1.1) Without Data NOrmalization / Standardization

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remoun
```

```
import pandas as pd
```

```
data = pd.read_csv('/content/drive/My Drive/ISL/advertising.csv')
```

```
data.head()
```

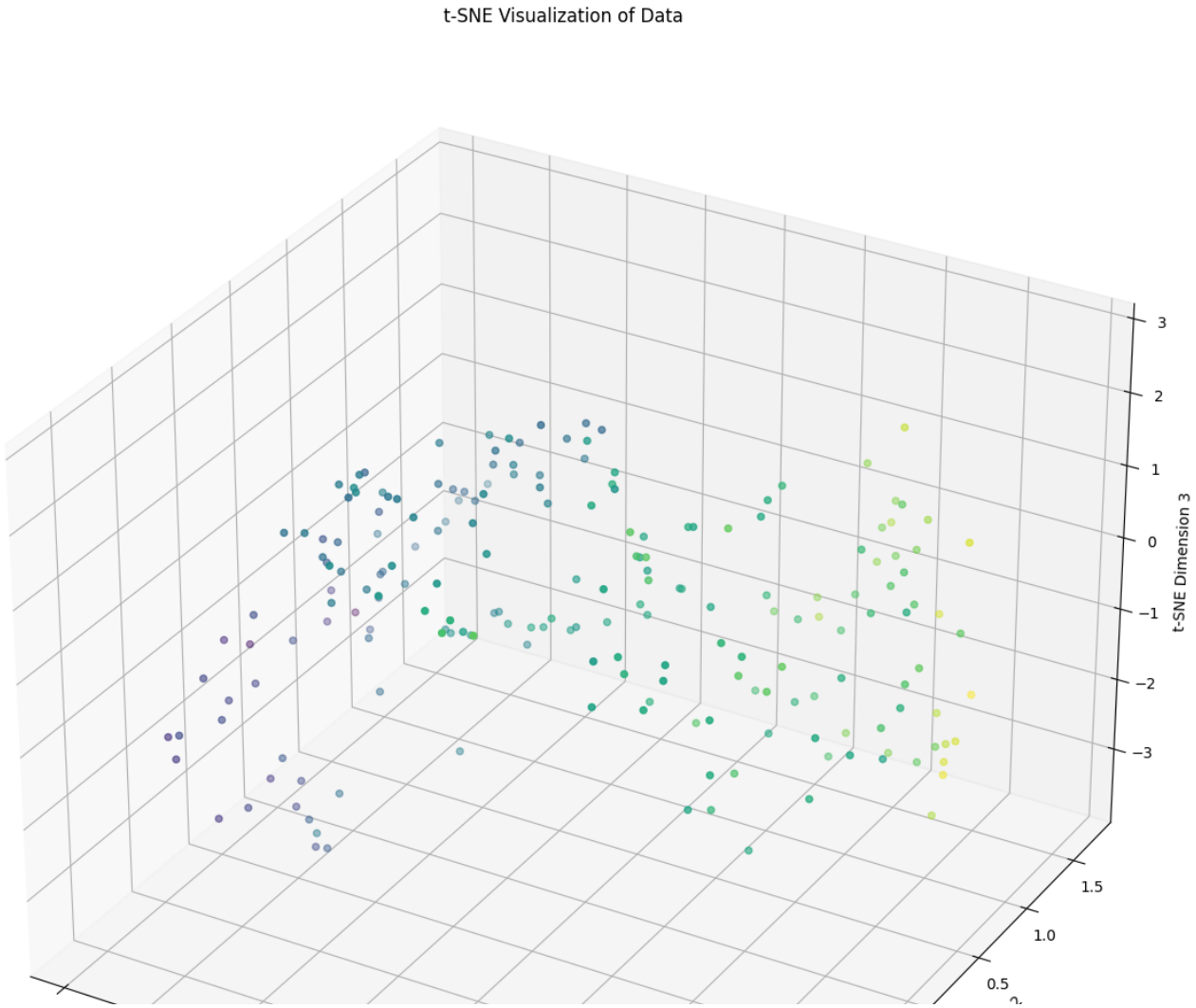
	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9

