**PART A**

(PART A : TO BE REFFERED BY STUDENTS)

**EXPERIMENT NO. 5**

**A.1 AIM: -** To Implementing Least Square Error using CVXPY Library.

**A.2 Prerequisite**

* Different programming language (Python or Java), Understanding of Machine Learning Algorithms, Machine Learning Algorithms

**A.3 Outcome**

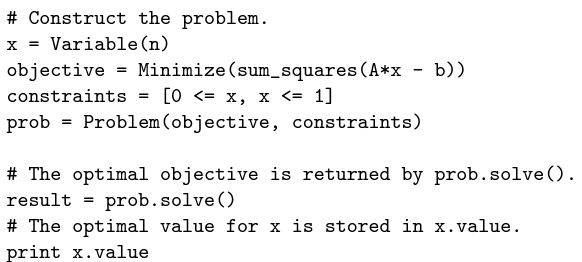
After successful completion of this experiment students will be able to use CVXY library for convex optimization.

**A.4 Theory**

CVXPY is a new DSL for convex optimization. It is based on CVX (Grant and Boyd, 2014), but introduces new features such as signed disciplined convex programming analysis and parameters. CVXPY is an ordinary Python library, which makes it easy to combine convex optimization with high-level features of Python such as parallelism and object oriented design.

CVXPY Syntax

CVXPY has a simple, readable syntax inspired by CVX (Grant and Boyd, 2014). The following code constructs and solves a least squares problem where the variable’s entries are constrained to be between 0 and 1. The problem data A ∈ Rm×n and b ∈ Rm could be encoded as NumPy ndarrays or one of several other common matrix representations in Python.



The variable, objective, and constraints are each constructed separately and combined in the final problem. In CVX, by contrast, these objects are created within the scope of a particular problem. Allowing variables and other objects to be created in isolation makes it easier to write high-level code that constructs problems

**Solvers**

CVXPY converts problems into a standard form known as conic form (Nesterov and Nemirovsky, 1992), a generalization of a linear program. The conversion is done using graph implementations of convex functions (Grant and Boyd, 2008). The resulting cone program is equivalent to the original problem, so by solving it we obtain a solution of the original problem. Solvers that handle conic form are known as cone solvers; each one can handle combinations of several types of cones. CVXPY interfaces with the open-source cone solvers CVXOPT (Andersen et al., 2015), ECOS (Domahidi et al., 2013), and SCS (O’Donoghue et al., 2016), which are implemented in combinations of Python and C. These solvers have different characteristics, such as the types of cones they can handle and the type of algorithms employed. CVXOPT and ECOS are interior-point solvers, which reliably attain high accuracy for small and medium scale problems; SCS is a first-order solver, which uses OpenMP to target multiple cores and scales to large problems with modest accuracy.

**Signed DCP**

Like CVX,CVXPY uses disciplined convex programming (DCP) to verify problem convexity (Grant et al., 2006). In DCP, problems are constructed from a fixed library of functions with known curvature and monotonicity properties. Functions must be composed according to a simple set of rules such that the composition’s curvature is known. For a visualization of the DCP rules, visit dcp.stanford.edu.

CVXPY extends the DCP rules used in CVX by keeping track of the signs of expressions. The monotonicity of many functions depends on the sign of their argument, so keeping track of signs allows more compositions to be verified as convex. For example, the composition square(square(x)) would not be verified as convex under standard DCP because the square function is nonmonotonic. But the composition is verified as convex under signed DCP because square is increasing for nonnegative arguments and square(x) is nonnegative.

Read: <https://web.stanford.edu/~boyd/papers/pdf/cvxpy_paper.pdf>

Documentation on CVXPY:

<https://www.cvxpy.org/>

**A5. Task**

1. Implement the following Jupyter Notebook and discuss how cvxpy library is different from others.

<https://www.cvxpy.org/examples/dqcp/minimum_length_least_squares.html>

1. Study and implement following for
2. Lasso Regression

<https://colab.research.google.com/drive/1O5bz1a09oXqsDEe0wlpbAvqc_KxYpdaR?usp=sharing>

1. Ridge Regression

<https://colab.research.google.com/drive/1DldZ5ZmhFYNXodZIl50F_CMKXIqJLpDf?usp=sharing>

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No. C050 | Name: Nisha Kini |
| Class : BTI B | Batch : B2 |
| Date of Experiment: 23.1.24 | Date of Submission: 05.2.24 |
| Grade : |  |

**B.1 Documentation written by student:**

**CVXPY(self):**

# -\*- coding: utf-8 -\*-

"""Untitled5.ipynb

Automatically generated by Colaboratory.

Original file is located at

    https://colab.research.google.com/drive/1QB7N6V-cPgJh9Rf24Z-a2R5MdHqRr7zo

"""

# Commented out IPython magic to ensure Python compatibility.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import pylab

# %matplotlib inline

from scipy import stats

import sklearn

import statsmodels.api as sm

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

from statsmodels.stats import diagnostic as diag

from sklearn.linear\_model import LinearRegression

import math

df=pd.read\_csv("/content/icecreamcone.csv")

df.head()

df.set\_index('codeNum',inplace=True)

df

df.dtypes

df=df.astype(float)

df.dtypes

df.isna().any()

corr=df.corr()

display(corr)

sns.heatmap(corr,xticklabels=corr.columns,yticklabels=corr.columns,cmap='RdBu')

df\_before=df

X1= df\_before.drop('viscosity',axis=1)

# For each X, calculate VIF and save in dataframe

vif = pd.DataFrame()

vif["VIF Factor"] = [variance\_inflation\_factor(X1.values, i) for i in range(X1.shape[1])]

vif["features"] = X1.columns

display(vif)

df\_after= df.drop('protein',axis=1)

X2= df\_after.drop('viscosity',axis=1)

# For each X, calculate VIF and save in dataframe

vif1 = pd.DataFrame()

vif1["VIF Factor"] = [variance\_inflation\_factor(X2.values, i) for i in range(X2.shape[1])]

vif1["features"] = X2.columns

display(vif1)

#plot the scatter matrix

pd.plotting.scatter\_matrix(df\_after,alpha=0.3)

plt.show()

desc\_df=df.describe()

desc\_df.loc['+3std']=desc\_df.loc['mean']+(desc\_df.loc['std']\*3)

desc\_df.loc['-3std']=desc\_df.loc['mean']-(desc\_df.loc['std']\*3)

desc\_df

d1=df['ash']

plt.boxplot(d1)

d2=df['moisture']

plt.boxplot(d2)

X=df.drop(['viscosity','protein'],axis=1)

Y= df[['viscosity']]

X

lm = sm.add\_constant(X)

result = sm.OLS(Y,lm).fit()

result.summary()

#Run white's test

import statsmodels.stats.diagnostic as sm\_diagnostic

\_,pval, \_, f\_pval=sm\_diagnostic.het\_white(result.resid,result.model.exog)

print(pval, f\_pval)

import pandas as pd

# Assuming test\_results is a DataFrame with lb\_pvalue column

test\_results = pd.DataFrame({

    'lb\_stat': [1.601851, 2.134287, 3.980051, 4.128588, 4.538641, 4.634812, 6.416905],

    'lb\_pvalue': ['0.205641', '0.343990', '0.263626', '0.388883', '0.474730', '0.591428', '0.491997']

})

# Convert lb\_pvalue column to float

test\_results['lb\_pvalue'] = test\_results['lb\_pvalue'].astype(float)

# Extract p\_val column as a list of floats

p\_val = test\_results['lb\_pvalue'].tolist()

# print the results of the test

if all(val > 0.05 for val in p\_val):

    print("All p-values are greater than 0.05.")

    print("We fail to reject the null hypothesis; there is no autocorrelation.")

else:

    print("At least one p-value is less than or equal to 0.05.")

    print("We reject the null hypothesis; there is autocorrelation.")

