Problem statement 1

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np
#from sklearn import preprocessing, cross_validation
import pandas as pd
```

```
import numpy as np

np.random.seed(123)

allwalks = []

for i in range(250):
    randwalk = [0]
    for x in range(100):
        step = randwalk[-1]
        dice = np.random.randint(1,7)
        if dice <= 2 :
            step = max(0, step - 1)

        elif dice<=5:
            step += 1

        else:
            step = step + np.random.randint(1,7)

    print(step)
```

```
1
2
0
0
0
0
1
1
0
1
1
1
0
0
5
1
```

```
0
2
1
1
0
1
1
1
1
1
1
1
1
1
1
1
1
0
0
0
1
0
1
1
0
0
0
0
1
1
1
1
0
0
0
0
1
0
1
1
0
1
1
1
1
1
0
1
0
1
1
1
1
1
```

```
1
4
0
0
0
0
0
1
1
1
0
1
1
1
4
0
1
1
1
0
0
0
0
0
1
1
1
0
2
1
0
6
0
0
1
6
0
3
4
1
1
6
1
0
5
1
0
1
5
0
1
1
1
0
```

```
0
1
1
5
1
1
1
0
0
0
0
5
0
0
0
0
1
4
1
1
1
1
1
0
1
1
0
1
1
1
1
0
1
1
1
0
1
3
3
1
1
0
1
0
1
1
1
1
1
1
1
0
0
1
```

```
0
0
0
1
0
0
6
1
2
0
1
1
1
3
1
1
1
1
1
0
1
1
1
1
1
0
1
0
4
1
1
1
1
1
0
1
0
0
1
0
1
1
1
1
1
1
1
0
0
1
2
0
0
0
```

```
0
1
1
0
4
1
1
1
4
1
2
1
0
1
1
0
1
1
```

Problem statement 2

Random data for multiple linear regression

```
import numpy as np
import pandas as pd
import scipy
import random
from scipy.stats import norm
random.seed(1)
n_features = 4
X = []
for i in range(n_features):
  X_i = scipy.stats.norm.rvs(0, 1, 100)
  X.append(X_i)
#print(X)
eps = scipy.stats.norm.rvs(0, 0.25,100)
y = 1 + (0.4 * X[0]) + eps + (0.5 * X[1]) + (0.3 * X[2]) + (0.4 * X[3])
data_mlr = {'X0': X[0],'X1':X[1],'X2':X[2],'X3':X[3],'Y': y }
df = pd.DataFrame(data_mlr)
print(df.head())
print(df.tail())
print(df.info())
print(df.describe())
#df.to_csv('file1.csv')
```

```
          X0        X1        X2        X3         Y
0  1.229187 -0.272740 -2.976897  0.234832  0.514580
1 -0.504641  0.137005  1.243000  0.360801  1.094348
2 -1.716661 -0.621364 -0.874199 -0.667714 -0.531753
3 -0.401698 -0.171870  0.105726  0.719925  1.319178
4  2.338396 -0.389372  0.371695 -0.211898  2.015665
          X0        X1        X2        X3         Y
95  0.538347  0.947229  1.702177  1.827932  2.946796
96 -2.124628 -0.023599 -0.737292  0.959677  0.382169
97  0.690109 -1.404677  1.814179 -1.350687  0.695210
98 -0.473165  0.264512 -0.011203  2.202028  1.598639
99  0.272442  0.936071 -1.283808  1.368407  1.735643
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   X0      100 non-null    float64
 1   X1      100 non-null    float64
 2   X2      100 non-null    float64
 3   X3      100 non-null    float64
 4   Y       100 non-null    float64
dtypes: float64(5)
memory usage: 4.0 KB
None
               X0          X1          X2          X3           Y
count  100.000000  100.000000  100.000000  100.000000  100.000000
mean    -0.152297    0.030112   -0.063573    0.006363    0.978705
std      0.924978    0.975453    1.096232    1.060847    0.913494
min     -2.206992   -2.118134   -2.976897   -2.946925   -1.051042
25%     -0.729312   -0.621701   -0.698830   -0.674253    0.377647
50%     -0.158125    0.089349   -0.141087   -0.037818    0.964189
75%      0.498398    0.641858    0.574192    0.620150    1.549381
max      2.338396    3.159726    3.210134    2.292966    3.643752
```

Random data for logistic regression

