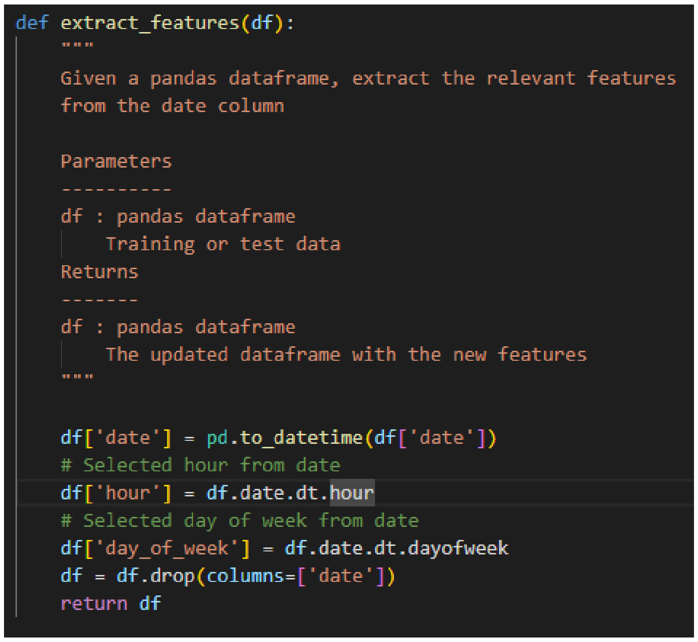
Question 1

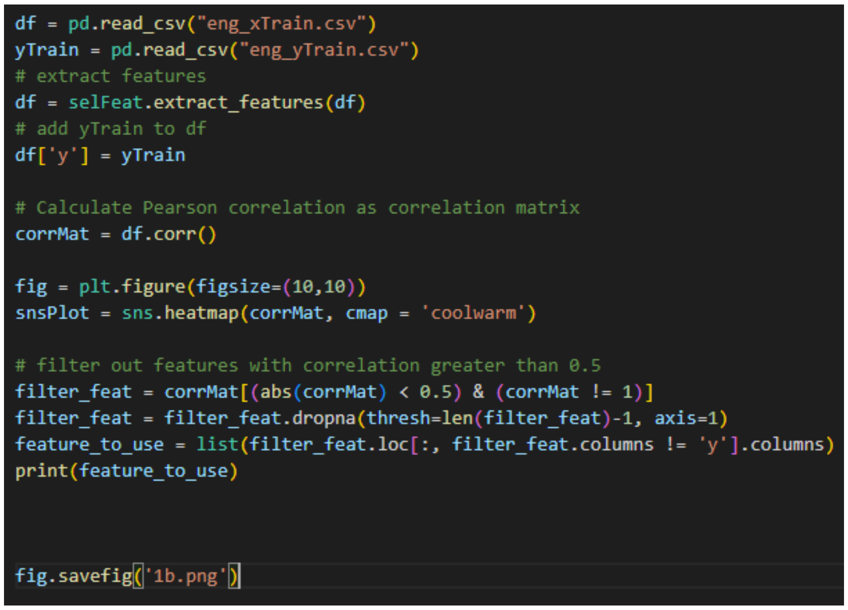
(a)

I chose ‘hour’ and ‘day of the week’.

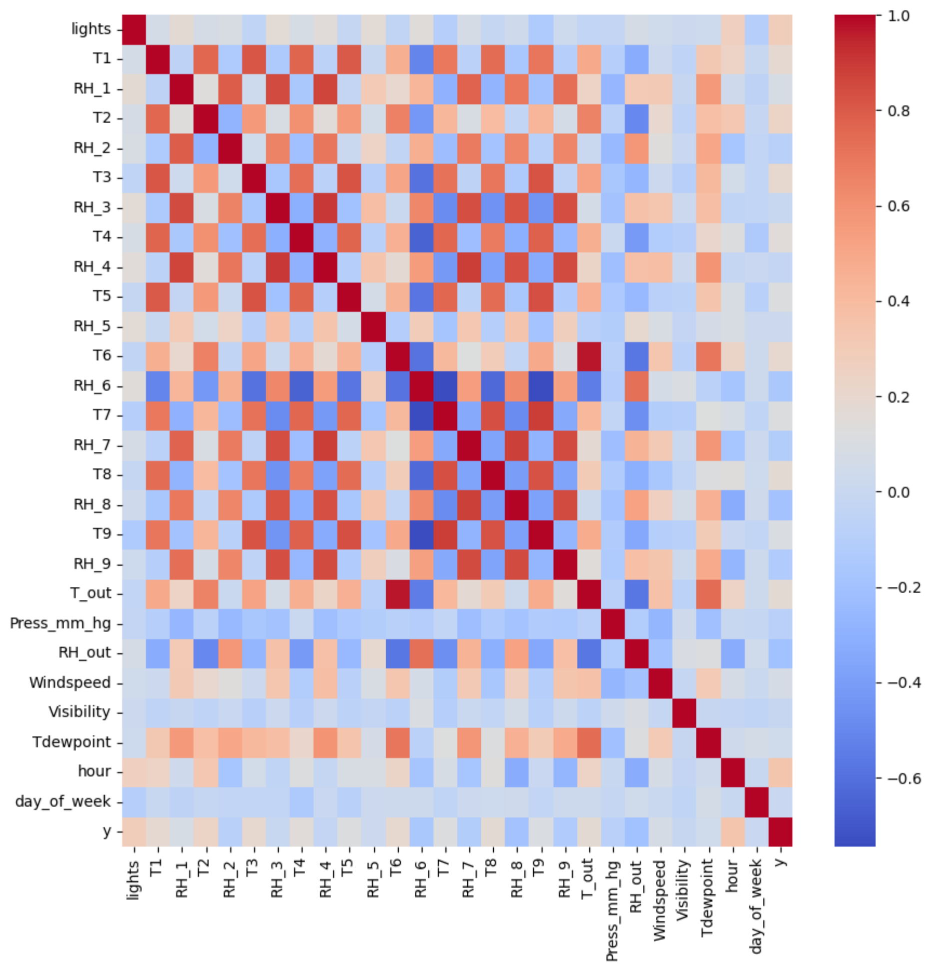


```python
def extract_features(df):
    """
    Given a pandas dataframe, extract the relevant features
    from the date column

    Parameters
    ----------
    df : pandas dataframe
        Training or test data
    Returns
    -------
    df : pandas dataframe
        The updated dataframe with the new features
    """

    df['date'] = pd.to_datetime(df['date'])
    # Selected hour from date
    df['hour'] = df.date.dt.hour
    # Selected day of week from date
    df['day_of_week'] = df.date.dt.dayofweek
    df = df.drop(columns=['date'])
    return df
```