```
features = data[['TV', 'Radio', 'Newspaper']].values
```

3-D tsn3 visualization

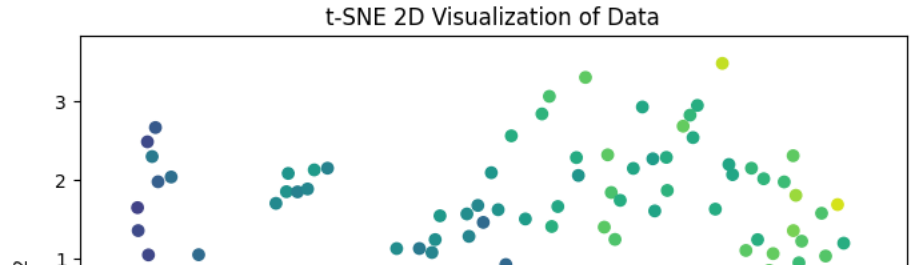
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=3, perplexity=30, n_iter=3000)
reduced_features = tsne.fit_transform(features)
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(15,20))
ax = fig.add_subplot(111, projection='3d')

sales = data['Sales'].values

ax.scatter(reduced_features[:, 0], reduced_features[:, 1], reduced_features[:, 2], c=sales, cmap='viridis')
ax.set_xlabel('t-SNE Dimension 1')
ax.set_ylabel('t-SNE Dimension 2')
ax.set_zlabel('t-SNE Dimension 3')
ax.set_title('t-SNE Visualization of Data')

plt.show()
```



2-D tsne visualtion

```
tsne = TSNE(n_components=2, perplexity=30, n_iter=300)  # for 2D visualization

reduced_features = tsne.fit_transform(features)

plt.figure(figsize=(8, 6))


sales = data['Sales'].values

plt.scatter(reduced_features[:, 0], reduced_features[:, 1], c=sales, cmap='viridis')
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.title('t-SNE 2D Visualization of Data')

plt.show()
```



Modelling

```
#  Splitting the data into training and test data set , for this assignment taking the first 80 % samples as training and las

total = len(features)

train_no = int( 0.8 * total )
test_no = total - train_no

X = features[0:train_no]
y = data['Sales'].values[0:train_no]

test_features = features[train_no : 201 ]
test_labels = data['Sales'].values[train_no : 201]
```

```
X.shape
```

```
(160, 3)
```

```
import numpy as np
# Padding of 1 to matrix X
ones = np.ones((X.shape[0], 1))
X_padded = np.hstack((X , ones))
```

Linear Model without Regularization

$$\hat{\beta}_{ls} = (X'X)^{-1}X'Y$$

```
import numpy as np
```

```
#  As we know the parameter for linear regression if supposed as B then B = (X.T X) inv Xt y
X_padded = np.array(X_padded)
y = np.array(y)

B_linear = np.linalg.inv( X_padded.T @ X_padded ) @ ( X_padded.T @ y )
B_linear
```

```
array([5.47207688e-02, 1.04776019e-01, 3.06638052e-03, 4.48194957e+00])
```

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)

# Get the coefficients (parameters)
intercept = model.intercept_
coefficients = model.coef_

print("Intercept:", intercept)
print("Coefficients:", coefficients)
```

```
Intercept: 4.481949566663138
Coefficients: [0.05472077 0.10477602 0.00306638]
```

As we can see the coefficients values are exaclty same as the one calucated by inbuilt libraires

```
X_padded.shape
```

```
(160, 4)
```

linear Model with regularizartion (Ridge Regression)

$$\hat{\beta} = \hat{\beta}_{ridge} = (X'X + \lambda I_p)^{-1} X'Y$$

```
# As we know for Ridge regression the equarion looks like the above
lemda = 0.01
B_ridge = np.linalg.inv( X_padded.T @ X_padded + lemda * np.identity(4) ) @ ( X_padded.T @ y )

B_ridge
```

```
array([5.47259510e-02, 1.04799327e-01, 3.07865631e-03, 4.47996558e+00])
```

```
from sklearn.linear_model import Ridge
alpha = 0.01  # You can adjust the alpha value
ridge_model = Ridge(alpha=alpha)
ridge_model.fit(X, y)

# Get the coefficients (parameters)
coefficients = ridge_model.coef_
intercept = ridge_model.intercept_

print("Intercept:", intercept)
print("Coefficients:", coefficients)
```

```
Intercept: 4.4819501842851786
Coefficients: [0.05472077 0.10477598 0.00306639]
```

As we can see the parameters are same for

> The one calculated statistically

> calculated with inbuilt libraires

Computing errors

