STA 6106 – EXAM 1

Solutions proposed by Anmol Sureshkumar Panchal , UID:4446829

Question 1:

a)

Bootstrap Method

The bootstrap method is a statistical technique for estimating quantities about a population by averaging estimates from multiple small data samples. Importantly, samples are constructed by drawing observations from a large data sample one at a time and returning them to the data sample after they have been chosen. This allows a given observation to be included in a given small sample more than once. This approach to sampling is called sampling with replacement.

The bootstrap method can be used to estimate a quantity of a population. This is done by repeatedly taking small samples, calculating the statistic, and taking the average of the calculated statistics. We can summarize this procedure as follows:

1. Choose a number of bootstrap samples to perform
2. Choose a sample size
3. For each bootstrap sample
4. Draw a sample with replacement with the chosen size
5. Calculate the statistic on the sample
6. Calculate the mean of the calculated sample statistics.

In our case we have a small sample size of 33 data points, so we will just perform resample operations to create a sample for bootstrapping. So we will first perform sampling of our dataset by:

```
data = pd.read_csv(r"C:\Users\anmol\Downloads\mtcars.csv")
## r before your normal string helps it to convert normal string to raw string
# Resampling from our dataset
from sklearn.utils import resample
boot = resample(data.iloc[:,1:2], replace=False, n_samples=32, random_state=1)
boot
```

The scikit-learn library provides an implementation that will create a single bootstrap sample of a dataset.

The resample() scikit-learn function can be used. It takes as arguments the data array, whether or not to sample with replacement, the size of the sample, and the seed for the pseudorandom number generator used prior to the sampling.

The bootstrap setup is as follows:

1. x1, x2, . . . , xn is a data sample drawn from a distribution F.
2. u is a statistic computed from the sample.
3. F∗ is the empirical distribution of the data (the resampling distribution). ∗ ∗ ∗ 4. x1, x2, . . . , x is a resample of the data of the same size as the original sample n ∗ 5. u is the statistic computed from the resample.

Then the bootstrap principle says that

1. F∗ ≈ F.
2. The variation of u is well-approximated by the variation of u .

Our ∗ real interest is in point 2: we can approximate the variation of u by that of u . We will exploit this to estimate the size of confidence intervals.

Why the resample is the same size as the original sample?

This is straightforward: the variation of the statistic u will depend on the size of the sample.If we want to approximate this variation we need to use resamples of the same size.

For example, we can create a bootstrap that creates a sample with replacement with 4 observations and uses a value of 1 for the pseudorandom number generator.

Then create a function called bootstrap to evaluate the bootstrap estimates based on number of iterations.

def bootstrap(d, iterations, func=identity, func_axis=None, dtype=None):

Parameters
----
a : array_like
Array containing numbers computation is desired.
If `a` is not an array, a conversion is attempted.
iterations: int
Number of bootstrap iterations to execute.
func: callable function
func_axis: integer, optional
Axis of the array that is used to separate different independent measurements for usage in a function with more than one argument.
The shape of this axis must match the number of arguments of the given function func.
dtype : data-type, optional
Type to use in computing statistics.
For integer inputs, the default is `float64`; for floating point inputs, it is the same as the input dtype.

Now we will see rest of the function which takes desired measurements and evaluates the function for bootstrap means.

```
    n = __number_measurements(a, func_axis)
    bootstrap_values = [func(*(__array_mean_indices(a, numpy.random.randint(0, high=n, size=n),
func_axis=func_axis, dtype=dtype))) for i in range(iterations)]

    # Return the average value and the error of this averaged value
    return numpy.mean(bootstrap_values), math.sqrt(float(iterations)/float(iterations -
1))*numpy.std(bootstrap_values)
```

We will see some examples you use to see the output of such function.

```
    >>> a = numpy.random.normal(loc=5.0, scale=2.0, size=1000)
    >>> mean_a, error_a = bootstrap(a, 100)
    >>> (mean_a > 4.9, mean_a < 5.1)
    (True, True)
    >>> (error_a > 2.0/math.sqrt(1000 - 1) - 0.01, error_a < 2.0/math.sqrt(1000 - 1) + 0.01)
    (True, True)
```

In our case our generated output looks like this:

```
bootstrap(boot.values, 100, func=identity, func_axis=None, dtype=None)
(20.215718750000004, 1.1566078697306692)
```

Here, 20.21 is the mean of the sample boot and 1.15 is the error of this mean compared to sample.