(b)

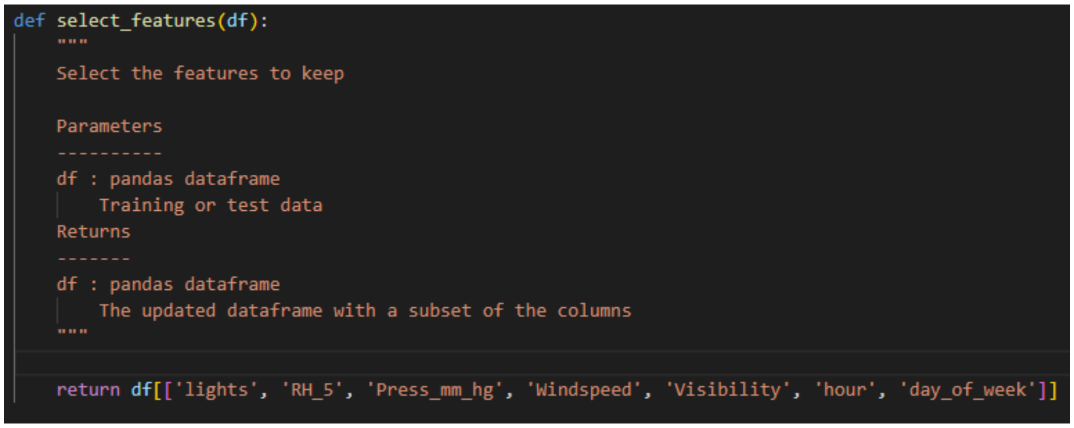


```python
df = pd.read_csv("eng_xTrain.csv")
yTrain = pd.read_csv("eng_yTrain.csv")
# extract features
df = selFeat.extract_features(df)
# add yTrain to df
df['y'] = yTrain

# Calculate Pearson correlation as correlation matrix
corrMat = df.corr()

fig = plt.figure(figsize=(10,10))
snsPlot = sns.heatmap(corrMat, cmap = 'coolwarm')

# filter out features with correlation greater than 0.5
filter_feat = corrMat[(abs(corrMat) < 0.5) & (corrMat != 1)]
filter_feat = filter_feat.dropna(thresh=len(filter_feat)-1, axis=1)
feature_to_use = list(filter_feat.loc[:, filter_feat.columns != 'y'].columns)
print(feature_to_use)


fig.savefig('1b.png')
```



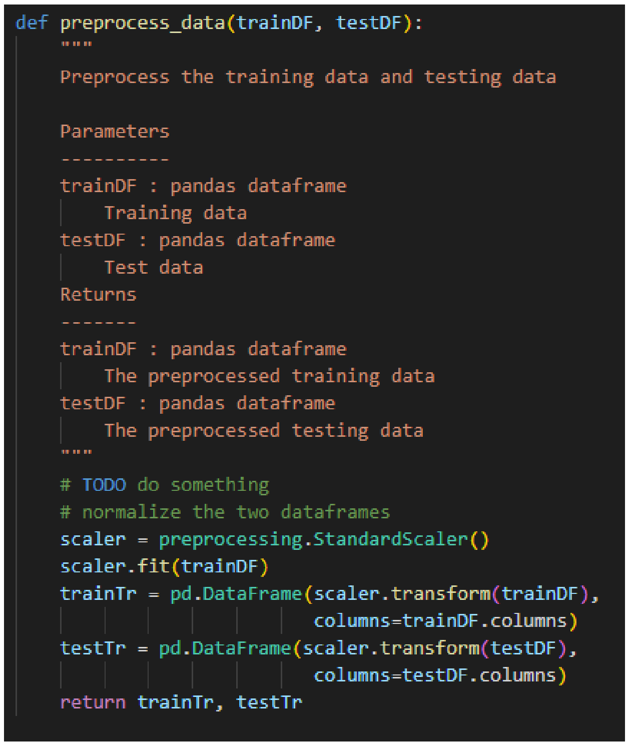
(c)

By using a threshold of 0.5, I was able to identify the features that should be used to train a linear regression model. All features with correlation greater than 0.5 were dropped and ['lights', 'RH_5', 'Press_mm_hg', 'Windspeed', 'Visibility', 'hour', 'day_of_week'] were used.



