, x_valid, y_train, y_valid = train_test_split(x, y,train_size = .70, test_size=


degree = 4

polyf = PolynomialFeatures(degree)
Phi = polyf.fit_transform(x_train)

Model2 = ARDRegression(threshold_lambda = np.inf).fit(Phi,y_train)

Phi_valid = polyf.fit_transform(x_valid)

# W_polyf, _, _, _ = np.linalg.lstsq(Phi_valid, y_valid, rcond=None)
# y_predict = Phi_valid @ W_polyf
y_predict, y_std_meas = Model2.predict(Phi_valid, return_std = True)
```

```
alpha = Model2.lambda_
sigma = np.sqrt(1 / Model2.alpha_)
S = Model2.sigma_
MSE = np.mean((y_predict - y_valid)**2)
print('Subpart A.I: ')
print(f'alpha = {alpha}')
print(f'sigma = {sigma}')
#print(f"Posterior mean w: {m}")
# print(f"Posterior covariance w:")
# print(S)
print(f'MSE = {MSE:0.2f}')

fig, ax = plt.subplots()
ax.plot(y_predict, y_valid, 'o')
yys = np.linspace(
    y_valid.min(),
    y_valid.max(),
    100)
ax.plot(yys, yys, 'r-')
ax.set_xlabel('Predictions');
ax.set_ylabel('Observations');
plt.title('Problem 2.AI');
```

```
Subpart A.I:
alpha = [1.000e+00 1.070e-01 1.028e-02 1.578e-01 6.252e-04 2.515e-02 8.549e+03
 4.332e+00 9.023e-01 1.021e+02 4.091e+05 2.767e+05 2.125e+04 1.610e+04
 4.370e+05]
sigma = 87.29885695511797
MSE = 8887.39
```



Subpart A.II

What is the noise variance you estimated for the Capacity?

In [195…

```
#part 2.A.II

print(f'Prolbem 2.A.II: The estimated Noise Variance for Capacity is: {sigma:0.3}')
```

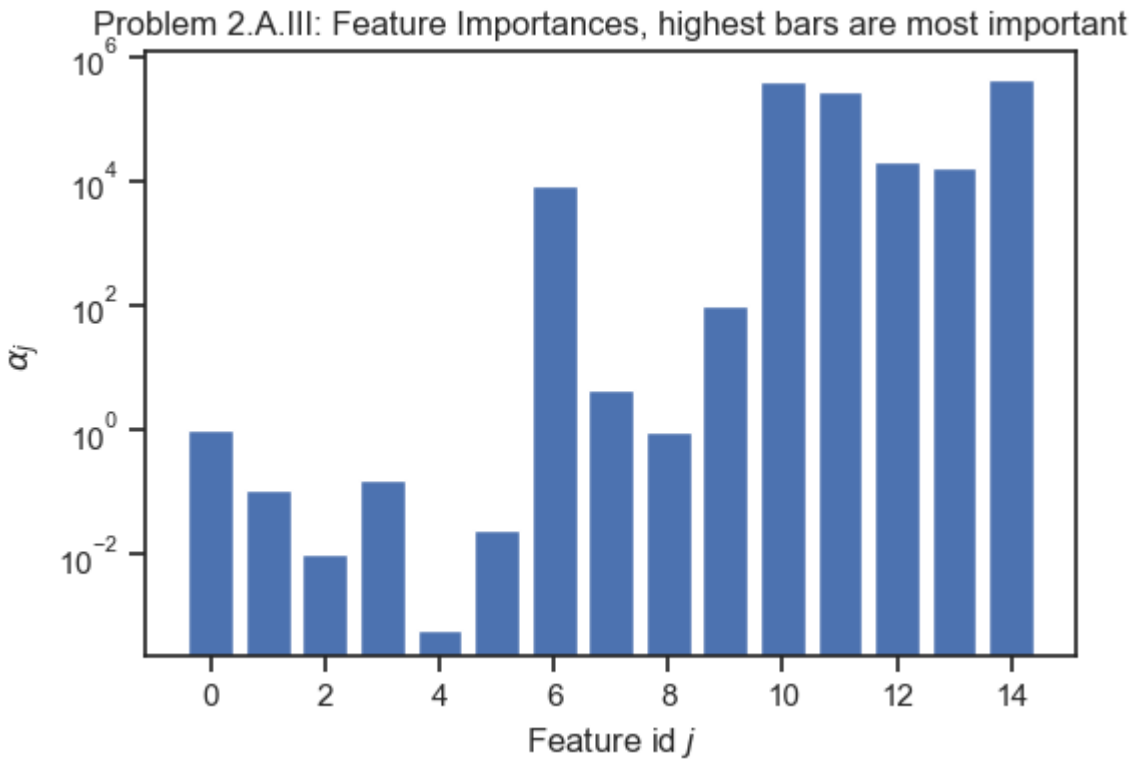
```
Prolbem 2.A.II: The estimated Noise Variance for Capacity is: 87.3
```

Subpart A.III

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

In [196…