We compute CV value i.e coefficient of variation as following:

```
CV = np.sqrt(np.var(boot))/np.mean(boot)
print(CV)

#Another way to obtain coefficient of variation is shown below:
b_cov = scipy.stats.variation(boot)
print(b_cov)
>>>mpg    0.295264 dtype: float64
```

The we calculate the bias of our sample by formula = (sample mean - CV)/N

```
a= np.mean(boot)
N=32
bias =(a - CV)/N
print(bias)
```

>>>mpg 0.618605 dtype: float64

Then we calculate the std. Error (se) as: Std deviation(sample) / N
n=32
se = np.std(boot) / n
print("Std error of this sample is:", se)
>>>Std error of this sample is: mpg 0.185376
dtype: float64

Now we will see our function running for 100 iterations and mean and error value it evaluates as follows:
mean_a, error_a = bootstrap(boot.values, 100)
print(mean_a,error_a)
>>>19.9450625 1.047244740978112

This shows that we dont have any values greater than 34 or less than 10 in our dataset.
(mean_a > 34, mean_a < 10)
>>>(False, False)
(error_a > 2.0/math.sqrt(1000 - 1) - 0.01, error_a < 2.0/math.sqrt(1000 - 1) + 0.01)
>>>(True, False)

CODE:-

```
import pandas as pd
import numpy as np
import math
from sklearn.metrics import f1_score
from sklearn.model_selection import train_test_split
import matplotlib as mp
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
import scipy.stats

data = pd.read_csv(r"C:\Users\anmol\Downloads\mtcars.csv")
## r before your normal string helps it to convert normal string to raw string


# Summary statistics for the dataframe

data.describe()

print ("DataFrame Index: ", data.index)
print(data.values)
```

```
# Sort your dataframe
data.sort_values(by =['mpg','Cars'], ascending=[True,True])
# Resampling from our dataset
from sklearn.utils import resample
boot = resample(data.iloc[:,1:2], replace=False, n_samples=32, random_state=1)
print(boot)
import math
import numpy
import numpy.random


def __array_mean_indices(a, indices, func_axis=None, dtype=None):

    if func_axis == None:
        return (numpy.mean(a.flat[indices], dtype=dtype), )
    else:
        return tuple(numpy.mean(numpy.reshape(numpy.take(a, [j,], axis=func_axis),
-1)[indices]) for j in range(a.shape[func_axis]))

def __number_measurements(a, func_axis=None):
    """ Calculates the number of measurements of an array from the array and the function
axis.
    """
    if func_axis == None:
        return a.size
    else:
        return a.size / a.shape[func_axis]

def identity(x):
    """
    Identity function used as default function in the resampling methods.

    """
    return x
def bootstrap(a, iterations, func=identity, func_axis=None, dtype=None):

    # Calculate the number of measurements
    n = __number_measurements(a, func_axis)
    # Evaluate the function on the bootstrap means
    bootstrap_values = [func(*(__array_mean_indices(a, numpy.random.randint(0, high=n,
size=n), func_axis=func_axis, dtype=dtype))) for i in range(iterations)]

    # Return the average value and the error of this averaged value
    return numpy.mean(bootstrap_values), math.sqrt(float(iterations)/float(iterations -
1))*numpy.std(bootstrap_values)
    print (numpy.std(bootstrap_values))
>>>>
bootstrap(boot.values, 100, func=identity, func_axis=None, dtype=None)
```

```
>>>>
z = np.mean(boot.values)
v = np.std(boot.values)
print("The sample mean and std deviation is:->",z,v)
>>>>
CV = np.sqrt(np.var(boot))/np.mean(boot)
print(CV)
#Another way to obtain coeffiecient of variation is shown below:
b_cov = scipy.stats.variation(boot)
>>>>
a= np.mean(boot)
N=32
bias =(a - CV)/N
print(bias)
>>>>
n=32
se =  np.std(boot) / n
print("Std error of this sample is:", se)
>>>>
mean_a, error_a = bootstrap(boot.values, 100)
print(mean_a,error_a)
>>>>
(mean_a > 34, mean_a < 10)
>>>>
(error_a > 2.0/math.sqrt(1000 - 1) - 0.01, error_a < 2.0/math.sqrt(1000 - 1) + 0.01)
print(b_cov)
```

Now we have computed statistics using bootstrap so we will proceed to jackknife computation of estimates.

Other properties of bootstrap:

- *Typical values of B, the number of bootstrap samples, are ≥ 200 for standard error estimation.*
- *Bootstrap methods can also assess more complicated accuracy measures, like biases, prediction errors, and confidence intervals.*
- *Bootstrap confidence intervals add another factor of 10 to the computational Burden.*

The payoff for heavy computation:

- *An increase in the statistical problems that can be analyzed.*
- *A reduction in the assumption of the analysis.*
- *The elimination of the routine but tedious theoretical calculations usually associated with accuracy assessment.*

b)

CODE:-

```
from scipy.special import erfinv
import numpy as np
from astropy.stats import jackknife_resampling
from astropy.stats import jackknife_stats
test_statistic = np.mean
d = boot.values
import numpy as np
from astropy.stats import jackknife_resampling


resamples = jackknife_resampling(d)
resamples
x = scipy.stats.variation

def jackknife_resampling(data):
    n = data.shape[0]
    assert n > 0, "data must contain at least one measurement"

    resamples = np.empty([n, n-1])

    for i in range(n):
        resamples[i] = np.delete(data, i)

    return resamples


def jackknife_stats(data, statistic, conf_lvl=0.95):
    stat_data = statistic(data)
    jack_stat = np.apply_along_axis(statistic, 1, resamples)
    mean_jack_stat = np.mean(jack_stat, axis=0)
    # jackknife bias
    bias = (n-1)*(mean_jack_stat - stat_data)

    # jackknife standard error
    std_err = np.sqrt((n-1)*np.mean((jack_stat - mean_jack_stat)*(jack_stat -
                                mean_jack_stat), axis=0))

    # bias-corrected "jackknifed estimate"
    estimate = stat_data - bias
    # jackknife confidence interval
```

```
    assert (conf_lvl > 0 and conf_lvl < 1), "confidence level must be in (0,1)."
    z_score = np.sqrt(2.0)*erfinv(conf_lvl)
    conf_interval = estimate + z_score*np.array((-std_err, std_err))

    return estimate, bias, std_err, conf_interval

jackknife_stats(resamples,np.std, conf_lvl=0.95)
jackknife_stats(resamples,np.mean, conf_lvl=0.95)
plt.hist(x, 25, histtype='step');
```