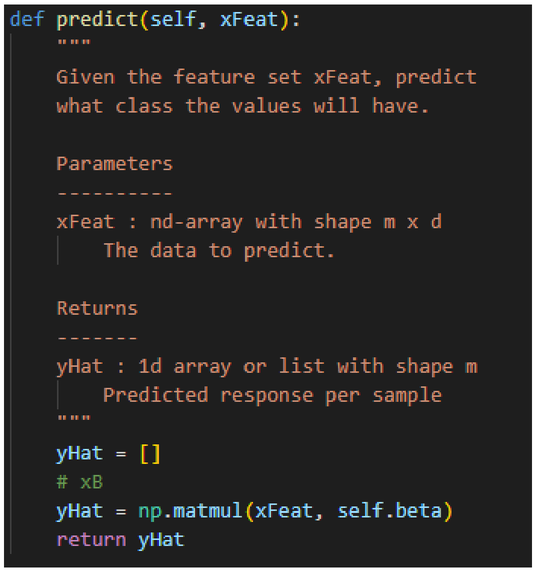
```
def select_features(df):
    """
    Select the features to keep

    Parameters
    ----------
    df : pandas dataframe
        Training or test data
    Returns
    -------
    df : pandas dataframe
        The updated dataframe with a subset of the columns
    """

    return df[['lights', 'RH_5', 'Press_mm_hg', 'Windspeed', 'Visibility', 'hour', 'day_of_week']]
```

(d)
For the preprocessing step. I used StandardScaler() to standardize the data.



```python
def preprocess_data(trainDF, testDF):
    """
    Preprocess the training data and testing data

    Parameters
    ----------
    trainDF : pandas dataframe
        Training data
    testDF : pandas dataframe
        Test data
    Returns
    -------
    trainDF : pandas dataframe
        The preprocessed training data
    testDF : pandas dataframe
        The preprocessed testing data
    """
    # TODO do something
    # normalize the two dataframes
    scaler = preprocessing.StandardScaler()
    scaler.fit(trainDF)
    trainTr = pd.DataFrame(scaler.transform(trainDF),
                           columns=trainDF.columns)
    testTr = pd.DataFrame(scaler.transform(testDF),
                          columns=testDF.columns)
    return trainTr, testTr
```

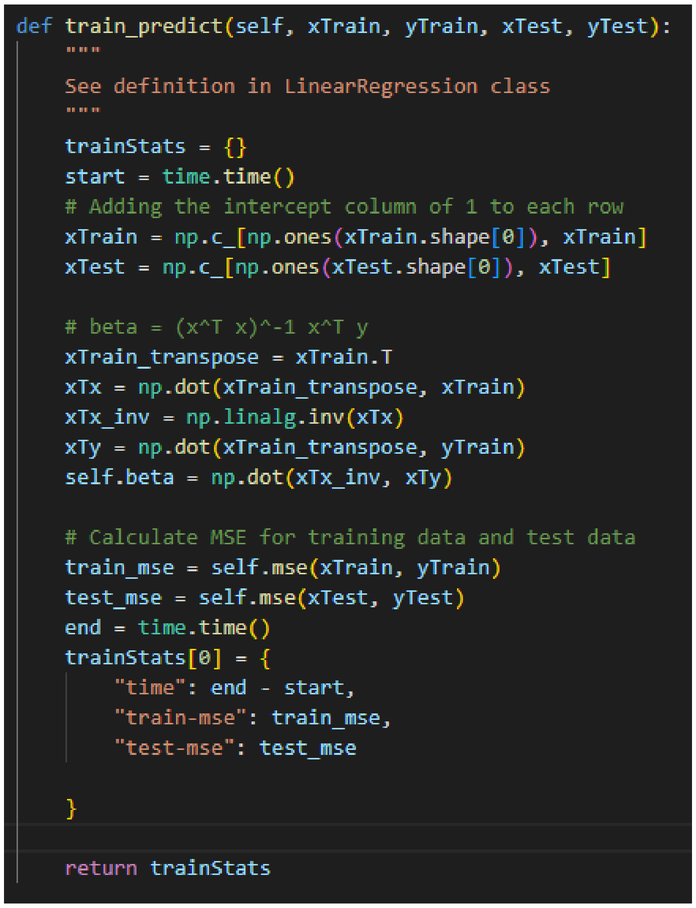
Question 2
(a)
For the predict function, in order to implement ŷ=Xbeta, I used matmul.



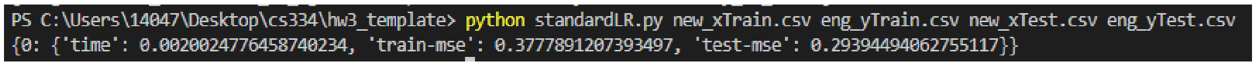
```python
def predict(self, xFeat):
    """
    Given the feature set xFeat, predict
    what class the values will have.

    Parameters
    ----------
    xFeat : nd-array with shape m x d
        The data to predict.

    Returns
    -------
    yHat : 1d array or list with shape m
        Predicted response per sample
    """
    yHat = []
    # xB
    yHat = np.matmul(xFeat, self.beta)
    return yHat
```

(b)