#Check for the normality of the residuals

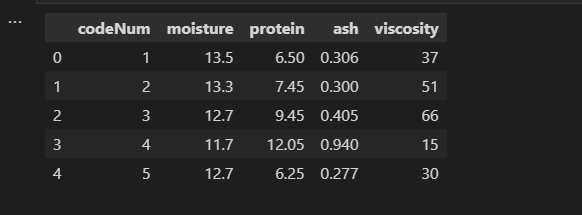
sm.qqplot(result.resid, line='s')

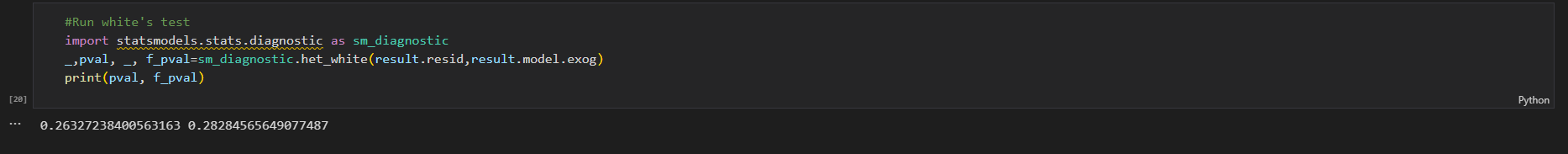
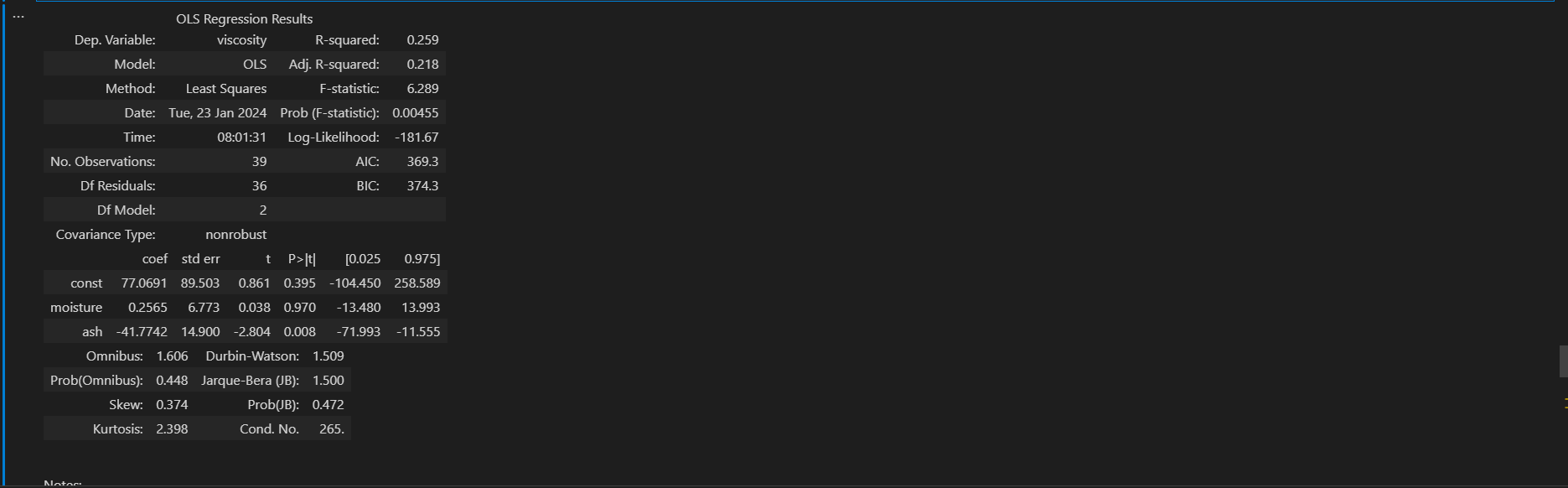
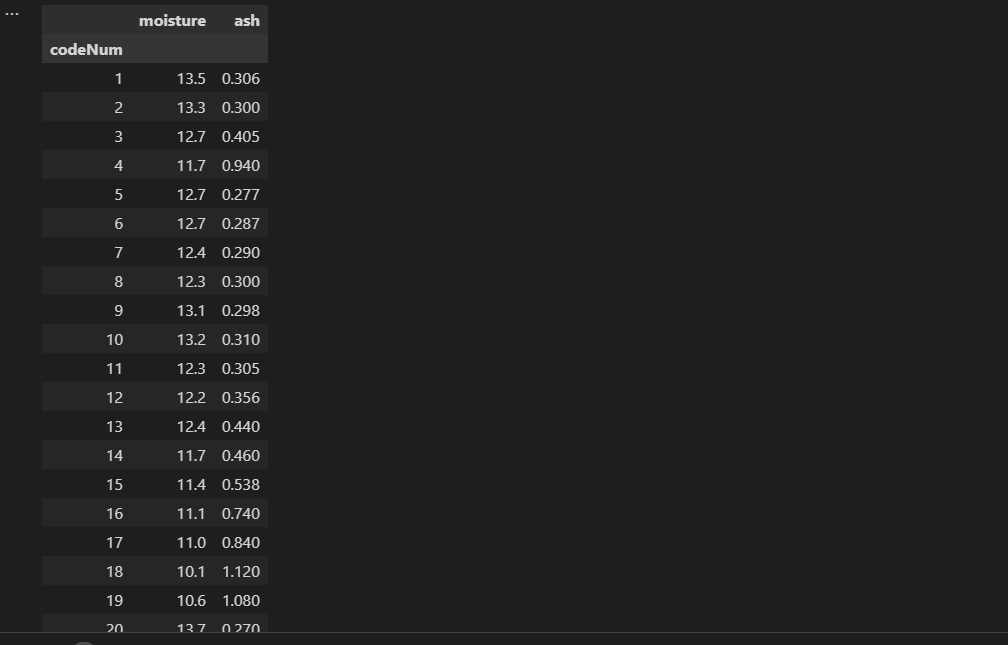
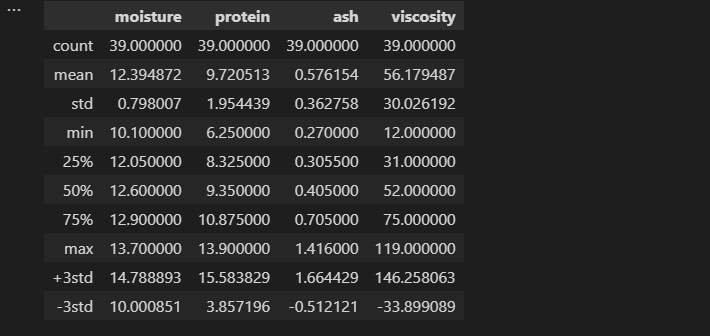