First we will generate samples from dataset by resampling. Jackknife resampling is a technique to generate 'n' deterministic samples of size 'n-1' from a measured sample of size 'n'. Basically, the i-th sample, (1<=i<=n), is generated by means of removing the i-th measurement of the original sample. Like the bootstrap resampling, this statistical technique finds applications in estimating variance, bias, and confidence intervals.
Now to calculate jackknife estimations we create a function whose parameters are explained below:

def jackknife_stats(data, statistic, conf_lvl=0.95):

Parameters
    ----
    data : numpy.ndarray
        Original sample (1-D array).
    statistic : function
        Any function (or vector of functions) on the basis of the measured
        data, e.g, sample mean, sample variance, etc. The jackknife estimate of
        this statistic will be returned.
    conf_lvl : float, optional
        Confidence level for the confidence interval of the Jackknife estimate.
        Must be a real-valued number in (0,1). Default value is 0.95.

    Returns
    ----
    estimate : numpy.float64 or numpy.ndarray
        The i-th element is the bias-corrected "jackknifed" estimate.

    bias : numpy.float64 or numpy.ndarray

The i-th element is the jackknife bias.

std_err : numpy.float64 or numpy.ndarray
The i-th element is the jackknife standard error.

conf_interval : numpy.ndarray
If ``statistic`` is single-valued, the first and second elements are
the lower and upper bounds, respectively. If ``statistic`` is
vector-valued, each column corresponds to the confidence interval for
each component of ``statistic``. The first and second rows contain the
lower and upper bounds, respectively.

Parameters
---
data : numpy.ndarray
Original sample (1-D array) from which the jackknife resamples will be
generated.

Returns
---
resamples : numpy.ndarray
The i-th row is the i-th jackknife sample, i.e., the original sample
with the i-th measurement deleted.

Now how to use my code to generate output will be explained via examples to show their usage and implemenation.

Examples
---
1. Obtaining Jackknife resamples:

```
>>> import numpy as np
>>> from astropy.stats import jackknife_resampling
>>> from astropy.stats import jackknife_stats
>>> data = np.array([1,2,3,4,5,6,7,8,9,0])
>>> resamples = jackknife_resampling(data)
>>> resamples
array([[ 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  0.],
       [ 1.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  0.],
       [ 1.,  2.,  4.,  5.,  6.,  7.,  8.,  9.,  0.],
```

```
       [ 1.,  2.,  3.,  5.,  6.,  7.,  8.,  9.,  0.],
       [ 1.,  2.,  3.,  4.,  6.,  7.,  8.,  9.,  0.],
       [ 1.,  2.,  3.,  4.,  5.,  7.,  8.,  9.,  0.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  8.,  9.,  0.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  9.,  0.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  0.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.]])
>>> resamples.shape
(10, 9)
```

2. Obtain Jackknife estimate for the mean, its bias, its standard error,
and its 95% confidence interval:

```
>>> test_statistic = np.mean
>>> estimate, bias, stderr, conf_interval = jackknife_stats(data, test_statistic, 0.95)
>>> estimate
4.5
>>> bias
0.0
>>> stderr
0.95742710775633832
>>> conf_interval
array([ 2.62347735,  6.37652265])
```

3. Example for two estimates

```
>>> test_statistic = lambda x: (np.mean(x), np.var(x))
>>> estimate, bias, stderr, conf_interval = jackknife_stats(data, test_statistic, 0.95)
>>> estimate
array([ 4.5       ,  9.16666667])
>>> bias
array([ 0.        , -0.91666667])
>>> stderr
array([ 0.95742711,  2.69124476])
>>> conf_interval
array([[  2.62347735,   3.89192387],
       [  6.37652265,  14.44140947]])
```

IMPORTANT: Note that confidence intervals are given as columns

"""

Now from the code we can see

```
# jackknife bias
    bias = (n-1)*(mean_jack_stat - stat_data)
```

Is used to generate bias of our sample distribution.