```
#part 2.A.III


fig, ax = plt.subplots()
ax.bar(np.arange(15), alpha)
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$')
ax.set_yscale("log")
ax.set_title(f'Problem 2.A.III: Feature Importances, highest bars are most important');
```



Subpart B.I - Fit the Power

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

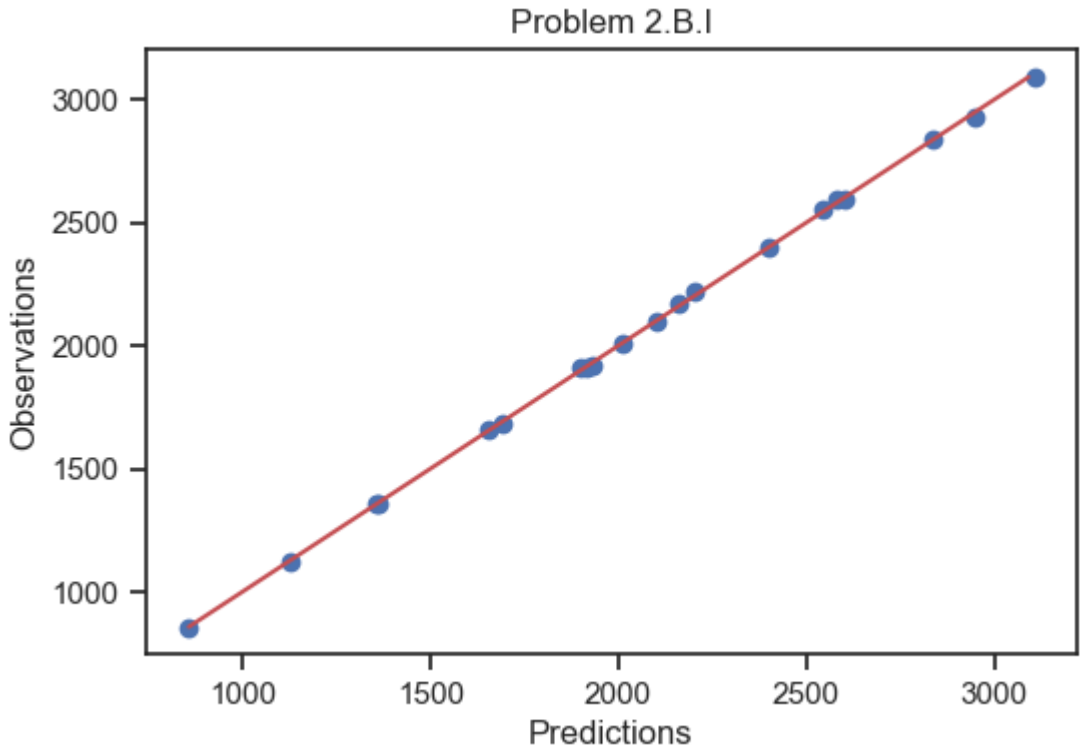
In [197…

```
#part 2.B.I
z  = data['Power'].values

x_trainB, x_validB, y_trainB, y_validB = train_test_split(x, z ,train_size = .70, test_
```

```
degree = 4

polyB = PolynomialFeatures(degree)
PhiB = polyB.fit_transform(x_trainB)

Model2B = ARDRegression(threshold_lambda = np.inf).fit(PhiB,y_trainB)