Closed form solution of linear regression: beta = $(X^T X)^{-1} X^T y$



```python
def train_predict(self, xTrain, yTrain, xTest, yTest):
    """
    See definition in LinearRegression class
    """
    trainStats = {}
    start = time.time()
    # Adding the intercept column of 1 to each row
    xTrain = np.c_[np.ones(xTrain.shape[0]), xTrain]
    xTest = np.c_[np.ones(xTest.shape[0]), xTest]

    # beta = (x^T x)^-1 x^T y
    xTrain_transpose = xTrain.T
    xTx = np.dot(xTrain_transpose, xTrain)
    xTx_inv = np.linalg.inv(xTx)
    xTy = np.dot(xTrain_transpose, yTrain)
    self.beta = np.dot(xTx_inv, xTy)

    # Calculate MSE for training data and test data
    train_mse = self.mse(xTrain, yTrain)
    test_mse = self.mse(xTest, yTest)
    end = time.time()
    trainStats[0] = {
        "time": end - start,
        "train-mse": train_mse,
        "test-mse": test_mse

    }

    return trainStats
```



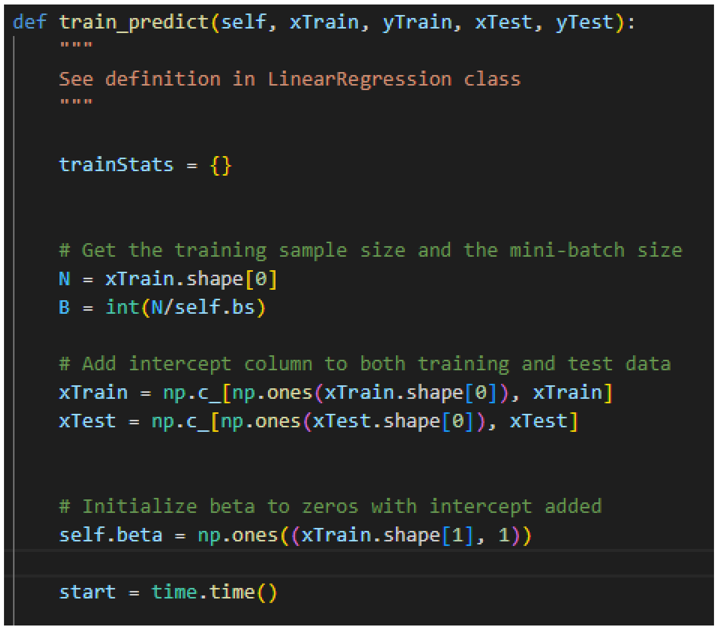
```
PS C:\Users\14047\Desktop\cs334\hw3_template> python standardLR.py new_xTrain.csv eng_yTrain.csv new_xTest.csv eng_yTest.csv
{0: {'time': 0.0020024776458740234, 'train-mse': 0.3777891207393497, 'test-mse': 0.293944940627551117}}
```

Question 3
(a)

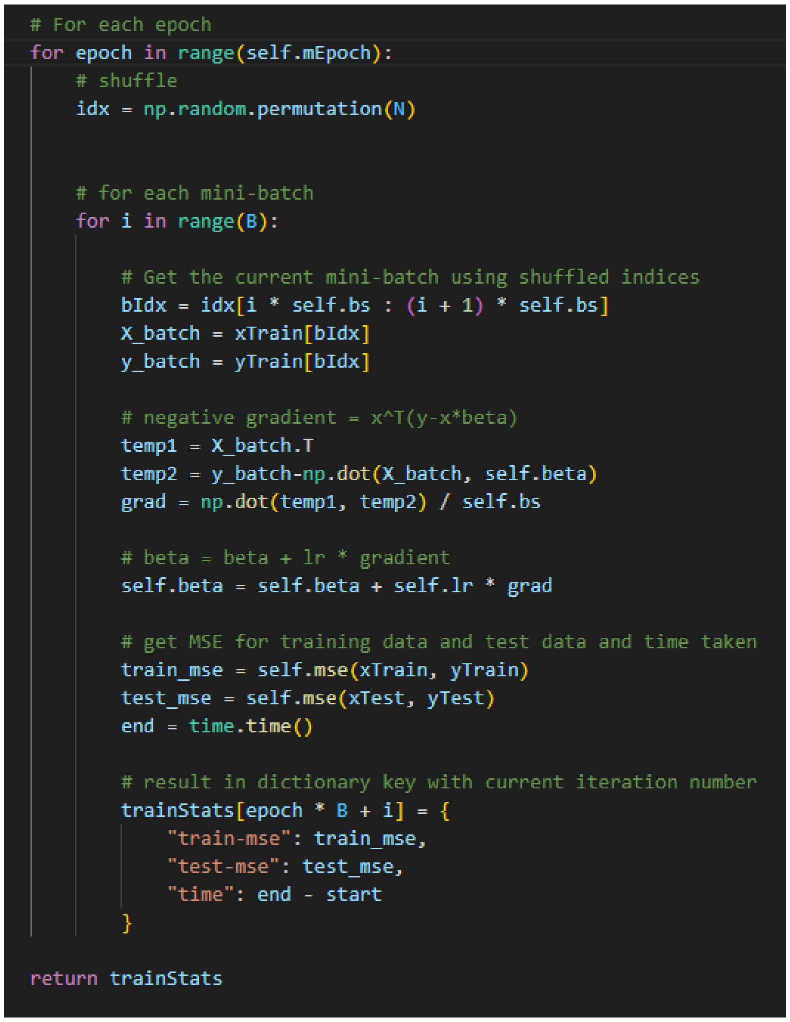
Negative gradient = X^T(y-Xbeta)

beta$^+$=beta+negative gradient

Dictionary key is the iteration number.