```
print(test_features.shape)
test_labels.shape
```

```
(40, 3)
(40,)
```

```
test_features = np.array(test_features)
test_labels = np.array(test_labels)

# padding test features with 1

ones = np.ones((test_features.shape[0] , 1))
test_features_padded = np.hstack((test_features , ones))
```

```
linear_pred_train = X_padded @ B_linear
ridge_pred_train = X_padded @ B_ridge

linear_pred_test = test_features_padded @ B_linear
ridge_pred_test = test_features_padded @ B_ridge


linear_train_error_1 = ((y -  linear_pred_train).T @ ( y -  linear_pred_train )) / ( 2 * ( X_padded.shape[0]))
ridge_train_error_1 =   ((y -  ridge_pred_train).T @ ( y -  ridge_pred_train )) / ( 2 * ( X_padded.shape[0]))

linear_test_error_1 = ((test_labels -  linear_pred_test).T @ ( test_labels -  linear_pred_test )) / ( 2 * ( test_features_pad
ridge_test_error_1 = ((test_labels -  ridge_pred_test).T @ ( test_labels -  ridge_pred_test )) /  (2 * ( test_features_padded



print("Linear Regression Training Error:", linear_train_error_1)
```

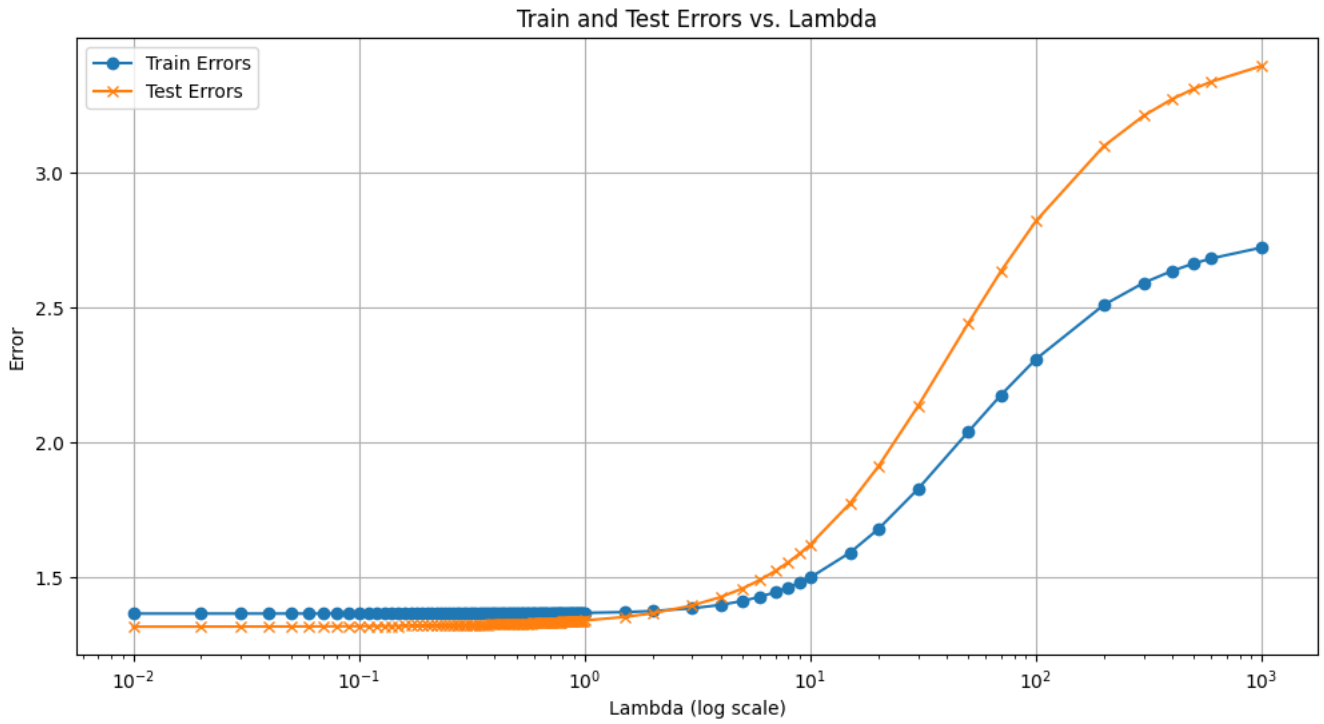
```
print("Ridge Regression Training Error:", ridge_train_error_1)
print("Linear Regression Testing Error:", linear_test_error_1)
print("Ridge Regression Testing Error:", ridge_test_error_1)
```