Phi_validB = polyB.fit_transform(x_validB)

y_predictB, y_std_measB = Model2B.predict(Phi_validB, return_std = True)

alphaB = Model2B.lambda_
sigmaB = np.sqrt(1 / Model2B.alpha_)
SB = Model2B.sigma_
MSEB = np.mean((y_predictB - y_validB)**2)
print('Subpart B.I: ')
print(f'alpha = {alphaB}')
print(f'sigma = {sigmaB}')
#print(f"Posterior mean w: {m}")
# print(f"Posterior covariance w:")
# print(S)
print(f'MSE = {MSEB:0.2f}')

fig, ax = plt.subplots()
ax.plot(y_predictB, y_validB, 'o')
yysB = np.linspace(
    y_validB.min(),
    y_validB.max(),
    100)
ax.plot(yysB, yysB, 'r-')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations');
plt.title('Problem 2.B.I')
```

```
Subpart B.I:
alpha = [1.000e+00 2.006e-03 5.948e+00 1.453e+00 3.446e-01 3.592e-01 5.640e+04
 5.887e+02 1.140e+02 1.086e+03 4.983e+05 4.764e+05 3.742e+05 3.867e+05
 4.938e+05]
sigma = 8.612456480366365
MSE = 84.91
```

Out[197… Text(0.5, 1.0, 'Problem 2.B.I')



Subpart B.II

What is the noise variance you estimated for the Power?

In [209…

```
#part 2.B.II

print(f'Problem 2.B.II: The estimated Noise Variance for Power is: {sigmaB:0.3}')
```

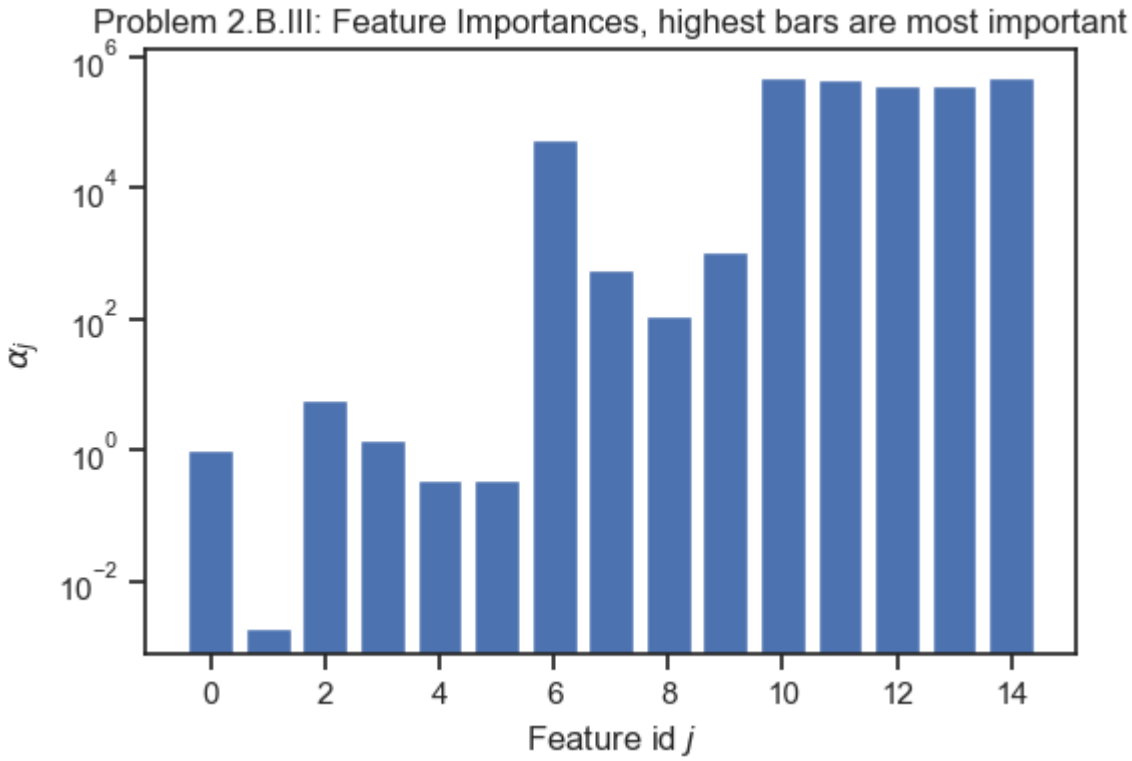
```
Problem 2.B.II: The estimated Noise Variance for Power is: 8.61
```

Subpart B.III

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

In [199…