```
n_features = 4
X = []
for i in range(n_features):
  X_i = scipy.stats.norm.rvs(0, 1, 100)
  X.append(X_i)
#print(X)
a1 = (np.exp(1 + (0.5 * X[0]) + (0.4 * X[1]) + (0.3 * X[2]) + (0.5 * X[3]))/(1 +
#print(a1)
y1 = []
for i in a1:
  if (i>=0.5):
    y1.append(1)
  else:
```

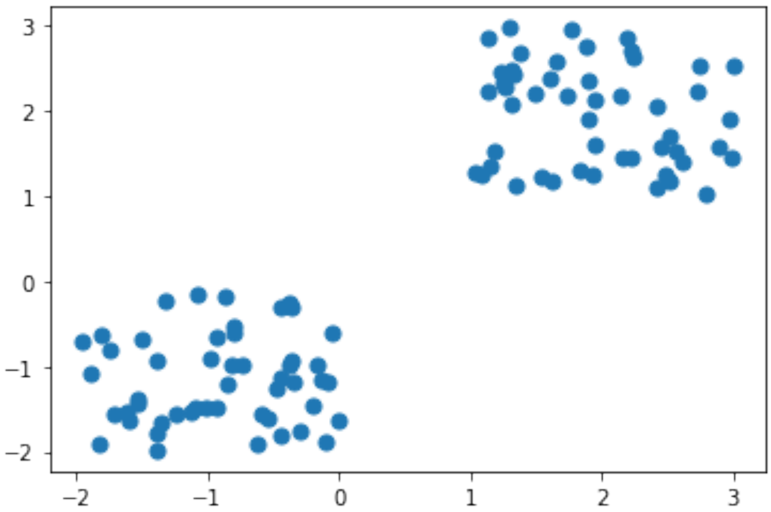
```
    y1.append(0)
#print(y1)
data_lr = {'X0': X[0],'X1':X[1],'X2':X[2],'X3':X[3],'Y': y1 }
df1 = pd.DataFrame(data_lr)
print(df.head())
print(df.tail())
print(df.info())
print(df.describe())
```

```
         X0        X1        X2        X3         Y
0  0.310326 -0.538983  0.522009 -0.630752  0.888074
1  0.026933  1.005510 -1.519784  0.317596  1.706252
2  1.454472 -1.948507  0.989502  1.673824  2.006770
3  0.299680 -1.090324 -0.968199  0.285824  0.376355
4  1.568637  0.042656 -0.204593  1.126121  2.629879
          X0        X1        X2        X3         Y
95 -0.408410  0.615359 -2.553963  1.017630  0.665036
96  1.271942  0.739803  1.451475 -2.180999  1.219121
97 -1.462029 -1.407781  0.657375  0.320367  0.309101
98 -0.355275 -1.795455  0.762725 -0.701842 -0.118037
99 -0.845194 -0.817530 -1.009229  0.328676 -0.005031
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   X0      100 non-null    float64
 1   X1      100 non-null    float64
 2   X2      100 non-null    float64
 3   X3      100 non-null    float64
 4   Y       100 non-null    float64
dtypes: float64(5)
memory usage: 4.0 KB
None
               X0          X1          X2          X3           Y
count  100.000000  100.000000  100.000000  100.000000  100.000000
mean     0.012668   -0.098384    0.002392    0.037210    1.007865
std      0.984979    1.056384    0.890788    1.027167    0.875303
min     -2.174584   -2.241255   -2.553963   -3.162631   -0.881631
25%     -0.761235   -0.900435   -0.565406   -0.652427    0.344661
50%     -0.013255   -0.008498   -0.023229    0.078830    1.035789
75%      0.724811    0.740714    0.684799    0.733959    1.709912
max      2.259437    2.192720    2.373255    2.320635    2.909347
```

Random data for K means clustering

```
X_a= -2 * np.random.rand(100,2)
X_b = 1 + 2 * np.random.rand(50,2)
X_a[50:100, :] = X_b
plt.scatter(X_a[ : , 0], X_a[ :, 1], s = 50)
```