Then also, we calculate std error using following:

```
    # jackknife standard error
    std_err = np.sqrt((n-1)*np.mean((jack_stat - mean_jack_stat)*(jack_stat -
                                mean_jack_stat), axis=0))
```

Then we calculate and show corrected bias value using

```
 # bias-corrected "jackknifed estimate"
    estimate = stat_data - bias
```

The we also calculated Confidence Interval using:

```
# jackknife confidence interval
    assert (conf_lvl > 0 and conf_lvl < 1), "confidence level must be in (0,1)."
    z_score = np.sqrt(2.0)*erfinv(conf_lvl)
    conf_interval = estimate + z_score*np.array((-std_err, std_err))

    return estimate, bias, std_err, conf_interval
```

Definition: The delete-1 **Jackknife Samples** are selected by taking the original data vector and deleting one observation from the set. Thus, there are n unique Jackknife samples, and the ith Jackknife sample vector is defined as:

$$\mathbf{X}_{[i]} = \{X_1, X_2, \ldots, X_{i-1}, X_{i+1}, \ldots, X_{n-1}, X_n\}$$

This procedure is generalizable to k deletions, which is discussed further below.

The ith **Jackknife Replicate** is defined as the value of the estimator $s(\cdot)$ evaluated at the ith Jackknife sample.

$$\hat{\theta}_{(i)} := s(\mathbf{X}_{[i]})$$

The Jackknife Standard Error is defined

$$SE(\hat{\theta})_{jack} = \left\{ \frac{n-1}{n} \sum_{i=1}^{n} (\hat{\theta}_{(i)} - \hat{\theta}_{(\cdot)})^2 \right\}^{1/2},$$

where $\hat{\theta}_{(\cdot)}$ is the empirical average of the Jackknife replicates:

$$\hat{\theta}_{(\cdot)} = \frac{1}{n} \sum_{i=1}^{n} \hat{\theta}_{(i)}$$

(Source: http://people.bu.edu/aimcinto/jackknife.pdf)

The (n − 1)/n factor in the formula above looks similar to the formula for the standard error of the sample mean, except that there is a quantity (n − 1) included in the numerator. As motivation for this estimator, I consider the case that does not actually need any resampling methods: that of the sample mean. Here, the Jackknife estimator above is an unbiased estimator of the variance of the sample mean.

The **Jackknife Bias** is defined as

$$\widehat{\text{bias}}_{jack} = (n-1)(\hat{\theta}_{(\cdot)} - \hat{\theta}),$$

where $\hat{\theta}$ is the estimator taking the entire sample as argument. Jackknife Bias is just the average of the deviations of the replicates, which are sometimes called *Jackknife Influence Values*, multiplied by a factor $(n-1)$. The bias of the sample mean is 0, so I cannot take the function $s(\cdot) = \bar{x}$ to get an idea of what the multiplier should be, as I did previously for the Jackknife SE. Instead, I consider as an estimator the uncorrected variance of the sample:

$$\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

If I take as my estimator $\hat{\theta}$ the (biased) sample variance, I have that its bias, that is, $\mathrm{E}[\hat{\theta} - \theta]$, is equal to $-\sigma^2/n$. If I use the Jackknife bias as an estimate for the bias of my estimator, and I have that my estimator $\hat{\theta}$ is equal to the uncorrected sample variance, then the Jackknife bias formula reduces to $-S^2/n$, where S^2 is now the regular, corrected, unbiased estimator of sample variance. Thus, the bias here is constructed from a heuristic notion to emulate the bias of the uncorrected sample variance.

The above synopsis gave a rationale based on familiar sample-based estimators. Here is another justification: Assume that for any fixed n the expected value of an estimator is the parameter estimand plus some bias term, call it $b_1(\theta)/n$. Then, as the average of the Jackknife replicates has $(n-1)$ terms, the expected value of the average is

$$E[\hat{\theta}_{(\cdot)}] = \frac{1}{n}\sum_{i=1}^{n} E[\hat{\theta}_{(i)}] = \theta + \frac{b_1(\theta)}{n-1}$$

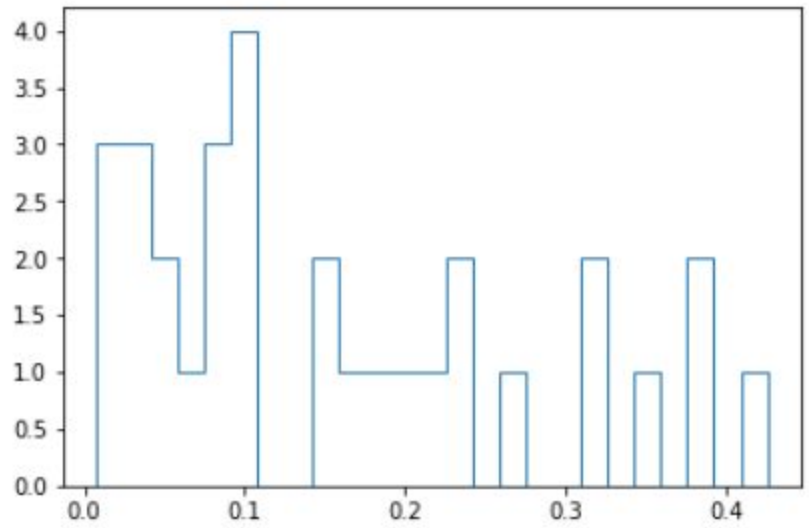
From this observation it follows that the bias of the Jackknife replicates estimator is

$$E[\hat{\theta} - \hat{\theta}_{(\cdot)}] = \theta + \frac{b_1(\theta)}{n} - \theta - \frac{b_1(\theta)}{n-1} = \frac{b_1(\theta)}{n(n-1)}$$