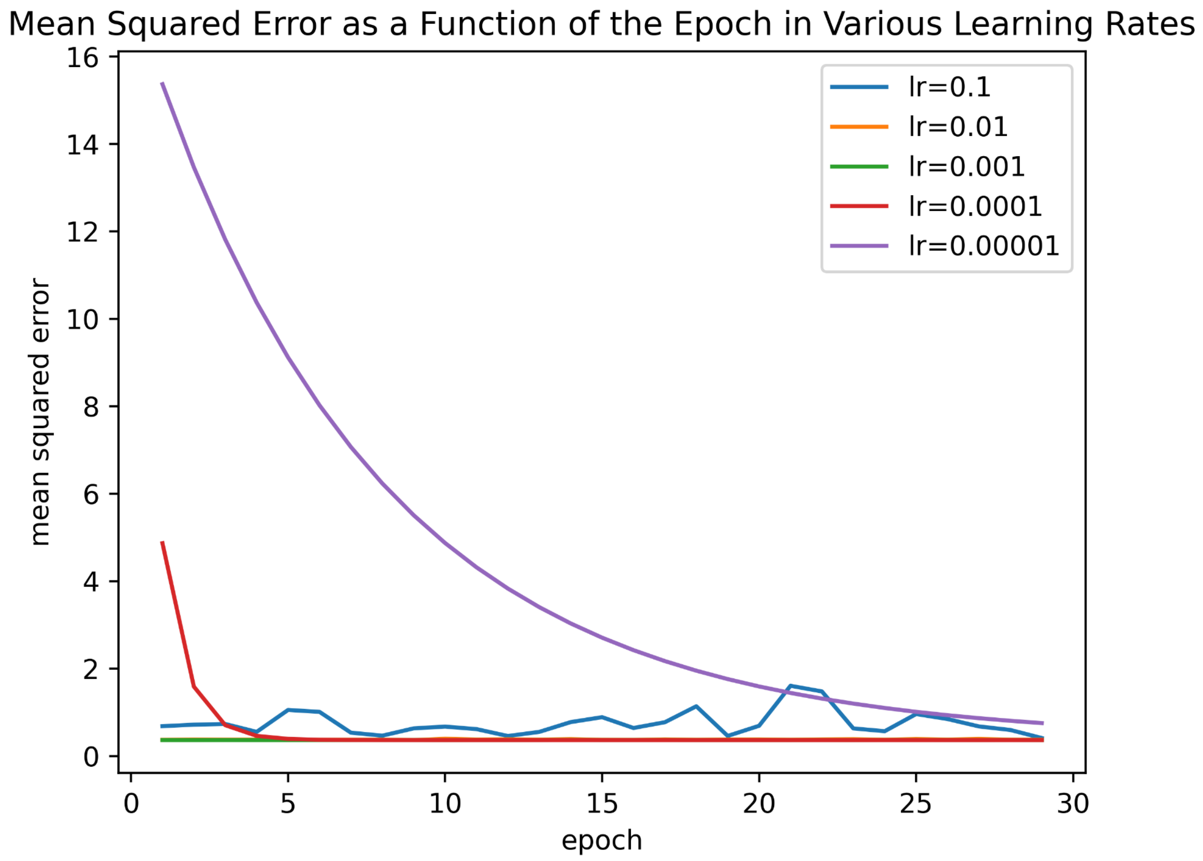


```
def train_predict(self, xTrain, yTrain, xTest, yTest):
    """
    See definition in LinearRegression class
    """

    trainStats = {}


    # Get the training sample size and the mini-batch size
    N = xTrain.shape[0]
    B = int(N/self.bs)

    # Add intercept column to both training and test data
    xTrain = np.c_[np.ones(xTrain.shape[0]), xTrain]
    xTest = np.c_[np.ones(xTest.shape[0]), xTest]


    # Initialize beta to zeros with intercept added
    self.beta = np.ones((xTrain.shape[1], 1))

    start = time.time()
```