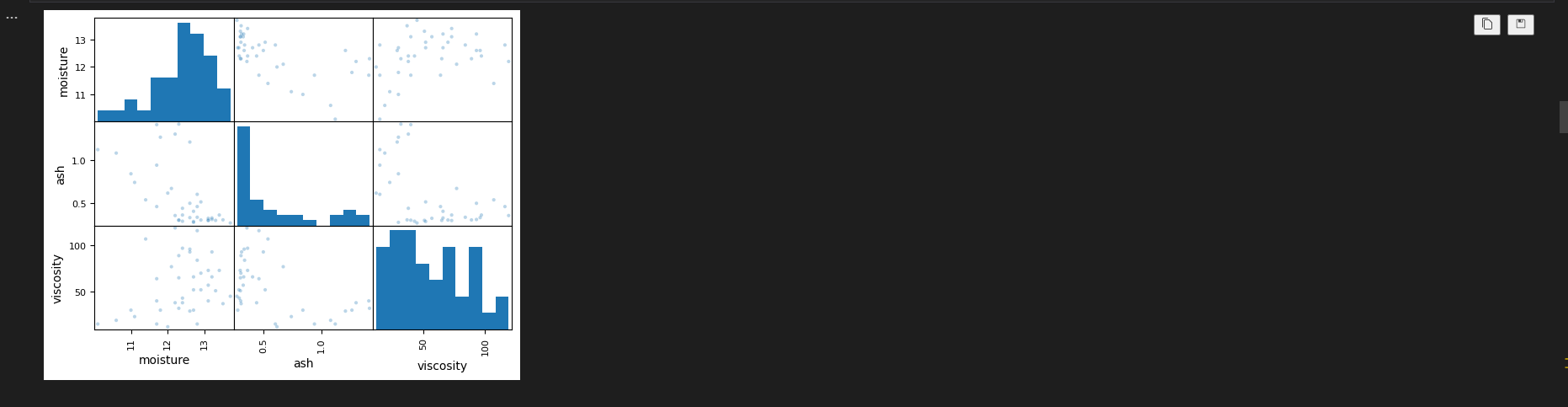
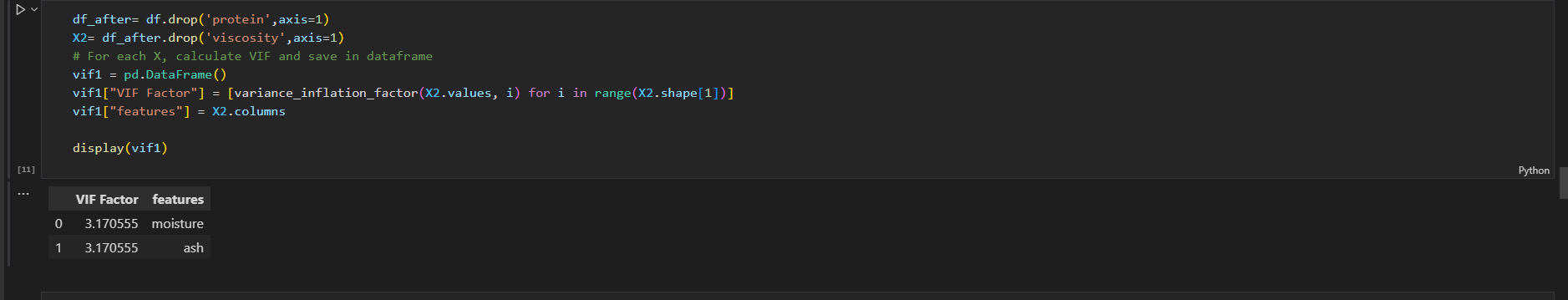
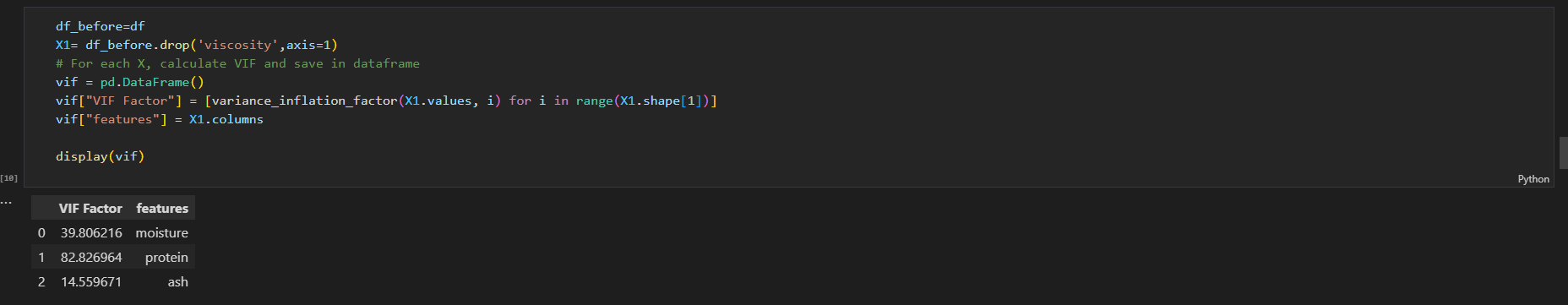
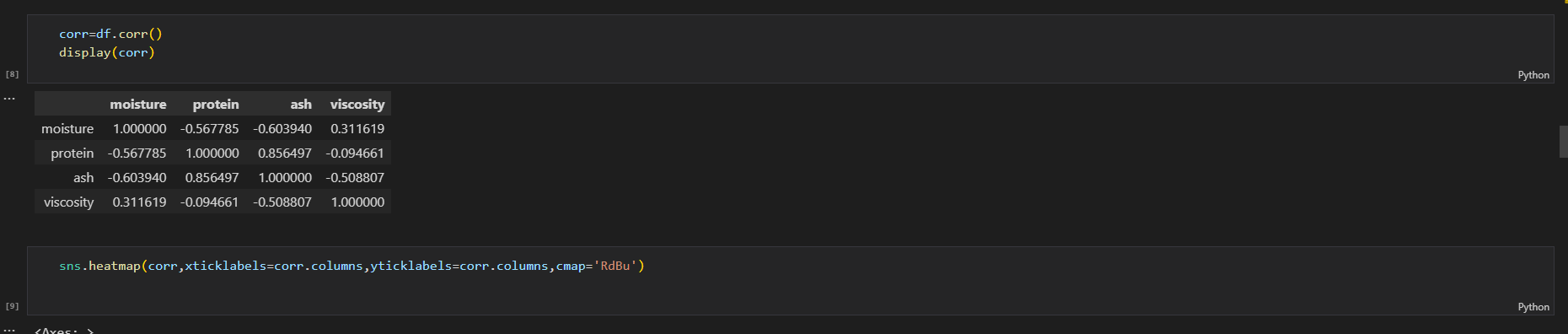
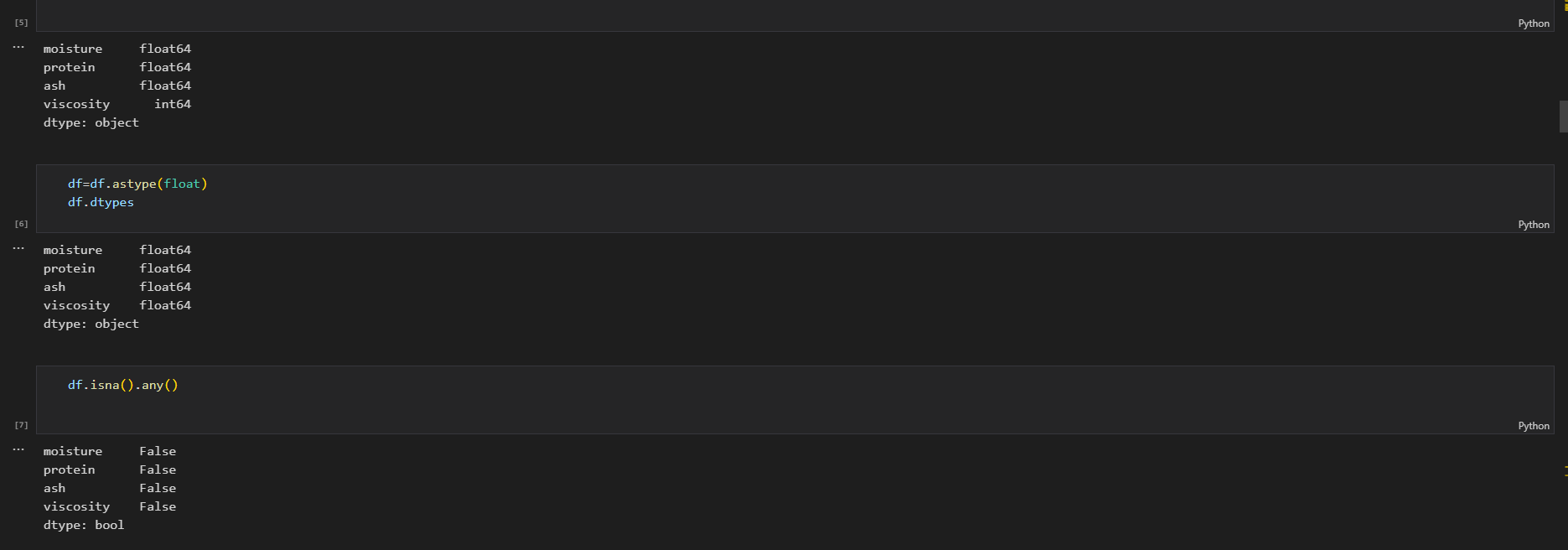
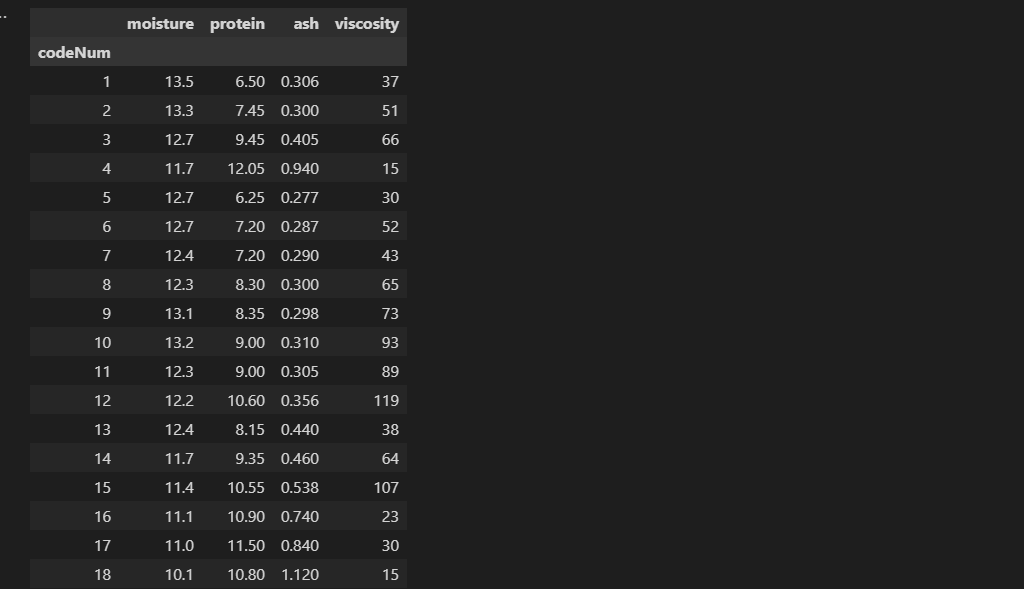
pylab.show()