```
#part 2.B.III


fig, ax = plt.subplots()
ax.bar(np.arange(15), alphaB)
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$')
ax.set_yscale("log")
ax.set_title(f'Problem 2.B.III: Feature Importances, highest bars are most important');
```



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).

In [200…
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecturebook
download(url)
```

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

In [201…
```
raw_data = pd.read_csv('challenger_data.csv')
```

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:

In [202…
```
clean_data_0 = raw_data.dropna()
```

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

In [203…
```
clean_data = clean_data_0[:-1]
```

Let's extract the features and the labels:

In [204…
```
x = clean_data['Temperature'].values
```

In [205…
```
y = clean_data['Damage Incident'].values.astype(np.float)
```

```
C:\Users\Ben\AppData\Local\Temp/ipykernel_19272/4164561558.py:1: DeprecationWarning: `n
p.float` is a deprecated alias for the builtin `float`. To silence this warning, use `fl
oat` by itself. Doing this will not modify any behavior and is safe. If you specifically
wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/relea
se/1.20.0-notes.html#deprecations
  y = clean_data['Damage Incident'].values.astype(np.float)
```

Part A - Perform logistic regression

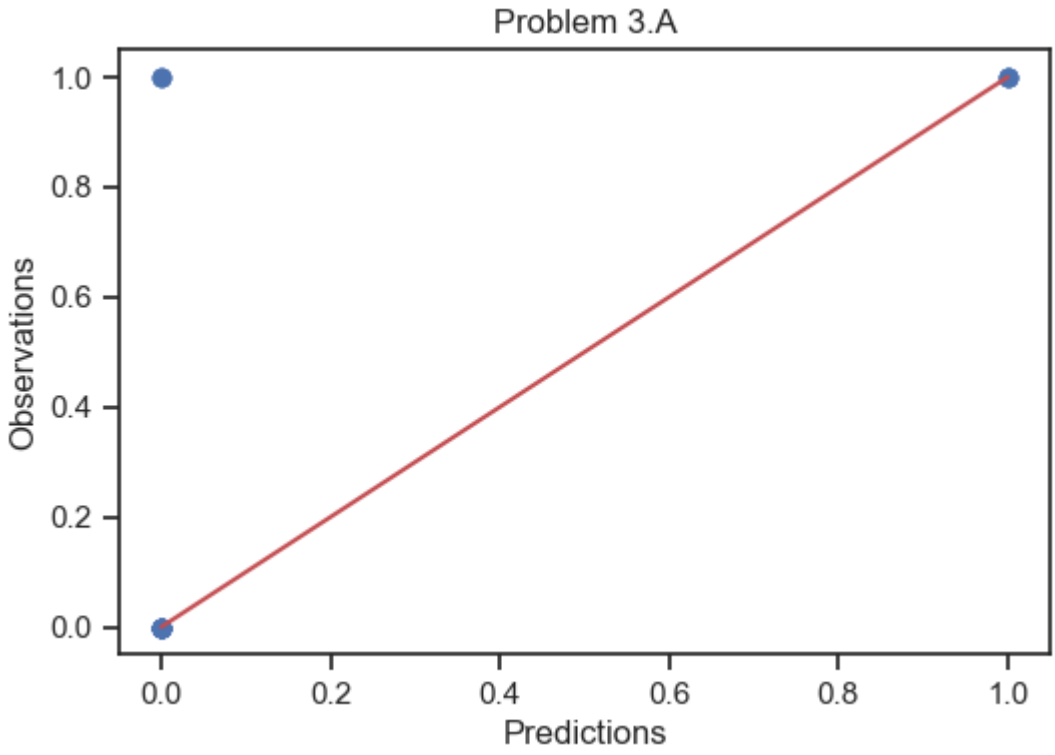
Perform logistic regression between the temperature (x) and the damage label (y). Do not bother doing a validation because there are not a lot of data. Just use a very simple model so that you don't overfit.