```
plt.show()
data_kmeans = {'X0': X_a[:,0],'X1':X_a[:,1]}
df3 = pd.DataFrame(data_kmeans)
print(df.head())
print(df.tail())
print(df.info())
print(df.describe())
```




```
         X0        X1        X2        X3         Y
0  0.310326 -0.538983  0.522009 -0.630752  0.888074
1  0.026933  1.005510 -1.519784  0.317596  1.706252
2  1.454472 -1.948507  0.989502  1.673824  2.006770
3  0.299680 -1.090324 -0.968199  0.285824  0.376355
4  1.568637  0.042656 -0.204593  1.126121  2.629879
          X0        X1        X2        X3         Y
95 -0.408410  0.615359 -2.553963  1.017630  0.665036
96  1.271942  0.739803  1.451475 -2.180999  1.219121
97 -1.462029 -1.407781  0.657375  0.320367  0.309101
98 -0.355275 -1.795455  0.762725 -0.701842 -0.118037
99 -0.845194 -0.817530 -1.009229  0.328676 -0.005031
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   X0      100 non-null    float64
 1   X1      100 non-null    float64
 2   X2      100 non-null    float64
 3   X3      100 non-null    float64
 4   Y       100 non-null    float64
dtypes: float64(5)
memory usage: 4.0 KB
None
               X0          X1          X2          X3           Y
count  100.000000  100.000000  100.000000  100.000000  100.000000
mean     0.012668   -0.098384    0.002392    0.037210    1.007865
std      0.984979    1.056384    0.890788    1.027167    0.875303
min     -2.174584   -2.241255   -2.553963   -3.162631   -0.881631
25%     -0.761235   -0.900435   -0.565406   -0.652427    0.344661
50%     -0.013255   -0.008498   -0.023229    0.078830    1.035789
75%      0.724811    0.740714    0.684799    0.733959    1.709912
max      2.259437    2.192720    2.373255    2.320635    2.909347
```

Problem statement - 3

Linear Regression using gradient descent

```
X = df.iloc[:,0].values
#print(X)
y = df.iloc[:,4].values
b1 = 0
b0 = 0
l = 0.001
epochs = 100

n = float(len(X))
for i in range(epochs):
  y_p = b1*X + b0
  loss = np.sum(y_p - y1)**2
  d1 = (-2/n) * sum(X * (y - y_p))
  d0 = (-2/n) * sum(y - y_p)
  b1 = b1 - (l*d1)
  b0 = b0 - (l*d0)

print(b1,b0)
```

Logistic regression using Gradient descent

```
X1 = df1.iloc[:,0:4].values
y1 = df1.iloc[:,4].values

def sigmoid(Z):
  return 1 /(1+np.exp(-Z))

def loss(y1,y_hat):
  return -np.mean(y1*np.log(y_hat) + (1-y1)*(np.log(1-y_hat)))

W = np.zeros((4,1))
b = np.zeros((1,1))

m = len(y1)
lr = 0.001
for epoch in range(1000):
  Z = np.matmul(X1,W)+b
  A = sigmoid(Z)
  logistic_loss = loss(y1,A)
  dz = A - y1
  dw = 1/m * np.matmul(X1.T,dz)
  db = np.sum(dz)

  W = W - lr*dw
  b = b - lr*db

  if epoch % 100 == 0:
    print(logistic_loss)
```

Linear Regreesion using L1 Regularization

```
X = df.iloc[:,0].values
#print(X)
y = df.iloc[:,4].values
b1 = 0
b0 = 0
l = 0.001
epochs = 100
lam = 0.1

n = float(len(X))
for i in range(epochs):
  y_p = b1*X + b0
  loss = np.sum(y_p - y1)**2 + (lam * b1)
  d1 = (-2/n) * sum(X * (y - y_p)) + lam
  d0 = (-2/n) * sum(y - y_p)
  b1 = b1 - (l*d1)
  b0 = b0 - (l*d0)

print(b1,b0)
```

Linear Regreesion using L2 Regularization

```
X = df.iloc[:,0].values
#print(X)
y = df.iloc[:,4].values
b1 = 0
b0 = 0
l = 0.001
epochs = 100
lam = 0.1

