Task 1

1. **We can treat it as a classification problem as the output values are discrete** not continuous as in the case of regression.

1. Instead of using RMSE, we can use another model: " R_squared model."
   R-squared evaluates the scatter of the data points around the fitted regression line. It is also called the coefficient of determination.

$$R^2 = \frac{\text{Variance explained by the model}}{\text{Total variance}}$$

$$\text{R-Squared} = \frac{SS_{regression}}{SS_{total}}$$

R2 calculate how much the regression Line is good than Mean Line

Interpretation of R2 score ⇒

1. If 0 → Linear and Regression Line are overlapping meaning model performance is the worst.
2. If 1 → SSr must be 0 not possible when there are no mistakes by the regression line, which is not possible.

As the R2 moves toward 1, the model performance improves.

Task 2

Implementation of gradient descent is =

```
def gradient_descent(phi_train, y_train, phi_dev, y_dev) :
   # Implement gradient_descent using Mean Squared Error Loss
   # You may choose to use the dev set to determine point of
convergence
```

```
    pre_mse = 1
    mse = pre_mse
    alpha = 0.05
    m = np.shape(phi_train)[0]  # total number of samples
    n = np.shape(phi_train)[1]  # total number of features

    phi_train = np.concatenate((np.ones((m, 1)), phi_train), axis=1)
    W = np.random.randn(n + 1, )
    mse = 10
    epsilon = 0.01

    while mse < pre_mse
        error = phi_train.dot(W) - y_train
    #     cost = (1 / 2 * m) * np.sum(error ** 2)
        W -= alpha * (1 / m) * phi_train.T.dot(error)
        pre_mse = mse
        mse = mean_squared_error(y_dev, phi_dev.dot(W))

    return W
```

Normalisation of dataset =In this normalisation, we reassigned the features by subtracting the mean and dividing by mean. Hence making them in the range of 0-1.

```
def get_test_features(file_path):
    """Read test data, perform required preproccessing /
augmentation
       and return final feature matrix.

    Args:
        file_path (str): path to test.csv

    Returns:
        phi_test: matrix of size (m,n) where m is number of test
instances
                  and n is the dimension of the feature space.
    """
    df1 = pd.read_csv(file_path)
    df1 = df1.drop(['type'], axis = 1)
```

```
    df1['fixed acidity'] = (df1['fixed acidity'] - np.min(df1['fixed
acidity'])) / (np.max(df1['fixed acidity']) - np.min(df1['fixed
acidity']))
    df1['residual sugar'] = (df1['residual sugar'] -
np.min(df1['residual sugar'])) / (np.max(df1['residual sugar']) -
np.min(df1['residual sugar']))
    df1['free sulfur dioxide'] = (df1['free sulfur dioxide'] -
np.min(df1['free sulfur dioxide'])) / (np.max(df1['free sulfur
dioxide']) - np.min(df1['free sulfur dioxide']))
    df1['total sulfur dioxide'] = (df1['total sulfur dioxide'] -
np.min(df1['total sulfur dioxide'])) / (np.max(df1['total sulfur
dioxide']) - np.min(df1['total sulfur dioxide']))
    df1.pH = (df1.pH - np.min(df1.pH)) / (np.max(df1.pH) -
np.min(df1.pH))
    df1.alcohol = (df1.alcohol - np.min(df1.alcohol)) /
(np.max(df1.alcohol) - np.min(df1.alcohol))

    phi_test = df1

    return phi_test
```

The stopping criterion = I calculated difference between the mse value at the phi_dev and y_dev set and pre_mse value and checked if it is greater than an epoch value = 10^-4 then stop the loop.

Implementation of stochastic gradient descent =

```
def sgd(phi, y, phi_dev, y_dev) :

    # Implement stochastic gradient_descent using Mean Squared Error
Loss
    # You may choose to use the dev set to determine point of
convergence

    pre_mse = 1
    mse = pre_mse
    epsilon = 0.01
    k=40
    alpha = 0.05

    while mse < pre_mse
```

```
        temp1 = phi.sample(k)
        temp2 = y.sample(k)

        phi_tr = temp1.iloc[:,0:13].values
        y_tr = temp2.iloc[:,-1].values

        m = np.shape(phi_tr)[0]  # total number of samples
        n = np.shape(phi_tr)[1]  # total number of features

        phi_tr = np.concatenate((np.ones((m, 1)), phi_tr), axis=1)
        w = np.random.randn(n + 1, )

        error = y_tr - phi_tr.dot(w)
        w -= alpha * (1 / m) * phi_tr.T.dot(error)
        alpha /= 1.02
        pre_mse = mse
        mse = mean_squared_error(y_dev, phi_dev.dot(w))

    return w
```

TASK 3 :

Implementation of p-norm :

```
  def pnorm(phi, y, phi_dev, y_dev, p) :
   # Implement gradient_descent with p-norm regularization using Mean
Squared Error Loss
   # You may choose to use the dev set to determine point of convergence
    w = np.random.random((phi.shape[1],1))
    n = phi.shape[0]
    L = []
    lam = 1e-5
    pre_rmse = compute_RMSE(phi_dev, w , y_dev) +
lam*np.sum(np.absolute(w)**p)
    alpha = 0.01
    while 1:
        grad = (phi.T)@(y - phi@w)*(-2/n) + lam*(w**(p-1))
        w -= alpha*grad/np.sqrt(np.sum(grad**2))
```

```
        rmse = compute_RMSE(phi_dev, w , y_dev) +
lam*np.sum(np.absolute(w)**p)
        conv = abs(pre_rmse - rmse)
        if conv < 1e-4:
            break
        pre_rmse = rmse
        L.append(rmse)

    return w
```

``

TASK 4 :

The code of the rmse plot

```
def plot_rmse(phi,y):

    rmse = []
    size_of_train = [0.1, 0.25, 0.5, 0.75, 0.9]

    for i in 1:5 :

        l = len(y)
        phi_train = phi[:, :int(size_of_train[i]*l)]
        y_train = y[:, :int(size_of_train[i]*l)]
        phi_dev = phi - phi_train
        y_dev = y - y_train
        w1 = gradient_descent(phi_train, y_train, phi_dev, y_dev)
        r1 = compute_RMSE(phi_dev, w1, y_dev)
        rmse[i] = r1

    plt.plot(size_of_train , rmse)
    plt.xlabel('size_of_train')
    plt.ylabel('rmse value')

    return 0
```

TASK 5 :

On viewing the correlation matrix of our dattset ,
The value of which is close to 0 is density and pH. Therefore they are the most useful data.

The values of which is close to 1 is residual sugar least useful and type since it is of string datatype.