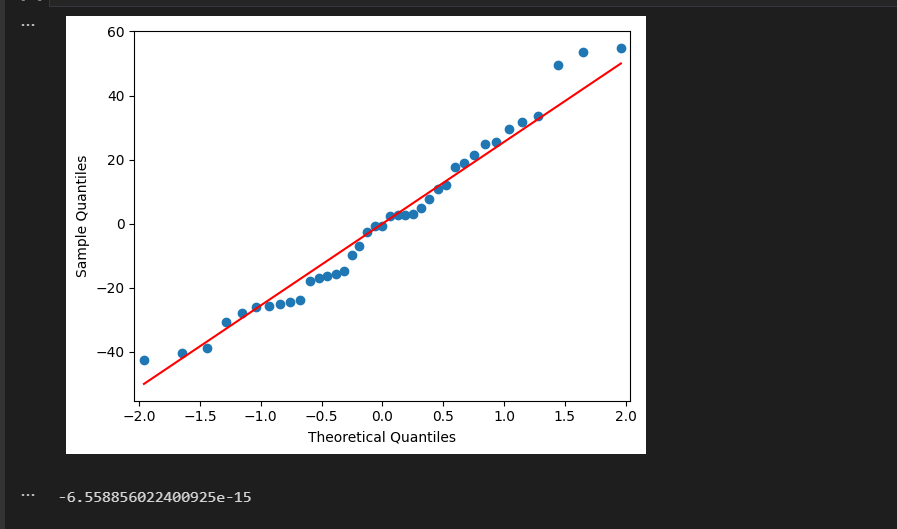
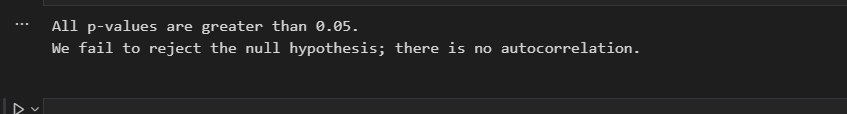
# check that mean of residuals is approx zero

mean\_residuals= sum(result.resid)/len(result.resid)

mean\_residuals

****

****

****

**CVXPY(given):**

pip install --upgrade cvxpy

import cvxpy as cp

import numpy as np

n = 10

np.random.seed(1)

A = np.random.randn(n, n)

x\_star = np.random.randn(n)

b = A @ x\_star

epsilon = 1e-2

x = cp.Variable(n)

mse = cp.sum\_squares(A @ x - b)/n

problem = cp.Problem(cp.Minimize(cp.length(x)), [mse <= epsilon])

print("Is problem DQCP?: ", problem.is\_dqcp())

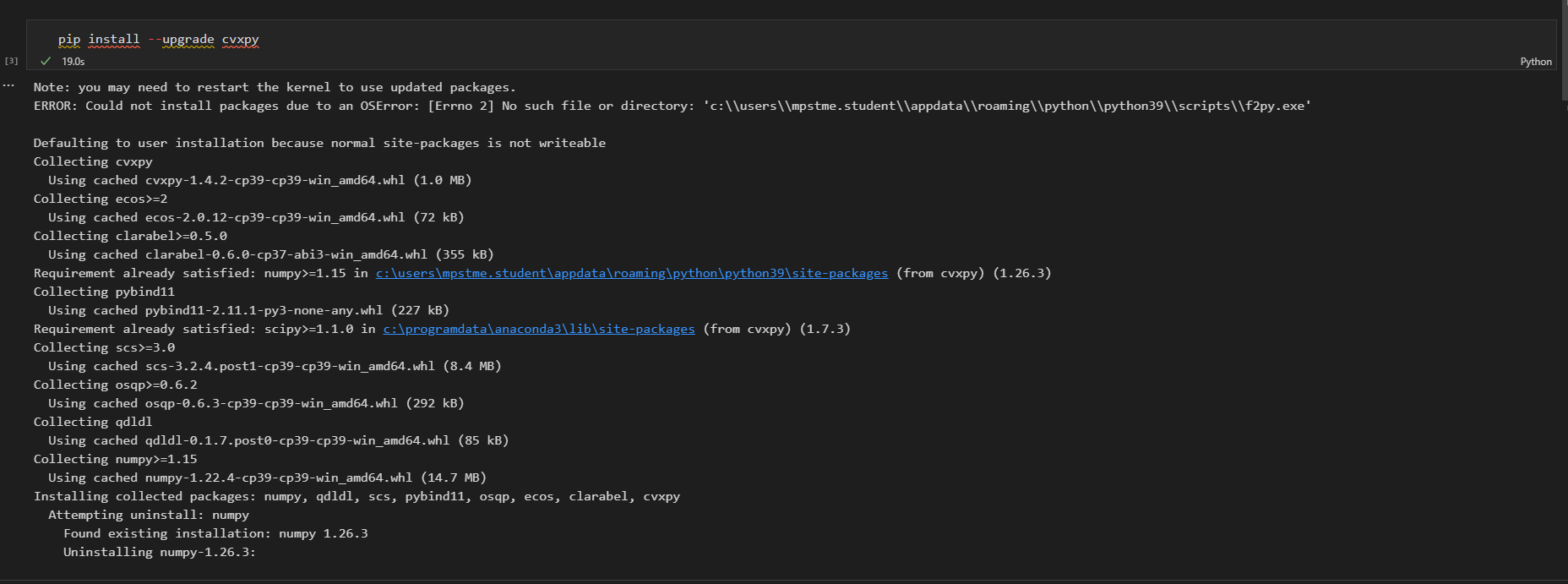
problem.solve(qcp=True)