(Source: http://people.bu.edu/aimcinto/jackknife.pdf)

Hence if we multiply this difference above by (n − 1), we get an unbiased estimator of the bias of our original estimator.
The Jackknife bias is asymptotically smaller than the bias of any given biased estimator. Another construction sometimes used in Jackknife estimation is the “pseudovalue,” and can be seen as a bias-corrected version of the estimator. The scheme is to treat the jackknife pseudovalues as if they were independent random variables. Finally we plotted a histogram plot for all coefficient of variation values obtained from our dataset.

```
plt.hist(x, 25, histtype='step');
```



The following code of line will generate JackKnife CI at 95% based on mean of the samples.

```
jackknife_stats(resamples,np.mean, conf_lvl=0.95)
(20.090625000000223,
 -2.2026824808563106e-13,
 1.065423959372812,
 array([18.00243241, 22.17881759]))
```

The following code of line will generate JackKnife CI at 95% based on standard deviation of the samples.

```
jackknife_stats(resamples,np.std, conf_lvl=0.95)
(6.075658514972499,
 -0.143628962671281,
 0.753809543883037,
 array([4.59821896, 7.55309807]))
```

The following code of line will generate JackKnife CI at 95% based on coefficient of variation of the samples.

```
jackknife_stats(d,x, conf_lvl=0.95)
(array([0.30256309]),
 array([-0.00729953]),
 0.033086437178220435,
 array([0.23771487, 0.36741132]))
```

c)

Now to calculate the Confidence Intervals we did implementation of a percentile bootstrap interval: Generate some 'B' bootstrap samples of the original data and calculate B statistics from these samples. Then take the alpha/2 and 1-alpha/2 quantiles of the statistic to form a confidence interval of level alpha. For CI = 95% we need for an alpha = .05 / 95% confidence interval so we have to look at the .025 and .975 quantiles of the bootstrap statistics.

```
def mean_confidence_interval(sample, confidence=0.95):
    a = 1.0 * np.array(sample)
    n = len(d)
    m, se = np.mean(d), scipy.stats.sem(d)
    h = se * scipy.stats.t.ppf((1 + confidence) / 2., n-1)
    return m, m-h, m+h
>>>mean_confidence_interval(resamples, confidence=0.95)
```

Scipy.stats.sem is used to Combined with the mean, the SEM enables you to establish a range around a mean that the majority of any future replicate experiments will most likely fall within.

pandas DataFrames don't have methods like SEM built in, but since DataFrame rows/columns are treated as lists, you can use any NumPy/SciPy method you like on them.

Scipy.stats.t.ppf gives us t-continuous random variable distributions which is Percent point function (inverse of cdf — percentiles).

```
>>>np.percentile(resamples, 0.95)
10.4
>>>scipy.stats.mstats.mquantiles (resamples,0.95)
array([32.4])
>>>scipy.stats.mstats.mquantiles (resamples,0.05)
array([10.4])
```

These above code lines gives you percentile/quantile C.I of the sample data.(resamples / boot)

Question 2:

CODE:-

```
import pandas as pd
import numpy as np
charlie = pd.read_csv("charlie1.csv")
X = charlie[['z1','z2']]
y = charlie['Data']

y_out = np.array(y[20:])
x_out = np.array(X[20:])

y = y[0:20]
X = X[0:20]

X = np.array(X)
y = np.array(y)

def Kernel(x, y, sigma):
    return np.exp(-np.linalg.norm(x-y)**2 / ( (sigma ** 2)))

def Gram_Matrix(x):
    K = np.zeros((len(x),len(x)))
    for i in range(0, len(x)):
        for j in range(0, len(x)):
            K[i, j] = Kernel(x[i], x[j], sigma)

    return K

def H(x):
    mat = np.zeros((len(x), len(x)))
    mat[0:len(x), 0:len(x)] = Gram_Matrix(x) + np.eye(len(x))/2*C
    return mat

def alpha():
#     a = 0.5*np.dot(np.linalg.inv(H_mat),(k + np.dot((2-np.dot(np.dot(e.T, np.linalg.inv(H_mat)),
k))/(np.dot(np.dot(e.T, np.linalg.inv(H_mat)), e)),e)))
    p1 = np.dot(np.dot(np.linalg.inv(H_mat), e.T),k)
    p2 = np.dot(np.dot(np.linalg.inv(H_mat), e.T), e)
    p3 = (2-p1)/p2
    p3 = k + np.dot(p3, e)
    a = 0.5*np.dot(np.linalg.inv(H_mat),p3)
    return a
e = np.ones(len(X))
k = np.zeros((len(X)))
```

```
sigma = 0.125
C = 1
for j in range(0, len(X)):
    k[j] = Kernel(X[j], X[j], sigma)

H_mat = H(X)
al = alpha()

def R_square():
    p1 = 0
    p2 = 0
    total = 0
    for s in range(0, len(X)):
        k = Kernel(X[s], X[s], sigma)
        for j in range(0, len(X)):
            p1 = p1 + al[j]*Kernel(X[s], X[j], sigma)
            for l in range(0, len(X)):
                p2 = p2 + al[j]*al[l]*Kernel(X[j], X[l], sigma)
        total = total + (k - 2 * p1 + p2)
    final = total/len(X)
    return final

final = R_square()

def classification(x):
    t_out = []
    t_in = []
    p = 0
    p1 = 0
    for z in range(0, len(x)):
        k = Kernel(x[z], x[z], sigma)
        for j in range(0, len(X)):
            p = p + al[j]*Kernel(x, X[j], sigma)
            for l in range(0, len(X)):
                p1 = p1 + al[j]*al[l]*Kernel(X[j], X[l], sigma)
        d = k - 2*p + p1
        if d <= final:
            t_in.append(x[z])
        else:
            t_out.append(x[z])

    return t_out, t_in

t_out, t_in = classification(x_out)
```