n = float(len(X))
for i in range(epochs):
  y_p = b1*X + b0
  loss = np.sum(y_p - y1)**2 + ((lam/2) * b1)
  d1 = (-2/n) * sum(X * (y - y_p)) + (lam *b1)
  d0 = (-2/n) * sum(y - y_p)
  b1 = b1 - (l*d1)
  b0 = b0 - (l*d0)

print(b1,b0)
```

Logistic regression using L1 regualrization

```
X1 = df1.iloc[:,0:4].values
y1 = df1.iloc[:,4].values
lam = 0.1
def sigmoid(Z):
  return 1 /(1+np.exp(-Z))

def loss(y1,y_hat):
  return -np.mean(y1*np.log(y_hat) + (1-y1)*(np.log(1-y_hat))) + (lam * (np.sum(

W = np.zeros((4,1))
b = np.zeros((1,1))

m = len(y1)
lr = 0.001
for epoch in range(1000):
  Z = np.matmul(X1,W)+b
  A = sigmoid(Z)
  logistic_loss = loss(y1,A)
  dz = A - y1
  dw = 1/m * np.matmul(X1.T,dz) + lam
  db = np.sum(dz)

  W = W - lr*dw
  b = b - lr*db

  if epoch % 100 == 0:
    print(logistic_loss)
```

Logistic regression using L2 regualrization

```
X1 = df1.iloc[:,0:4].values
y1 = df1.iloc[:,4].values
lam = 0.1
def sigmoid(Z):
  return 1 /(1+np.exp(-Z))

def loss(y1,y_hat):
  return -np.mean(y1*np.log(y_hat) + (1-y1)*(np.log(1-y_hat))) + (lam * (np.sum(

W = np.zeros((4,1))
b = np.zeros((1,1))

m = len(y1)
lr = 0.001
for epoch in range(1000):
  Z = np.matmul(X1,W)+b
  A = sigmoid(Z)
  logistic_loss = loss(y1,A)
  dz = A - y1
  dw = 1/m * np.matmul(X1.T,dz) + lam * W
  db = np.sum(dz)

  W = W - lr*dw
  b = b - lr*db

  if epoch % 100 == 0:
    print(logistic_loss)
```

K Means Clustering Algorithm

```
class K_Means:
    def __init__(self, k=2, tol=0.001, max_iter=300):
```

```
        self.k = k
        self.tol = tol
        self.max_iter = max_iter

    def fit(self,data):

        self.centroids = {}

        for i in range(self.k):
            self.centroids[i] = data[i]

        for i in range(self.max_iter):
            self.classifications = {}

            for i in range(self.k):
                self.classifications[i] = []

            for featureset in X:
                distances = [np.linalg.norm(featureset-self.centroids[centroid])
                classification = distances.index(min(distances))
                self.classifications[classification].append(featureset)

            prev_centroids = dict(self.centroids)

            for classification in self.classifications:
                self.centroids[classification] = np.average(self.classifications

            optimized = True

            for c in self.centroids:
                original_centroid = prev_centroids[c]
                current_centroid = self.centroids[c]
                if np.sum((current_centroid-original_centroid)/original_centroid
                    print(np.sum((current_centroid-original_centroid)/original_c
                    optimized = False

            if optimized:
                break

    def predict(self,data):
        distances = [np.linalg.norm(data-self.centroids[centroid]) for centroid
        classification = distances.index(min(distances))
        return classification

colors = 10*["g","r","c","b","k"]
```

```
X = df3.iloc[:,0:2].values
clf = K_Means()
clf.fit(X)

for centroid in clf.centroids:
    plt.scatter(clf.centroids[centroid][0], clf.centroids[centroid][1],
                marker="o", color="k", s=150, linewidths=5)

for classification in clf.classifications:
    color = colors[classification]
    for featureset in clf.classifications[classification]:
        plt.scatter(featureset[0], featureset[1], marker="x", color=color, s=150
```

Problem Statement - 4

Linear Regression from scratch using OOPS

```
import numpy as np

class LinearRegressionModel():

    def __init__(self, dataset, learning_rate, num_iterations):
        self.dataset = np.array(dataset)
        self.b = 0
        self.m = 0
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.M = len(self.dataset)
        self.total_error = 0
```