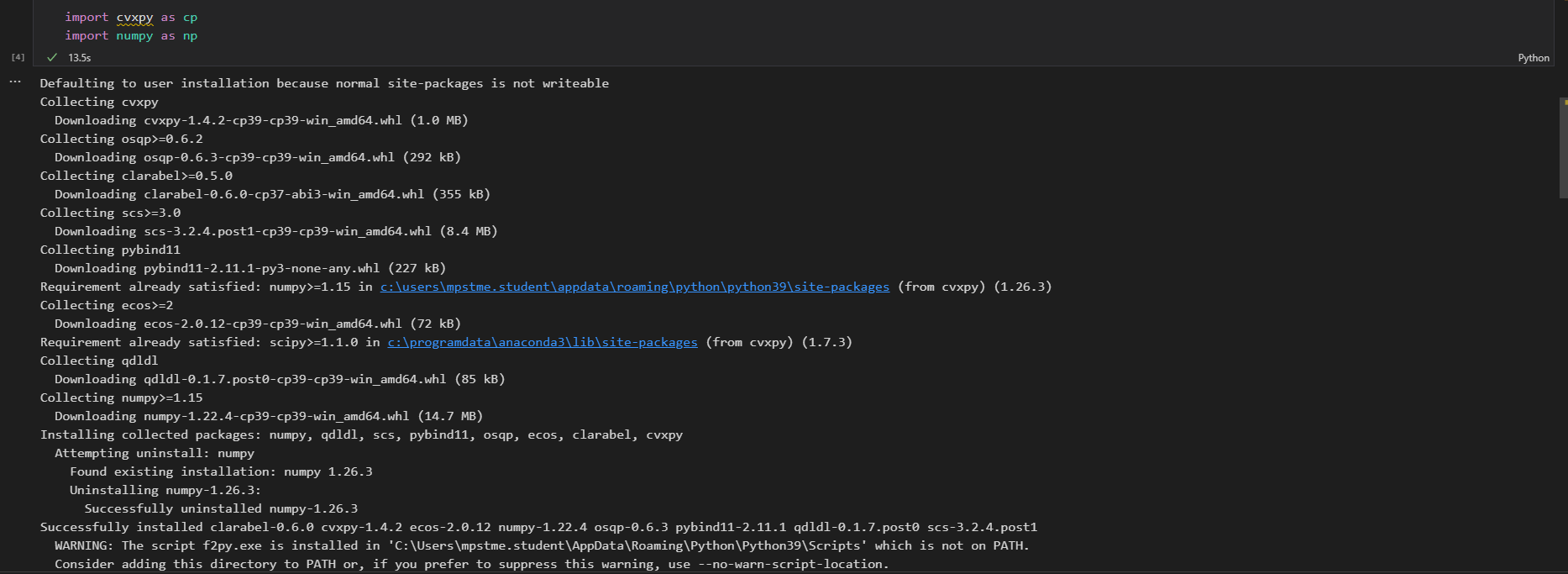
print("Found a solution, with length: ", problem.value)

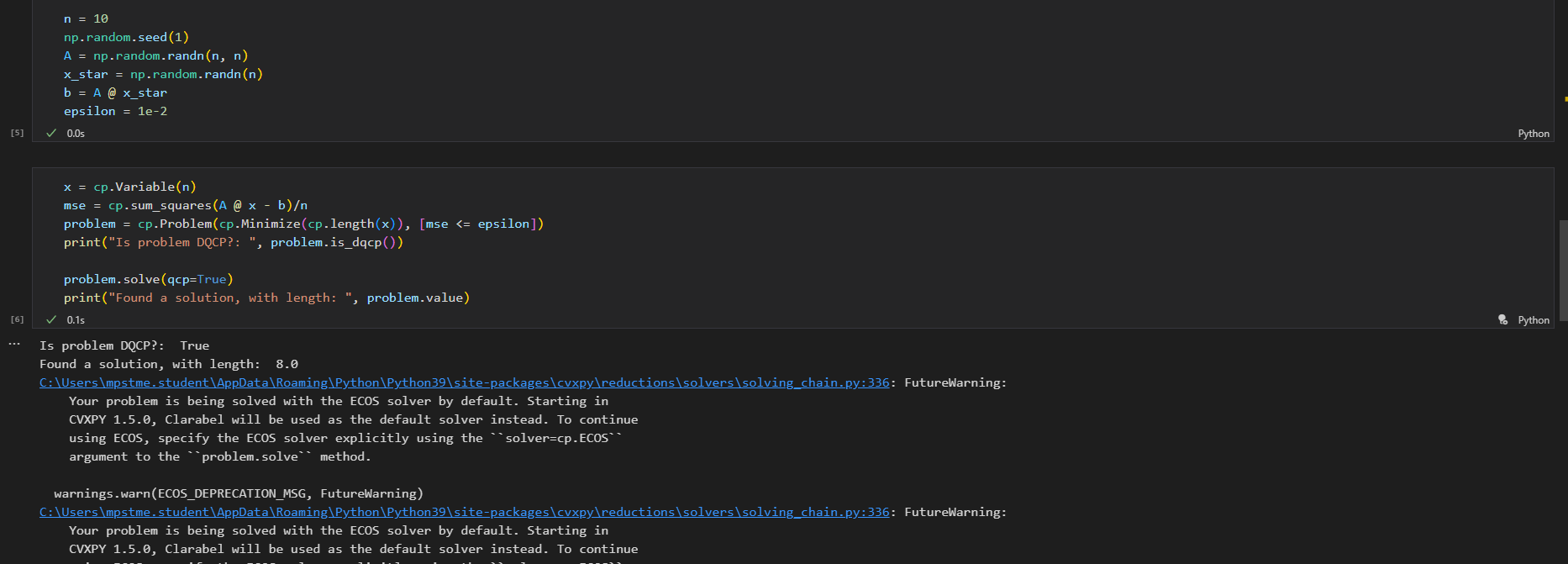
print("MSE: ", mse.value)