```
Linear Regression Training Error: 1.3671985804792106
Ridge Regression Training Error: 1.3671988581486008
Linear Regression Testing Error: 1.3184703580408181
Ridge Regression Testing Error: 1.3186815950080715
```

Experimenting with different values of lemda

```
lemdas = [ 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19,
test_errors = []
train_errors = []
for lemda in lemdas:
  B_ridge = np.linalg.inv( X_padded.T @ X_padded + lemda * np.identity(4) ) @ ( X_padded.T @ y )

  pred = test_features_padded @ B_ridge
  test_error = ((( test_labels - pred).T)@ (test_labels - pred) ) / ( 2 * test_features_padded.shape[0])
  test_errors.append(test_error)

  pred = X_padded @  B_ridge
  train_error = ((( y - pred).T) @ ( y - pred) ) / ( 2 * X_padded.shape[0])
  train_errors.append(train_error)
```

```
# Create the plot
plt.figure(figsize=(12, 6))
plt.plot((lemdas), train_errors, label='Train Errors', marker='o')
plt.plot((lemdas), test_errors, label='Test Errors', marker='x')
plt.xscale('log')  # To make the x-axis logarithmic
plt.xlabel('Lambda (log scale)')
plt.ylabel('Error')
plt.title('Train and Test Errors vs. Lambda')
plt.legend()
plt.grid(True)
plt.show()
```



#

1.2) With Normalization and standardizaion

```
min_vals = np.min(X, axis=0)
max_vals = np.max(X, axis=0)

min_vals_y = np.min(y , axis = 0 )
max_vals_y= np.max(y , axis=0)
```

```
X = (X - min_vals) / (max_vals - min_vals)
y =  (y - min_vals_y) / (max_vals_y - min_vals_y)
```

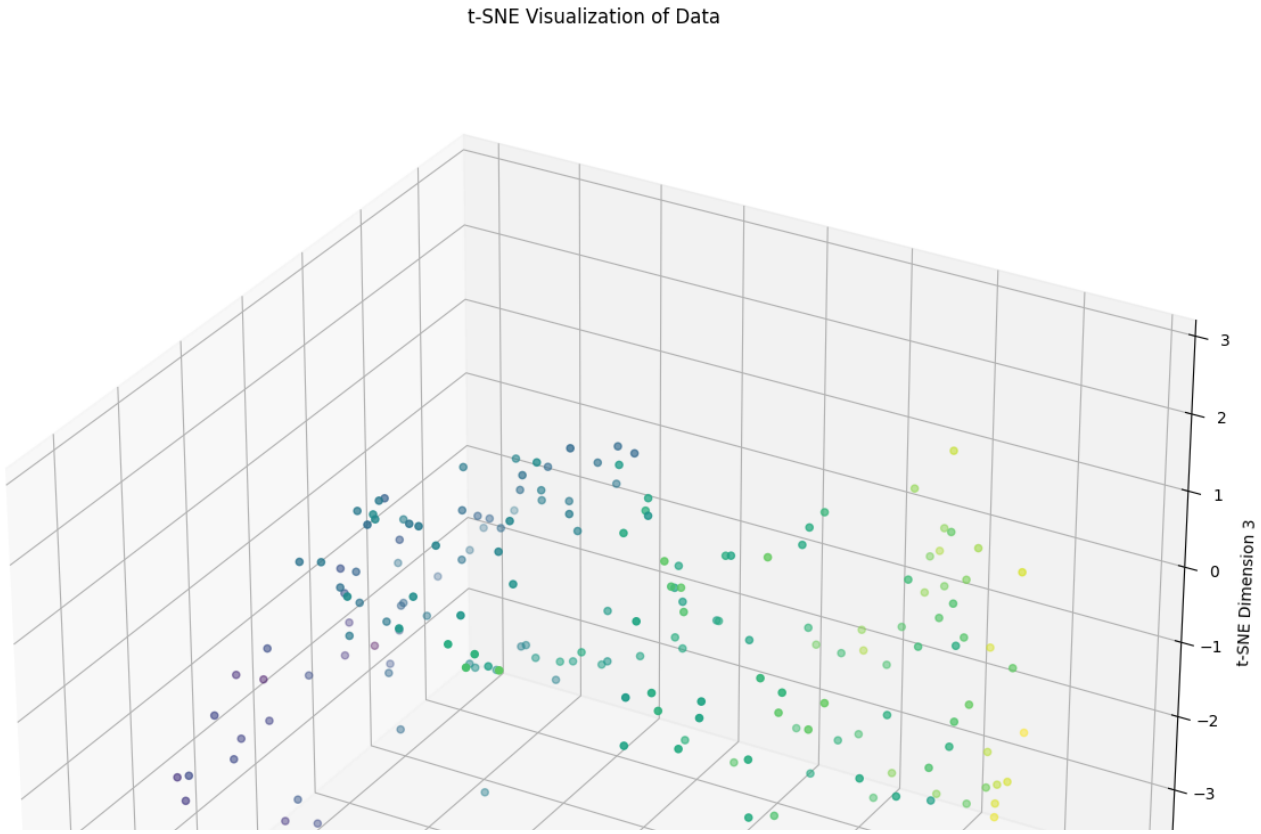
```
mean_vals = np.mean(X, axis=0)
std_devs = np.std(X, axis=0)