We have a dataset “Charlie.csv” which has Data Column having values 1 and -1 which are Original and New respectively with x1, x2, x3, x4 as other columns. We consider x = { x1, x2, x3, x4} and y = {Data} from the dataset “Charlie.csv”.
We generated our custom gaussian kernel by using following equation:

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{y}||^2}{\sigma^2}\right)$$

We defined and evaluated gaussian kernel function

```
def Kernel(x, y, sigma):
    return np.exp(-np.linalg.norm(x-y)**2 / ( (sigma ** 2)))
```

Using gram matrix,

```
def Gram_Matrix(x):
    K = np.zeros((len(x),len(x)))
    for i in range(0, len(x)):
        for j in range(0, len(x)):
            K[i, j] = Kernel(x[i], x[j], sigma)

    return K
```

Then we evaluated the H(x) equation given in image below:

```
def H(x):
    mat = np.zeros((len(x), len(x)))
    mat[0:len(x), 0:len(x)] = Gram_Matrix(x) + np.eye(len(x))/2*C
    return mat
```

Then according to the equation of alpha the function alpha is created:

```
def alpha():
#     a = 0.5*np.dot(np.linalg.inv(H_mat),(k + np.dot((2-np.dot(np.dot(e.T, np.linalg.inv(H_mat)),
k))/(np.dot(np.dot(e.T, np.linalg.inv(H_mat)), e)),e)))
    p1 = np.dot(np.dot(np.linalg.inv(H_mat), e.T),k)
    p2 = np.dot(np.dot(np.linalg.inv(H_mat), e.T), e)
    p3 = (2-p1)/p2
    p3 = k + np.dot(p3, e)
    a = 0.5*np.dot(np.linalg.inv(H_mat),p3)
    return a
e = np.ones(len(X))
k = np.zeros((len(X)))
```

Here np.dot is dot product of matrices, linalg is linear algebraic function and e is an identity matrix.

The solutions are given by:

$$\boldsymbol{\alpha} = \frac{1}{2}\mathbf{H}^{-1}\left(\mathbf{k} + \frac{2 - \mathbf{e}'\mathbf{H}^{-1}\mathbf{k}}{\mathbf{e}'\mathbf{H}^{-1}\mathbf{e}}\mathbf{e}\right). \tag{2}$$

where $\mathbf{H} = \mathbf{K} + \frac{1}{2C}\mathbf{I}_N$, $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_N)'$ represents the column vector of Lagrange multipliers. The matrix $\mathbf{K}$ is the Gram matrix and has entries $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, $\mathbf{e} = (1, \ldots, 1)'$, $\mathbf{I}_N$ denotes the identity matrix, and $\mathbf{k}$ denotes a vector with entries $k_j = k(\mathbf{x}_j, \mathbf{x}_j), j = 1, \ldots, N$.

For classification, a test vector $\mathbf{z}$ is in the target class if

$$d_z = K(\mathbf{z}, \mathbf{z}) - 2\sum_{j=1}^{N} \alpha_j K(\mathbf{z}, \mathbf{x}_j) + \sum_{j,l=1}^{N} \alpha_j \alpha_l K(\mathbf{x}_j, \mathbf{x}_l) \leq R^2. \tag{3}$$

and the radius R is found as follows:

$$R^2 = \frac{1}{N}\sum_{s=1}^{N}\left(K(\mathbf{x}_s, \mathbf{x}_s) - 2\sum_{j=1}^{N} \alpha_j K(\mathbf{x}_s, \mathbf{x}_j) + \sum_{j,l=1}^{N} \alpha_j \alpha_l K(\mathbf{x}_j, \mathbf{x}_l)\right). \tag{4}$$

Then we determined radius function using the given equation as above:

```
def R_square():
    p1 = 0
    p2 = 0
    total = 0
    for s in range(0, len(X)):
        k = Kernel(X[s], X[s], sigma)
        for j in range(0, len(X)):
            p1 = p1 + al[j]*Kernel(X[s], X[j], sigma)
            for l in range(0, len(X)):
                p2 = p2 + al[j]*al[l]*Kernel(X[j], X[l], sigma)
        total = total + (k - 2 * p1 + p2)
    final = total/len(X)
    return final
```