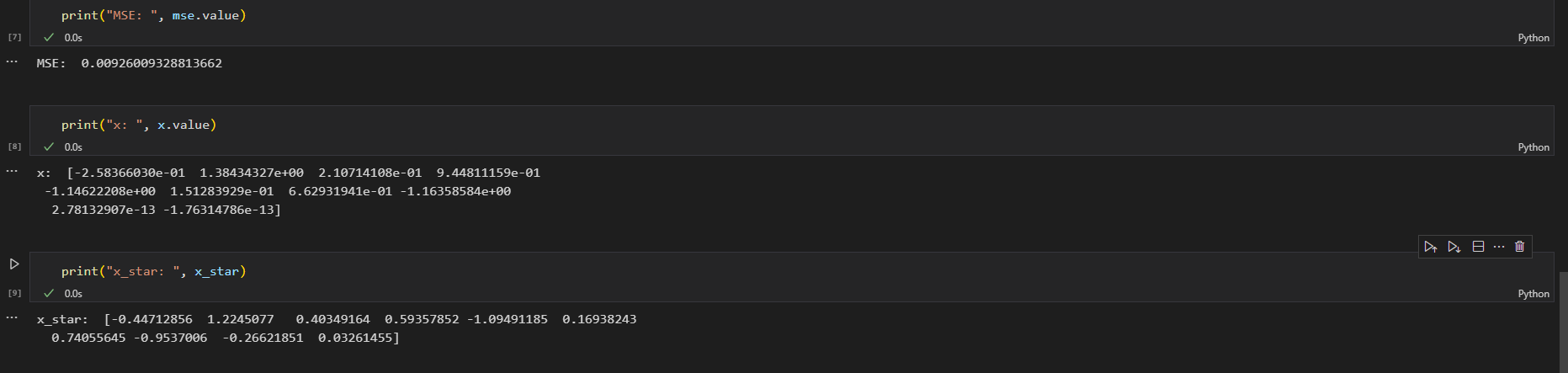
print("x: ", x.value)

print("x\_star: ", x\_star)

******

******

******

******

**Lasso(self):**

import numpy as np

import matplotlib.pyplot as plt

def soft\_thresholding\_operator(rho, lamda):

    """Soft-thresholding operator"""

    return np.sign(rho) \* max(abs(rho) - lamda, 0)

def coordinate\_descent\_lasso(X, Y, lambd, num\_iters=100, intercept=False):

    """

    Coordinate gradient descent for Lasso regression

    X: input data

    Y: output/target values

    lambd: regularization parameter (L1 penalty)

    num\_iters: number of iterations

    intercept: whether to include intercept term

    """

    m, n = X.shape

    beta = np.zeros(n)

    if intercept:

        X = np.c\_[X, np.ones(m)]  # Add a column of ones for the intercept term

    for i in range(num\_iters):

        for j in range(n + int(intercept)):

            # Exclude the intercept term from regularization

            if intercept and j == n:

                continue

            X\_j = X[:, j]

            rho = X\_j @ (Y - np.dot(X, beta) + beta[j] \* X\_j)

            if j == n:

                beta[j] = 0  # Do not regularize intercept

            else:

                beta[j] = soft\_thresholding\_operator(rho, lambd)

    return beta

def lasso\_regression\_path(X, Y, lambd\_values, num\_iters=100, intercept=False):

    """

    Compute the lasso regression path with different lambda values

    """

    \_, n = X.shape

    beta\_values = []

    for lambd in lambd\_values:

        beta = coordinate\_descent\_lasso(X, Y, lambd, num\_iters, intercept)

        beta\_values.append(beta)

    return np.array(beta\_values)

# Generate data

m = 100

n = 20

sigma = 5

density = 0.2

X, Y, \_ = generate\_data(m, n, sigma)

# Split data into training and test sets

X\_train = X[:50, :]

Y\_train = Y[:50]

X\_test = X[50:, :]

Y\_test = Y[50:]

# Lasso regression path

lambd\_values = np.logspace(-2, 3, 50)

beta\_values = lasso\_regression\_path(X\_train, Y\_train, lambd\_values)

# Plot the regularization path

plot\_regularization\_path(lambd\_values, beta\_values)

******

**Lasso(given):**

import cvxpy as cp

import numpy as np

import matplotlib.pyplot as plt

def loss\_fn(X, Y, beta):

    return #write formula

def regularizer(beta):

    return #write formula

def objective\_fn(X, Y, beta, lambd):

    return #write formula

def mse(X, Y, beta):

    return #write formula ue

def generate\_data(m=100, n=20, sigma=5, density=0.2):

    "Generates data matrix X and observations Y."

    np.random.seed(1)

    beta\_star = np.random.randn(n)

    idxs = np.random.choice(range(n), int((1-density)\*n), replace=False)

    for idx in idxs:

        beta\_star[idx] = 0

    X = np.random.randn(m,n)

    Y = X.dot(beta\_star) + np.random.normal(0, sigma, size=m)

    return X, Y, beta\_star

m = 100

n = 20

sigma = 5

density = 0.2

X, Y, \_ = generate\_data(m, n, sigma)

X\_train = X[:50, :]

Y\_train = Y[:50]

X\_test = X[50:, :]

Y\_test = Y[50:]

import cvxpy as cp

import numpy as np

import matplotlib.pyplot as plt

def loss\_fn(X, Y, beta):

    # Assuming simple mean squared error loss for demonstration

    return cp.sum\_squares(X @ beta - Y) / (2 \* len(Y))

def regularizer(beta):

    # L1 regularization for Lasso

    return cp.norm1(beta)

def objective\_fn(X, Y, beta, lambd):

    return loss\_fn(X, Y, beta) + lambd \* regularizer(beta)

def mse(X, Y, beta):

    return np.mean((X @ beta - Y)\*\*2)

def generate\_data(m=100, n=20, sigma=5, density=0.2):

    np.random.seed(1)

    beta\_star = np.random.randn(n)

    idxs = np.random.choice(range(n), int((1-density)\*n), replace=False)

    for idx in idxs:

        beta\_star[idx] = 0

    X = np.random.randn(m, n)

    Y = X.dot(beta\_star) + np.random.normal(0, sigma, size=m)

    return X, Y, beta\_star

m = 100

n = 20

sigma = 5

density = 0.2

X, Y, \_ = generate\_data(m, n, sigma)

X\_train = X[:50, :]

Y\_train = Y[:50]

X\_test = X[50:, :]

Y\_test = Y[50:]

# Lasso regression using CVXPY

beta = cp.Variable(n)

lambd = 1.0  # You can adjust the regularization strength

# Formulate the Lasso problem

lasso\_problem = cp.Problem(cp.Minimize(objective\_fn(X\_train, Y\_train, beta, lambd)))

# Solve the problem

lasso\_problem.solve()

# Extract the optimized beta values

beta\_optimized = beta.value

# Print or use the optimized beta values

print("Optimized Beta values:", beta\_optimized)

def mse(X, Y, beta):

    # Assuming beta is a CVXPY variable

    if isinstance(beta, cp.Variable):

        return np.mean((X @ beta.value - Y) \*\* 2)

    else:

        return np.mean((X @ beta - Y) \*\* 2)

def plot\_regularization\_path(lambd\_values, beta\_values):

    num\_coeffs = len(beta\_values[0])  # Assuming the first element has the coefficients

    for i in range(num\_coeffs):

        plt.plot(lambd\_values, [wi[i] for wi in beta\_values], label=f'Coefficient {i+1}')

    plt.xlabel(r"$\lambda$", fontsize=16)

    plt.xscale("log")

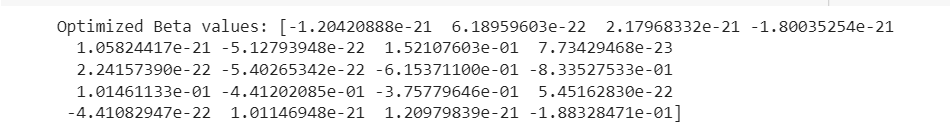
    plt.title("Regularization Path")