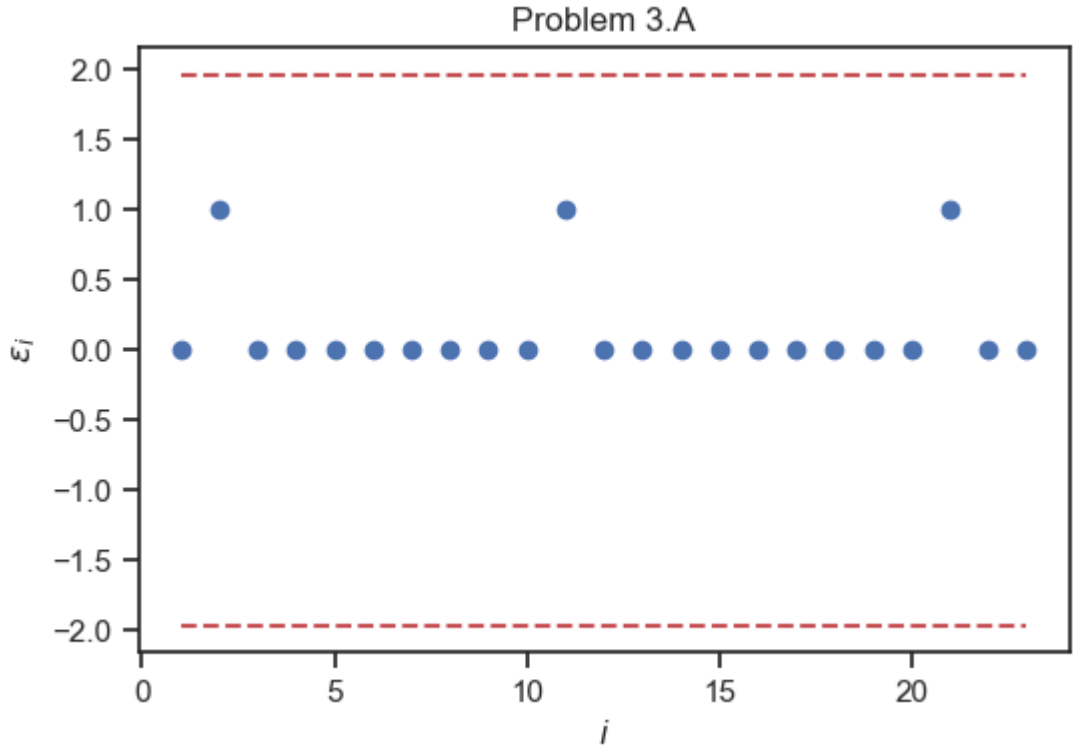
In [206…
```
from sklearn.linear_model import LogisticRegression

# The design matrix
Phi = np.hstack(
    [
        np.ones((x.shape[0], 1)),
        x[:, None]]
)

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

weights = model.coef_
#part a
y_predict = model.predict(Phi)
```

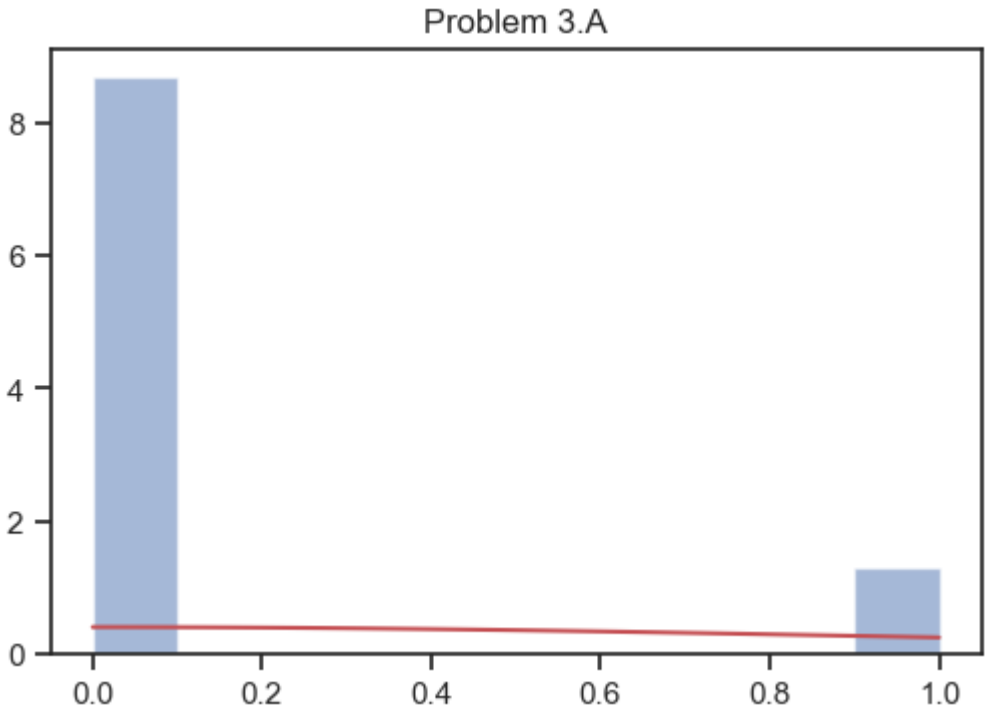
```
fig, ax = plt.subplots()
ax.plot(y_predict, y, 'o')
yys = np.linspace(
    y.min(),
    y.max(),
    100)
ax.plot(yys, yys, 'r-')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations');
plt.title('Problem 3.A')


MSE = np.mean((y_predict - y) **2)
print(f'MSE = {MSE}')



y_std = np.sqrt(np.var(y))
eps = ((y - y_predict) / y_std ) / 2.173
idx = np.arange(1,eps.shape[0] + 1)

fig, ax = plt.subplots()
ax.plot(idx, eps, 'o', label='Standarized errors')
ax.plot(idx, 1.96 * np.ones(eps.shape[0]), 'r--')
ax.plot(idx, -1.96 * np.ones(eps.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$\epsilon_i$');
plt.title('Problem 3.A')


fig, ax = plt.subplots()
ax.hist(eps, alpha=0.5, density=True)
ee = np.linspace(eps.min(), eps.max(), 100)
ax.plot(ee, st.norm.pdf(ee), 'r');
plt.title('Problem 3.A')
```

```
MSE = 0.13043478260869565
```