Now we perform classification on the dataset where we use 75% of the dataset as training data and 25% as testing data. So this is ensured by classification function as shown below:

```
def classification(x):
    t_out = []
    t_in = []
    p = 0
    p1 = 0
    for z in range(0, len(x)):
        k = Kernel(x[z], x[z], sigma)
        for j in range(0, len(X)):
            p = p + al[j]*Kernel(x, X[j], sigma)
            for l in range(0, len(X)):
                p1 = p1 + al[j]*al[l]*Kernel(X[j], X[l], sigma)
        d = k - 2*p + p1
        if d <= final:
            test_in.append(x[z])
        else:
            test_out.append(x[z])

    return test_out, test_in

import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm

clf = svm.OneClassSVM(kernel = 'rbf', gamma = 'auto')
clf.fit(t_out, t_in)

clf.predict(t_out)
n_error_outliers = t_out[t_out == 1].size
print("Number of errors = ",n_error_outliers,"/",y_out.size)
#classification rate
rate = n_error_outliers/y_out.size
print("Classification rate = ",100*(1-rate),"%")

t_out, t_in = classification(x_out)
df = pd.DataFrame(t_out)
import seaborn as sns
sns.pairplot(df)

df = pd.DataFrame(t_out)
import seaborn as sns
sns.pairplot(df)

l = df.iloc[0:,1:2]
x = np.linspace(0, 10, 10)
y = l
plt.plot(t_out, y_out, 'o', color='black');
```

print("This shows that all t_out i.e outliers and y_out = New points are detected as anomaly and shown below at -1,0 ")
print("Rest all points are not shown as they appear to be inside the circle of radius = final =0.47 and are not counted as anomaly i.e why we have t_in as empty set for any -1 value.")

In classification process our aim is to process and compute each data point and check whether its +1 - ‘Original’ or -1 ‘New’ and whether they fit inside the circle of radius final = 0.47. The points +1 are inside the circle and -1 are outside the circle as we treated ‘Original’ as one class SVM so we got all noble points in t_in and all outliers in t_out.

```
>>>t_out
[array([0.074196, 0.239359]),
 array([-1.51756, -0.21121]),
 array([ 1.408476, -0.87591 ]),
 array([ 6.298001, -3.67398 ]),
 array([ 3.802025, -1.99584 ]),
 array([ 6.490673, -2.73143 ]),
 array([ 2.738829, -1.37617 ]),
 array([ 4.958747, -3.94851 ]),
 array([ 5.678092, -3.85838 ]),
 array([ 3.369657, -2.10878 ])]

>>>import matplotlib.pyplot as plt
import matplotlib.font_manager
from sklearn import svm

clf = svm.OneClassSVM(kernel = 'rbf', gamma = 'auto')
clf.fit(t_out, t_in)

>>>clf.predict(t_out)
>>>n_error_outliers = t_out[t_out == 1].size
print("Number of errors = ",n_error_outliers,"/",y_out.size)
#classification rate
>>>rate = n_error_outliers/y_out.size
print("Classification rate = ",100*(1-rate),"%")
```

Number of errors = 2 / 10
Classification rate = 80.0 %

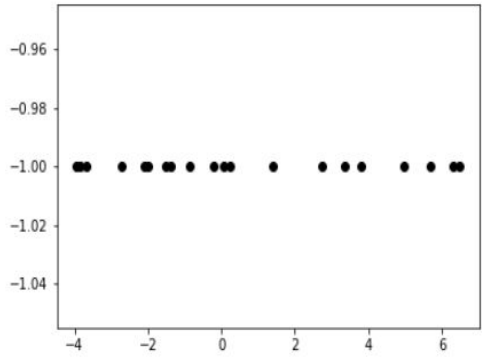
As we can see from the output of the code we got 80% classification rate and -1 as outliers which satisfies the problem given.Rest all points are not shown as they appear to be inside the circle of radius = final =0.47 and are not counted as anomaly i.e why we have t_in as an empty set for any -1 value.