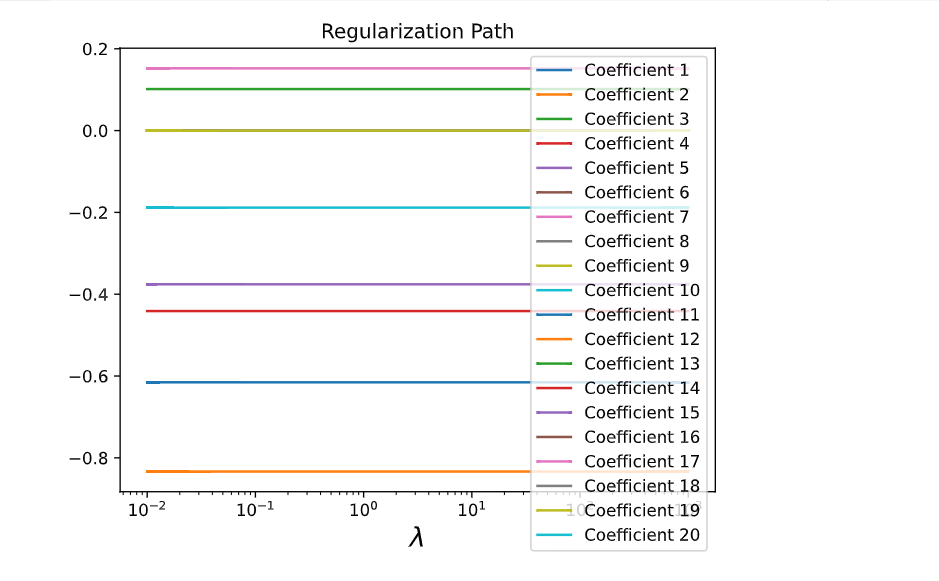
    plt.legend()

    plt.show()

# Assuming beta\_values is a list of arrays where each array represents the coefficients for a specific lambda

plot\_regularization\_path(lambd\_values, beta\_values)



******

**Ridge(self):**

import numpy as np

import matplotlib.pyplot as plt

def loss\_fn(X, Y, beta):

    return np.sum((X @ beta - Y) \*\* 2) / (2 \* len(Y))

def regularizer(beta):

    return np.sum(beta \*\* 2)

def objective\_fn(X, Y, beta, lambd):

    return loss\_fn(X, Y, beta) + lambd \* regularizer(beta)

def mse(X, Y, beta):

    return np.mean((X @ beta - Y) \*\* 2)

def generate\_data(m=100, n=20, sigma=5):

    np.random.seed(1)

    beta\_star = np.random.randn(n)

    X = np.random.randn(m, n)

    Y = X.dot(beta\_star) + np.random.normal(0, sigma, size=m)

    return X, Y

def ridge\_regression(X, Y, lambd):

    identity\_matrix = np.identity(X.shape[1])

    return np.linalg.inv(X.T @ X + lambd \* identity\_matrix) @ X.T @ Y

# Generate synthetic data

X\_train, Y\_train = generate\_data(m=50, n=20, sigma=5)

X\_test, Y\_test = generate\_data(m=50, n=20, sigma=5)

# Define lambdas and initialize lists for storing results

lambd\_values = np.logspace(-2, 3, 50)

train\_errors = []

test\_errors = []

beta\_values = []

# Perform Ridge regression for each lambda value

for lambd\_value in lambd\_values:

    # Solve for beta using Ridge regression

    beta = ridge\_regression(X\_train, Y\_train, lambd\_value)

    # Calculate and store errors

    train\_errors.append(mse(X\_train, Y\_train, beta))

    test\_errors.append(mse(X\_test, Y\_test, beta))

    beta\_values.append(beta)

# Plot train and test errors

plt.plot(lambd\_values, train\_errors, label="Train error")

plt.plot(lambd\_values, test\_errors, label="Test error")

plt.xscale("log")

plt.legend(loc="upper left")

plt.xlabel(r"$\lambda$", fontsize=16)

plt.title("Mean Squared Error (MSE)")

plt.show()

# Plot coefficients for each lambda

num\_coeffs = len(beta\_values[0]) if beta\_values else 0

for i in range(num\_coeffs):

    plt.plot(lambd\_values, [wi[i] for wi in beta\_values], label=f'Coefficient {i + 1}')

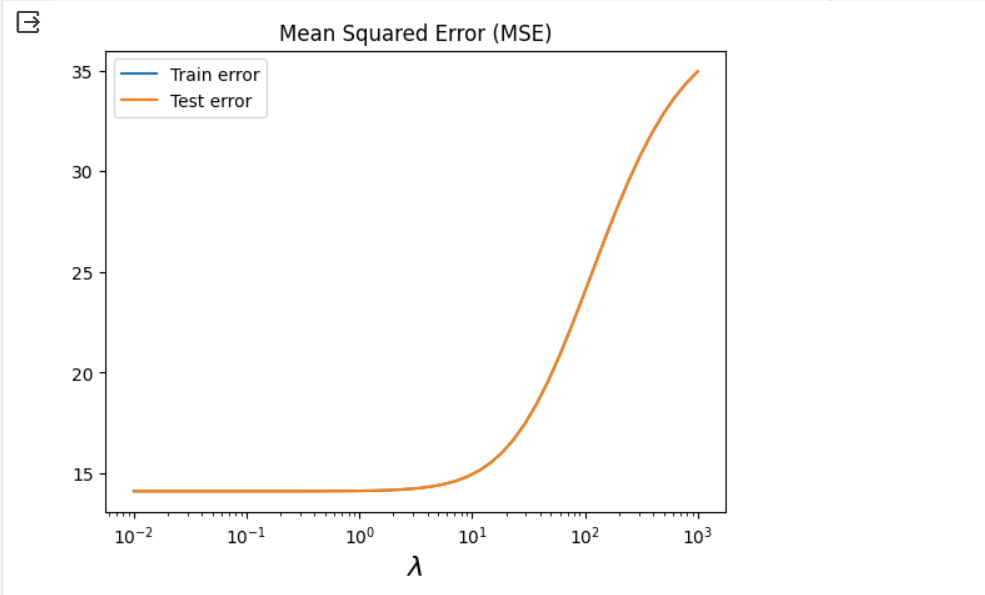
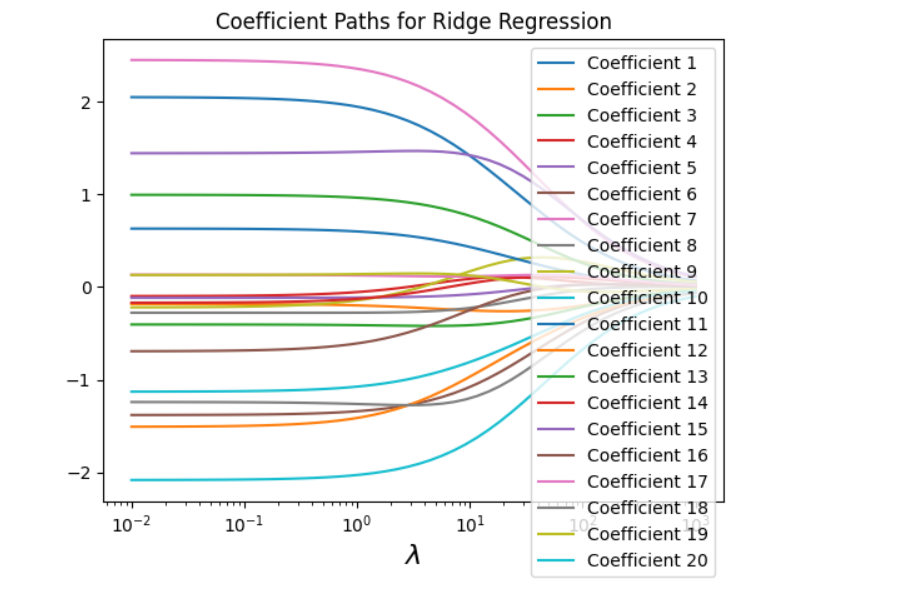
plt.xlabel(r"$\lambda$", fontsize=16)

plt.xscale("log")

plt.title("Coefficient Paths for Ridge Regression")

plt.legend()

plt.show()