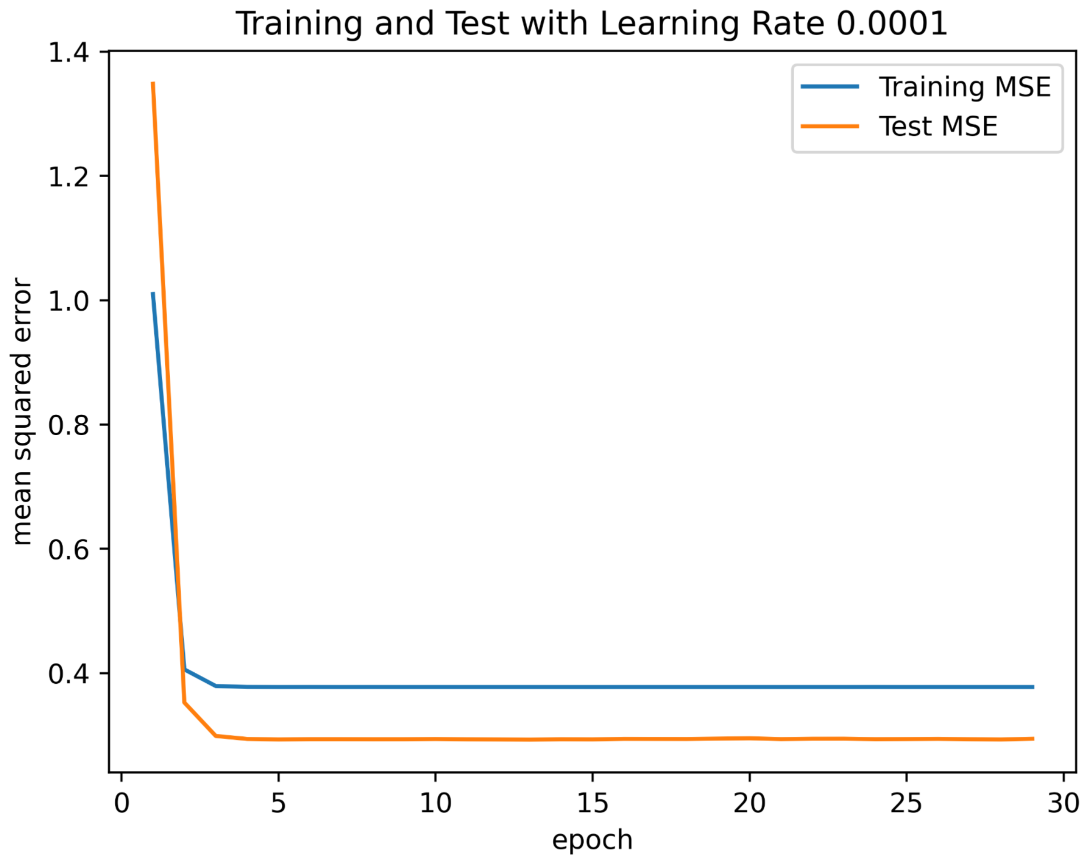


```
# For each epoch
for epoch in range(self.mEpoch):
    # shuffle
    idx = np.random.permutation(N)


    # for each mini-batch
    for i in range(B):

        # Get the current mini-batch using shuffled indices
        bIdx = idx[i * self.bs : (i + 1) * self.bs]
        X_batch = xTrain[bIdx]
        y_batch = yTrain[bIdx]

        # negative gradient = x^T(y-x*beta)
        temp1 = X_batch.T
        temp2 = y_batch-np.dot(X_batch, self.beta)
        grad = np.dot(temp1, temp2) / self.bs

        # beta = beta + lr * gradient
        self.beta = self.beta + self.lr * grad

        # get MSE for training data and test data and time taken
        train_mse = self.mse(xTrain, yTrain)
        test_mse = self.mse(xTest, yTest)
        end = time.time()

        # result in dictionary key with current iteration number
        trainStats[epoch * B + i] = {
            "train-mse": train_mse,
            "test-mse": test_mse,
            "time": end - start
        }

return trainStats
```

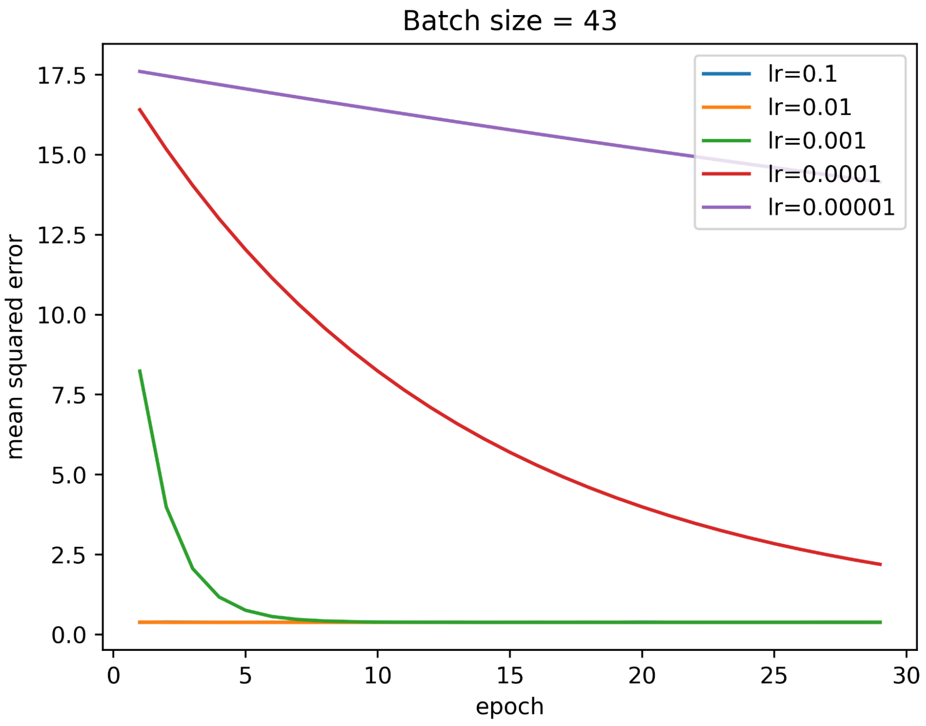
(b)
Learning rate = 0.0001 should be optimal. Looking into learning rate lower than 0.0001 showed too long to decrease the mean squared error and learning rate higher than 0.0001 showed divergence and the mean squared error was unsteady.