```
    def apply_gradient_descent(self):
        for i in range(self.num_iterations):
            self.do_gradient_step()

    def do_gradient_step(self):
        b_summation = 0
        m_summation = 0
        for i in range(self.M):
            x_value = self.dataset[i, 0]
            y_value = self.dataset[i, 1]
            b_summation += (((self.m * x_value) + self.b) - y_value)
            m_summation += (((self.m * x_value) + self.b) - y_value) * x_value
        self.b = self.b - (self.learning_rate * (1/self.M) * b_summation)
        self.m = self.m - (self.learning_rate * (1/self.M) * m_summation)

    def compute_error(self):
        for i in range(self.M):
            x_value = self.dataset[i, 0]
            y_value = self.dataset[i, 1]
            self.total_error += ((self.m * x_value) + self.b) - y_value
        return self.total_error

    def __str__(self):
        return "Results: b: {}, m: {}, Final Total error: {}".format(round(self.

    def get_prediction_based_on(self, x):
        return round(float((self.m * x) + self.b), 2) # Type: Numpy float.

def main():
    school_dataset = np.genfromtxt(DATASET_PATH, delimiter=",")
    lr = LinearRegressionModel(school_dataset, 0.0001, 1000)
    lr.apply_gradient_descent()
    hours = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
    for hour in hours:
        print("Studied {} hours and got {} points.".format(hour, lr.get_predicti
    print(lr)

if __name__ == "__main__": main()
```

* **italicized text*Logistic Regression from scratch using OOPS**

```
class LogisticRegression:
  def __init__(self, learning_rate, num_iters, fit_intercept = True, verbose = F
    self.learning_rate = learning_rate
    self.num_iters = num_iters
    self.fit_intercept = fit_intercept
    self.verbose = verbose
  def __add_intercept(self, X):
    intercept = np.ones((X.shape[0],1))
    return np.concatenate((intercept,X),axis=1)
  def __sigmoid(self,z):
    return 1/(1+np.exp(-z))
  def __loss(self, h, y):
    return (-y * np.log(h) - (1-y) * np.log(1-h)).mean()

  def fit(self,X,y):
    if self.fit_intercept:
      X = self.__add_intercept(X)
    self.theta = np.zeros(X.shape[1])

    for i in range(self.num_iters):
      z = np.dot(X,self.theta)
      h = self.__sigmoid(z)
      gradient = np.dot(X.T,(h-y))/y.size

      self.theta -= self.learning_rate * gradient

      z = np.dot(X,self.theta)
      h = self.__sigmoid(z)
      loss = self.__loss(h,y)

      if self.verbose == True and i % 1000 == 0:
        print(f'Loss: {loss}\t')
  def predict_probability(self,X):
    if self.fit_intercept:
      X = self.__add_intercept(X)
    return self.__sigmoid(np.dot(X,self.theta))
  def predict(self,X):
    return (self.predict_probability(X).round())
```

K Means from scratch using OOPS

```
class K_Means:
    def __init__(self, k=2, tol=0.001, max_iter=300):
        self.k = k
        self.tol = tol
        self.max_iter = max_iter
```

```
    def fit(self,data):

        self.centroids = {}

        for i in range(self.k):
            self.centroids[i] = data[i]

        for i in range(self.max_iter):
            self.classifications = {}

            for i in range(self.k):
                self.classifications[i] = []

            for featureset in X:
                distances = [np.linalg.norm(featureset-self.centroids[centroid])
                classification = distances.index(min(distances))
                self.classifications[classification].append(featureset)

            prev_centroids = dict(self.centroids)

            for classification in self.classifications:
                self.centroids[classification] = np.average(self.classifications

            optimized = True

            for c in self.centroids:
                original_centroid = prev_centroids[c]
                current_centroid = self.centroids[c]
                if np.sum((current_centroid-original_centroid)/original_centroid
                    print(np.sum((current_centroid-original_centroid)/original_c
                    optimized = False

            if optimized:
                break

    def predict(self,data):
        distances = [np.linalg.norm(data-self.centroids[centroid]) for centroid
        classification = distances.index(min(distances))
        return classification

colors = 10*["g","r","c","b","k"]
```