Out[206… Text(0.5, 1.0, 'Problem 3.A')


Problem 3.A
Observations
Predictions
0.0
0.2
0.4
0.6
0.8
1.0


Problem 3.A
ε_i
i
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
5
10
15
20



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

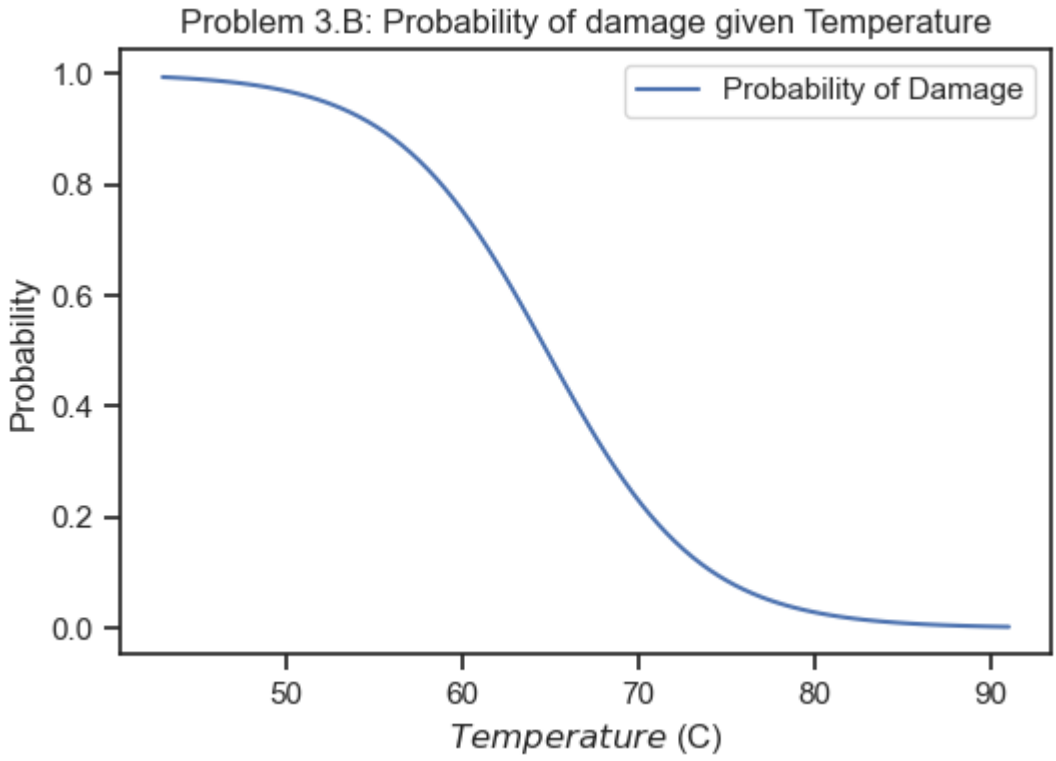
Plot the probability of damage as a function of temperature.