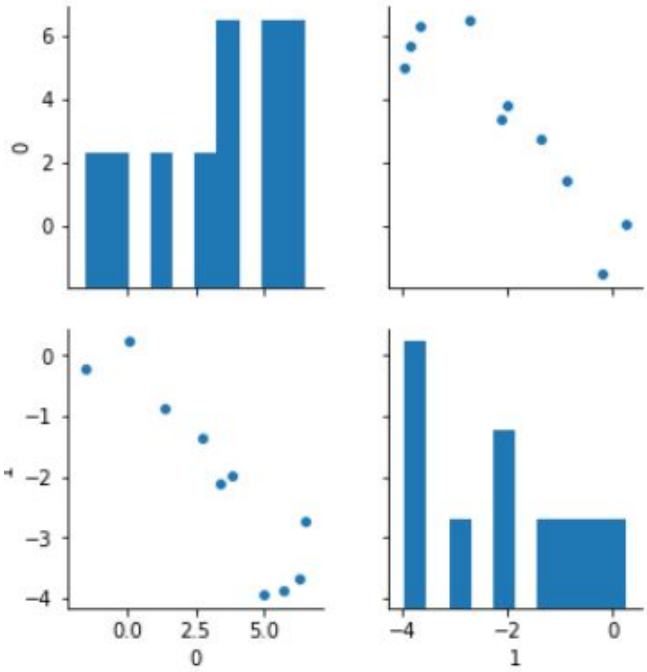
```
df = pd.DataFrame(t_out)
import seaborn as sns
sns.pairplot(df)

l = df.iloc[0:,1:2]
x = np.linspace(0, 10, 10)
y = l
plt.plot(t_out, y_out, 'o', color='black');

print("This shows that all t_out i.e outliers  and y_out = New points are detected as anomaly and shown below at -1,0 ")
print("Rest all points are not shown as they appear to be inside the circle of radius = final =0.47 and are not counted as anomaly i.e why we have t_in as empty set for any -1 value.")
```

This shows that all t_out i.e outliers and y_out = New points are detected as anomaly and shown below at -1,0
Rest all points are not shown as they appear to be inside the circle of radius = final =0.47 and are not counted as anomaly i.e why we have t_in as empty set for any -1 value.





Question 3:

CODE:-

```
import numpy as np
import scipy.stats as st
import seaborn as sns
import matplotlib.pyplot as plt
import math

i = 0
k = 0
n = 1000
z = np.random.uniform(0,1,n)
while i<n:
    u = np.random.uniform(0,1,1)
    y = np.random.exponential(scale=0.001,size = 1)
    k = k+1
    if u >= np.sqrt(2/math.pi)*np.exp(-y*2/2):
        i = i
    else:
        z[i] = y*(u < np.sqrt(2/math.pi)*np.exp(-y*2/2))
        i += 1

print(i, k)

>>>1000 1242

# P= P(Y accepted) =1/c
P=i/k
c = 1/P
print("Bounding Constant is c:", c)

>>>Bounding Constant is c: 1.242


sns.distplot(z, hist = True, kde = True)
plt.show()
```

Acceptance-Rejection method

Denote the density of X by f . This method requires a function g that majorizes f ,
g(x) ≥ f (x)
for all x. Now g will not be a density, since
c = {-∞, ∞}g(x)dx ≥ 1.
Assume that c < ∞. Then h(x) = g(x)/c is a density. Algorithm:

1. Generate Y having density h;
2. Generate U from U(0, 1), independent of Y ;
3. If U ≤ f (Y)/g(Y), then set X = Y ; else go back to step 1.
The random variable X generated by this algorithm has density f .

Validity of the Acceptance-Rejection method
Note
P(X ≤ x) = P(Y ≤ x|Y accepted).
Now,
P(Y ≤ x, Y accepted) ={x,−∞}f (y)/g(y)*h(y)dy =1/c*{x,−∞}f (y)dy,
and thus, letting x → ∞ gives
P(Y accepted) =1/c.
Hence,
P(X ≤ x) =P(Y ≤ x, Y accepted)/P(Y accepted)={x,−∞}f (y)dy.

c=sqrt(2e/π)≈1.32.
(Source ="https://www.scss.tcd.ie/Brett.Houlding/Domain.sites2/sslides5.pdf")

Answers:
a) Calculate the optimal constant C for acceptance rejection as a function of λ.
"""
print("The expected number of iterations of the algorithm required until an X is successfully generated is exactly the bounding constant C. In particular, we assume that the ratio f(x)/g(x) is bounded by a constant c > 0. And in practice we would want c as close to 1 as possible.")
print("C =", c)

"""
b) What is the best parameterλ∈(0,∞) you could use for the proposals.

"""
print("λ = scaling parameter i.e scale =0.001 , I have observed that smaller the scale value goes more optimal exponential distribution is generated. So in this case out of all scale values I would consider scale = 0.001 as best parameter for our λ.")
print(scale)
"""
c) Using the optimal λ, how many of the generated exponentially dis-tributed proposals do you expect to
accept (as a percentage)?

"""
print("The percentage of accepted distributed proposals")
print(100-((k-i)/k)*100)

"""
d)Write Python codes to generate positive normals using the Accept-Reject Algorithm.

"""
print("The positive normal distribution values are plotted as follow: ")
sns.distplot(z, hist = True, kde = True)
plt.show()

Output:

The expected number of iterations of the algorithm required until an X is successfully generated is exactly the bounding constant C. In particular, we assume that the ratio f(x)/g(x) is bounded by a constant c > 0. And in practice we would want c as close to 1 as possible.
C = 1.26
λ = scaling parameter i.e scale =0.001 , I have observed that smaller the scale value goes more optimal exponential distribution is generated. So in this case out of all scale values I would consider scale = 0.001 as best parameter for our λ.
[1. 0.5 0.25 0.1 0.01 0.001]
The percentage of accepted distributed proposals
79.36507936507937
The positive normal distribution values are plotted as follow: