**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**

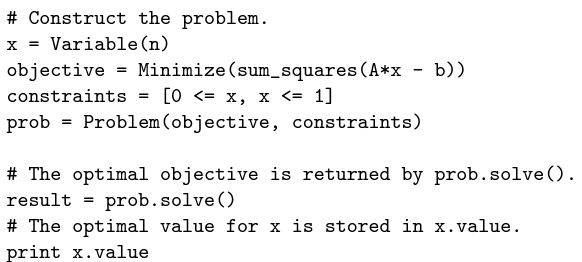
After successful completion of this experiment students will be able to use CVXY library for convex optimization.

**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.: C027 | Name: Vishesh Giyanani |
| Class : B | Batch : B1 |
| Date of Experiment: 09-02-2024 | Date of Submission: 09-02-2024 |
| Grade : |  |

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

1. Ridge regression

# Vishesh Giyanani

# C027

import cvxpy as cp

import numpy as np

import matplotlib.pyplot as plt

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

return cp.pnorm(*X* @ *beta* - *Y*, *p*=2)\*\*2

def regularizer(*beta*):

return cp.pnorm(*beta*, *p*=2)\*\*2

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

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

def mse(*X*, *Y*, *beta*):

return (1.0 / *X*.shape[0]) \* loss\_fn(*X*, *Y*, *beta*).value

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*)

# Generate an ill-conditioned data matrix

X = np.random.randn(*m*, *n*)

# Corrupt the observations with additive Gaussian noise

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

return X, Y

m = 100

n = 20

sigma = 5

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

X\_train = X[:50, :]

Y\_train = Y[:50]

X\_test = X[50:, :]

Y\_test = Y[50:]

beta = cp.Variable(n)

lambd = cp.Parameter(*nonneg*=True)

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

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))

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

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])

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()

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

2. Lasso Regression

import cvxpy as cp

import numpy as np

import matplotlib.pyplot as plt

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

return cp.norm2(*X* @ *beta* - *Y*)\*\*2

def regularizer(*beta*):

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 (1.0 / *X*.shape[0]) \* loss\_fn(*X*, *Y*, *beta*).value

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:]

beta = cp.Variable(n)

lambd = cp.Parameter(*nonneg*=True)

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

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))

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

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])

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")

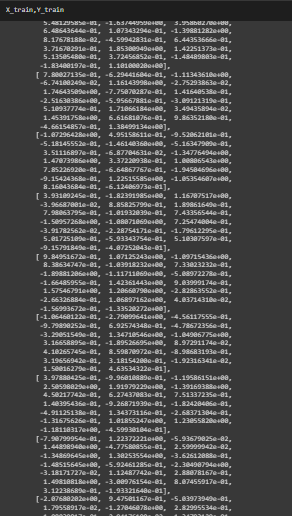
plt.show()

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

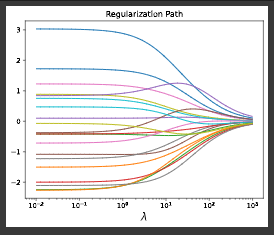
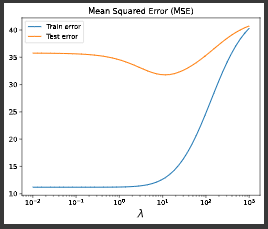
**B.2 Observations and learning:**

**Ridge regression**

***Input***

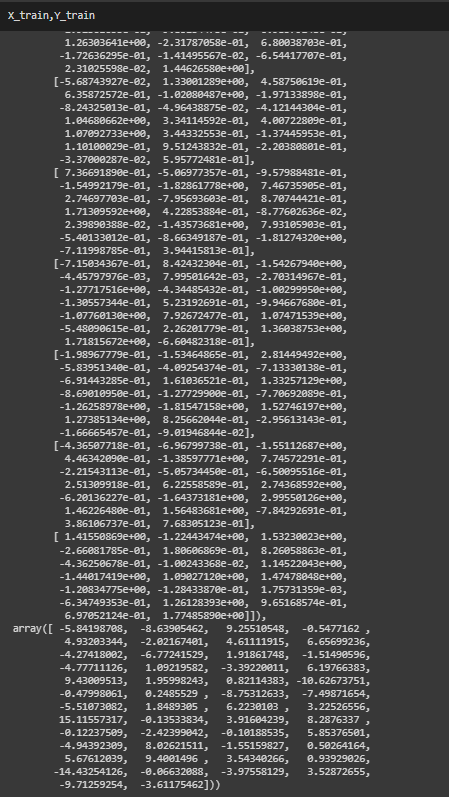
****

**Output**

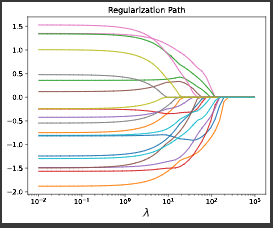
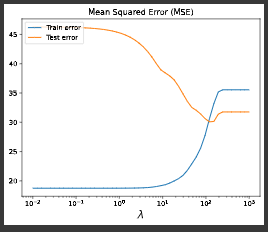
******

**Lasso Regression**

***Input:***

******

**Output**

******

**B.3 Conclusion:**

Learnt and implemented cvxpy library to solve optimization problems.