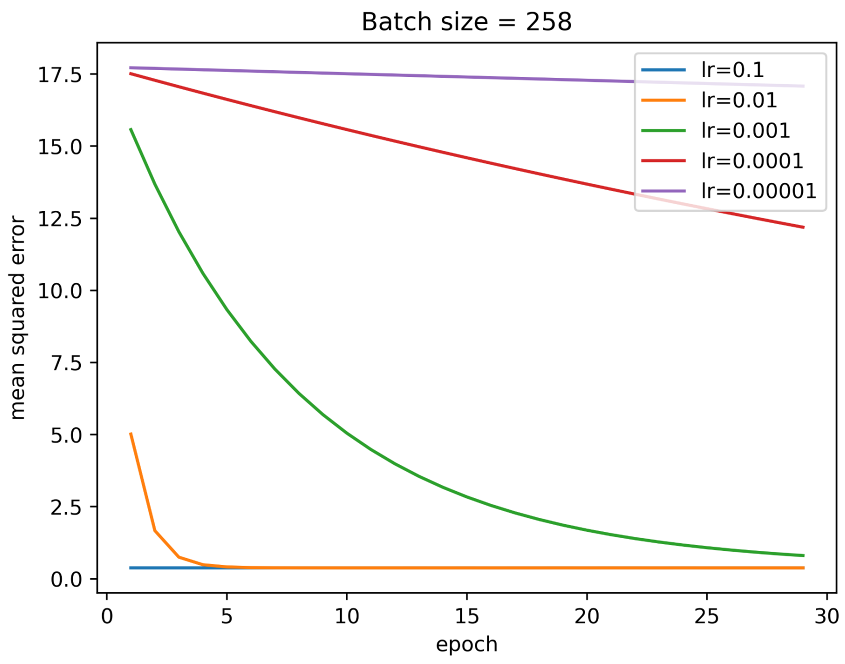


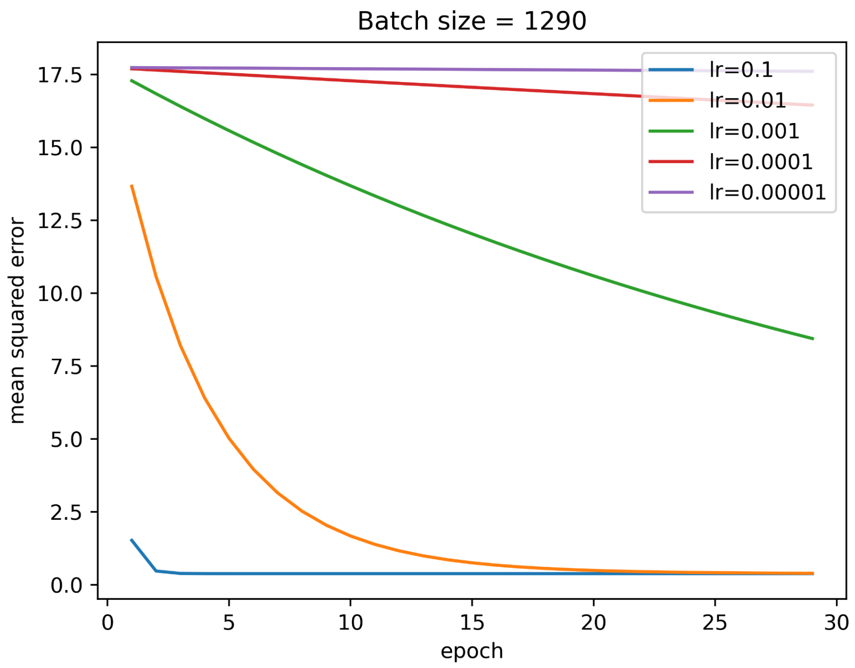
(c)

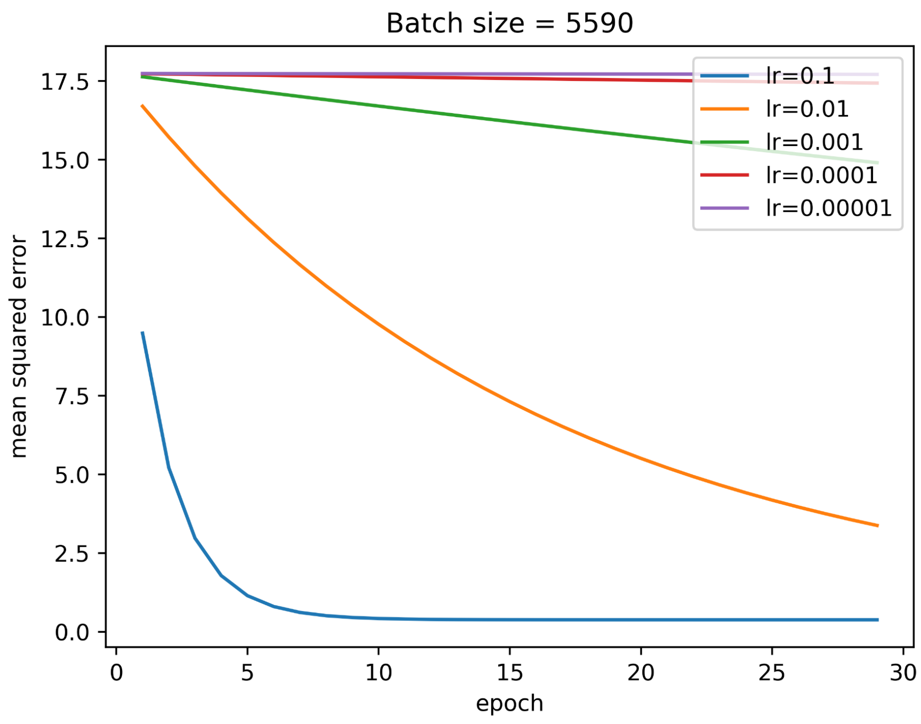

Training and Test with Learning Rate 0.0001
1.4
1.2
1.0
0.8
0.6
0.4
mean squared error
0
5
10
15
20
25
30
epoch
Training MSE
Test MSE