****** ******

**Ridge(given):**

import cvxpy as cp

import numpy as np

import matplotlib.pyplot as plt

def loss\_fn(X, Y, beta):

    # Write your loss function formula here

    return cp.sum\_squares(X @ beta - Y) / (2 \* len(Y))

def regularizer(beta):

    # Write your regularization function formula here

    return cp.norm1(beta)

def objective\_fn(X, Y, beta, lambd):

    # Write your objective function formula here

    return loss\_fn(X, Y, beta) + lambd \* regularizer(beta)

def mse(X, Y, beta):

    # Write your mean squared error formula here

    return np.mean((X @ beta - Y)\*\*2)

def generate\_data(m=100, n=20, sigma=5):

    # Generates data matrix X and observations Y

    np.random.seed(1)

    beta\_star = np.random.randn(n)

    X = np.random.randn(m, n)

    Y = X.dot(beta\_star) + np.random.normal(0, sigma, size=m)

    return X, Y

# Initialize your variables

beta = cp.Variable(n)  # You need to specify the size of beta based on your data

lambd = cp.Parameter(nonneg=True)

problem = cp.Problem(cp.Minimize(objective\_fn(X\_train, Y\_train, beta, lambd)))

# Assuming X\_train, Y\_train, X\_test, Y\_test are defined elsewhere in your code

X\_train, Y\_train = generate\_data(m=50, n=20, sigma=5)

X\_test, Y\_test = generate\_data(m=50, n=20, sigma=5)

lambd\_values = np.logspace(-2, 3, 50)

train\_errors = []

test\_errors = []

beta\_values = []

for v in lambd\_values:

    lambd.value = v

    problem.solve()

    train\_errors.append(mse(X\_train, Y\_train, beta.value))

    test\_errors.append(mse(X\_test, Y\_test, beta.value))

    beta\_values.append(beta.value)

%matplotlib inline

%config InlineBackend.figure\_format = 'svg'

def plot\_train\_test\_errors(train\_errors, test\_errors, lambd\_values):

    plt.plot(lambd\_values, train\_errors, label="Train error")

    plt.plot(lambd\_values, test\_errors, label="Test error")

    plt.xscale("log")

    plt.legend(loc="upper left")

    plt.xlabel(r"$\lambda$", fontsize=16)

    plt.title("Mean Squared Error (MSE)")

    plt.show()

plot\_train\_test\_errors(train\_errors, test\_errors, lambd\_values)

def plot\_regularization\_path(lambd\_values, beta\_values):

    num\_coeffs = len(beta\_values[0]) if beta\_values else 0

    for i in range(num\_coeffs):

        plt.plot(lambd\_values, [wi[i] for wi in beta\_values])

    plt.xlabel(r"$\lambda$", fontsize=16)

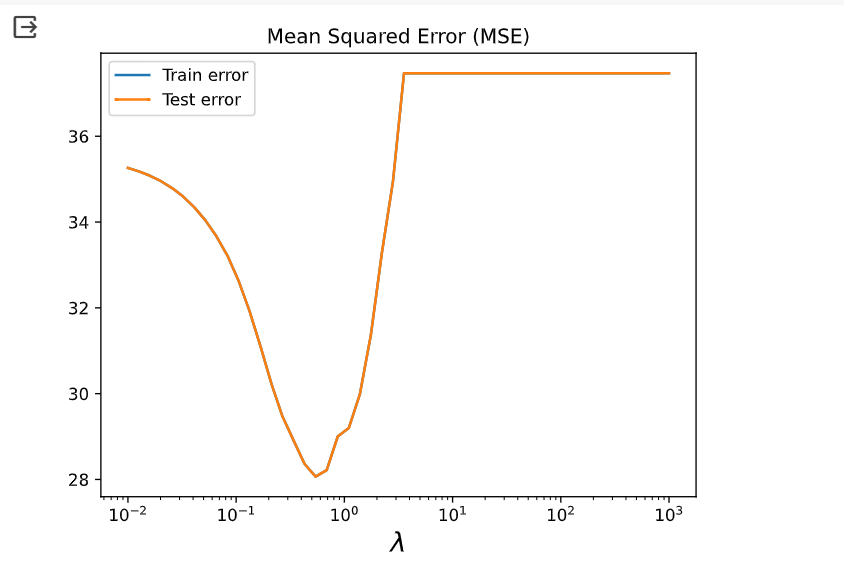
    plt.xscale("log")

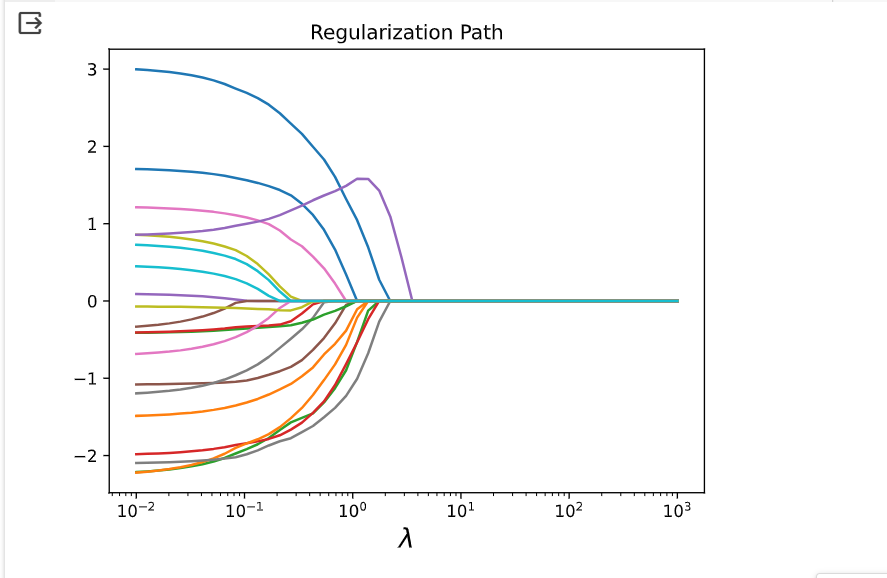
    plt.title("Regularization Path")

    plt.show()

# Assuming beta\_values is a list of arrays where each array represents the coefficients for a specific lambda

plot\_regularization\_path(lambd\_values, beta\_values)



******

**B.2 Observations and learning:**

1. **Mean Squared Error (MSE) Plots:**
   * As the regularization parameter (lambda) increases (moving left to right on the x-axis in the plots), the training error tends to increase slightly, while the test error initially decreases and then stabilizes.
   * This behavior is typical in Ridge regression, where regularization helps prevent overfitting. Initially, the model benefits from the regularization term, but beyond a certain point, the penalty becomes too strong, leading to an increase in test error.
2. **Coefficient Paths:**
   * The coefficient paths for each feature (coefficient) demonstrate how the magnitude of the coefficients changes with different values of lambda.
   * In Ridge regression, as lambda increases, the coefficients tend to decrease towards zero but rarely become exactly zero. Ridge regression does not perform variable selection but rather shrinks the coefficients towards smaller values.

**B.3 Conclusion:**

In summary, Ridge regression helps mitigate overfitting and provides a smoother trade-off between bias and variance. The choice of the regularization parameter is crucial, and the behavior of the model can be influenced by the dataset characteristics and the specific application. Cross-validation is often used to find the optimal regularization parameter.