In [207…

```
#part B

fig, ax = plt.subplots()
xx = np.linspace(min(x)-10,max(x)+10,100)
XX = np.hstack([np.ones((xx.shape[0], 1)), xx[:, None]])
predictions_xx = model.predict_proba(XX)

ax.plot(
    xx,
    predictions_xx[:, 1],
    label='Probability of Damage'
)
ax.set_xlabel('$Temperature$ (C)')
ax.set_ylabel('Probability')
plt.legend(loc='best');
plt.title('Problem 3.B: Probability of damage given Temperature')
```

Out[207…

```
Text(0.5, 1.0, 'Problem 3.B: Probability of damage given Temperature')
```



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.

In [208…

```
#part C

LaunchTemp = 31
# c_00 = cost of correctly picking 0 when 0 is true
# c_01 = cost of wrongly picking 0 when 1 is true
# c_11 = cost of correctly picking 1 when 1 is true
# c_10 = cost of wrongly picking 1 when 0 is true
cost_matrix = np.array(
    [
        [1.0, 500],
        [100, 1.0] #wrongly picking safe when it is dangerous is the most costly error
    ]
)

def expected_cost(cost_matrix, prediction_prob):
    """Calculate the expected cost of each decision.

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

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

print('x\tCost of 0\tCost of 1\tTrue label\tChoice')
print('-' * 80)
for i in range(x.shape[0]):
    exp_c = expected_cost(cost_matrix, predictions_xx[i])
    line = f'{x[i]:1.2f}\t{exp_c[0]:1.2f}'
    tmp = f'\t\t{exp_c[1]:1.2f}'
    correct_choice = True
    if exp_c[0] < exp_c[1]:
        line += '*'
        if y[i] == 1:
            correct_choice = False
    else:
        tmp += '*'
        if y[i] == 0:
            correct_choice = False
    line += tmp + f'\t\t{y[i]}'
    if correct_choice:
        line += '\t\tCORRECT'
    else:
        line += '\t\tWRONG'
    print(line)
exp_cost = np.einsum('ij,kj->ki', cost_matrix, predictions_xx)
#print(exp_cost)

xnewx = np.linspace(min(x)-30,max(x)+30,100)
fig, ax = plt.subplots()
exp_cost = np.einsum('ij,kj->ki', cost_matrix, predictions_xx)
decision_idx = np.argmin(exp_cost, axis=1)
ax.plot(xnewx, decision_idx)
ax.set_yticks([0, 1])
ax.set_yticklabels(['No damage ', 'damage'])
ax.set_ylabel('Decision')
ax.set_xlabel('Predictive probability of damage');
plt.title('Problem 3.C: Decision Making Probability')

print('Problem 3.c: According the the graph, no launch should occur under 74 degrees F
```

```
x       Cost of 0       Cost of 1       True label      Choice
--------------------------------------------------------------------------------
66.00   496.85          1.62*           0.0             WRONG
70.00   496.48          1.70*           1.0             CORRECT
69.00   496.06          1.78*           0.0             WRONG
68.00   495.60          1.87*           0.0             WRONG
67.00   495.08          1.98*           0.0             WRONG
72.00   494.50          2.09*           0.0             WRONG
73.00   493.85          2.22*           0.0             WRONG
```

```
70.00   493.13          2.36*           0.0             WRONG
57.00   492.33          2.52*           1.0             CORRECT
63.00   491.43          2.70*           1.0             CORRECT
70.00   490.43          2.90*           1.0             CORRECT
78.00   489.31          3.12*           0.0             WRONG
67.00   488.07          3.37*           0.0             WRONG
53.00   486.68          3.64*           1.0             CORRECT
67.00   485.14          3.95*           0.0             WRONG
75.00   483.43          4.29*           0.0             WRONG
70.00   481.53          4.66*           0.0             WRONG
81.00   479.42          5.08*           0.0             WRONG
76.00   477.08          5.55*           0.0             WRONG
79.00   474.49          6.06*           0.0             WRONG
75.00   471.63          6.63*           1.0             CORRECT
76.00   468.46          7.26*           0.0             WRONG
58.00   464.97          7.95*           1.0             CORRECT
Problem 3.c: According the the graph, no launch should occur under 74 degrees F without
risk of damage. Therefore the launch should not occur
```