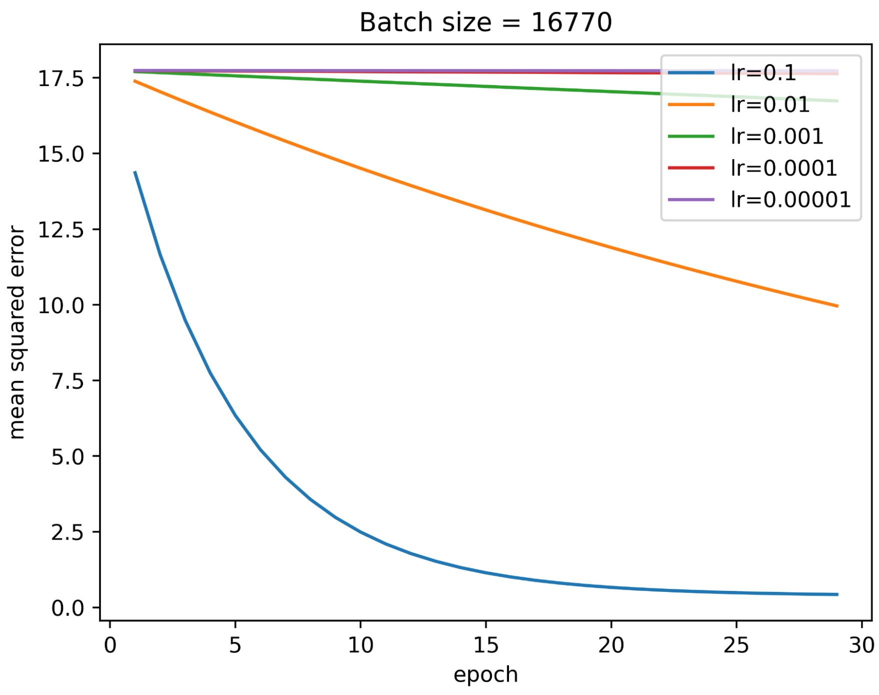
Question 4
(a)
Batch size = 43
Learning rate = 0.001 should be optimal. Looking into learning rate lower than 0.001 showed too long to decrease the mean squared error and learning rate higher than 0.001 showed divergence and the mean squared error was unsteady.
Batch size = 258
Learning rate = 0.01 should be optimal. Looking into learning rate lower than 0.01 showed too long to decrease the mean squared error and learning rate higher than 0.01 showed divergence and the mean squared error was unsteady.
Batch size = 1290, 5590, 16770
Learning rate = 0.1 should be optimal. Looking into learning rate lower than 0.1 showed too long to decrease the mean squared error and learning rate higher than 0.1 showed divergence and the mean squared error was unsteady.

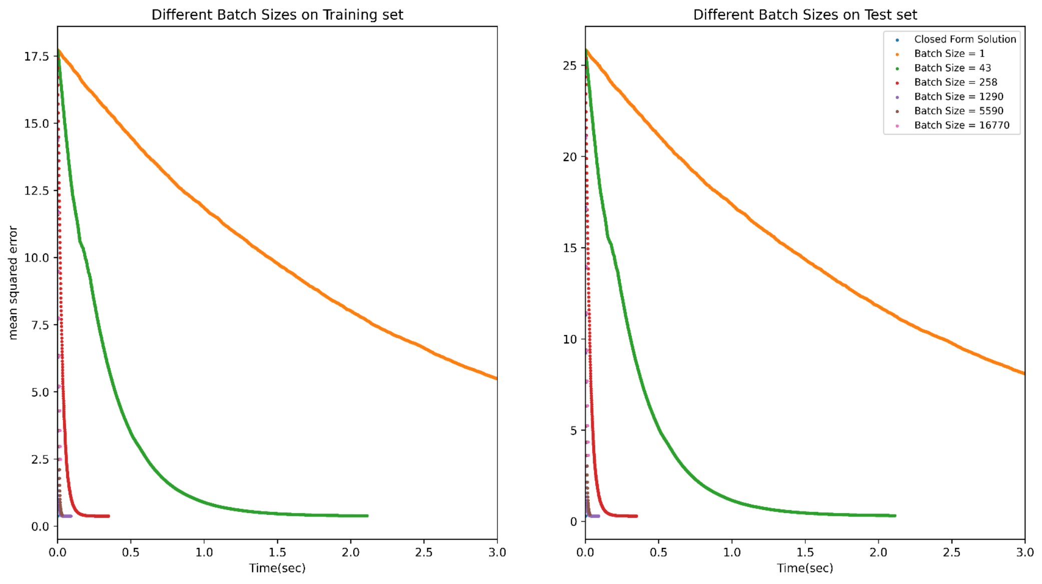



Batch size = 258
mean squared error
epoch
lr=0.1
lr=0.01
lr=0.001
lr=0.0001
lr=0.00001
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
25
30


Batch size = 1290
mean squared error
epoch
lr=0.1
lr=0.01
lr=0.001
lr=0.0001
lr=0.00001
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
25
30


Batch size = 5590
mean squared error
epoch
lr=0.1
lr=0.01
lr=0.001
lr=0.0001
lr=0.00001
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
25
30


Batch size = 16770
mean squared error
epoch
lr=0.1
lr=0.01
lr=0.001
lr=0.0001
lr=0.00001
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
25
30



(d)

Small batch sizes had a longer computation time and large batch sizes had a shorter computation time. Also, the closed form solution had the shortest computation time. However, the smaller batch size had better results as the function approached a lower mean squared error.