mean_vals_y = np.mean(y, axis=0)
std_devs_y = np.std(y, axis=0)
```

```
X = (X - mean_vals) / std_devs
y = ( y - mean_vals_y) /  std_devs_y
```

3-D tsne

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=3, perplexity=30, n_iter=3000)

reduced_features = tsne.fit_transform(features)
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(15,20))
ax = fig.add_subplot(111, projection='3d')

sales = data['Sales'].values

ax.scatter(reduced_features[:, 0], reduced_features[:, 1], reduced_features[:, 2], c=sales, cmap='viridis')
ax.set_xlabel('t-SNE Dimension 1')
ax.set_ylabel('t-SNE Dimension 2')
ax.set_zlabel('t-SNE Dimension 3')
ax.set_title('t-SNE Visualization of Data')

plt.show()
```



PAdding of X

```
# Padding of 1 to matrix X
ones = np.ones((X.shape[0], 1))
X_padded = np.hstack((X , ones))
```

Linear

```
X_padded = np.array(X_padded)
y = np.array(y)

B_linear = np.linalg.inv( X_padded.T @ X_padded ) @ ( X_padded.T @ y )
B_linear
```

```
array([8.86190469e-01, 2.90834750e-01, 1.27825629e-02, 3.88865703e-16])
```

Ridge

```
lemda = 0.01
B_ridge = np.linalg.inv( X_padded.T @ X_padded + (lemda * np.identity(4)) ) @ ( X_padded.T @ y )

B_ridge
```

```
array([8.86135872e-01, 2.90816822e-01, 1.27899138e-02, 3.88832274e-16])
```

Comparison of erros

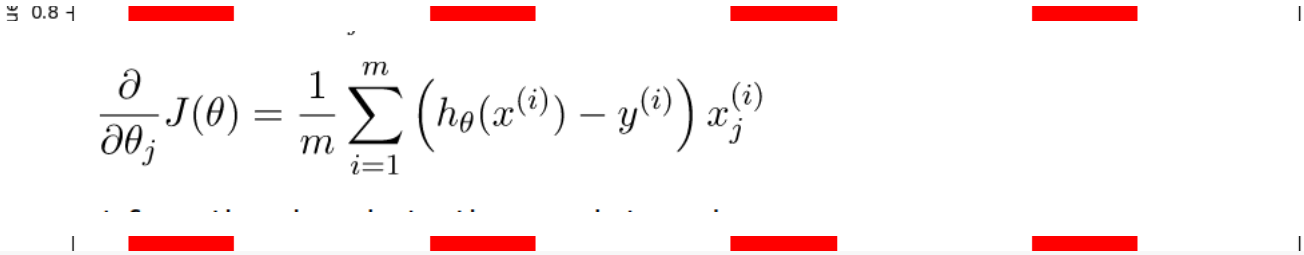
```
test_features = (test_features - min_vals) / (max_vals - min_vals)
test_features = (test_features - mean_vals) / std_devs

test_labels = (test_labels - min_vals_y) / (max_vals_y - min_vals_y)
test_labels = (test_labels - mean_vals_y) / std_devs_y
# padding test features with 1

ones = np.ones((test_features.shape[0] , 1))
test_features_padded = np.hstack((test_features , ones))
```

```
linear_pred_train = X_padded @ B_linear
ridge_pred_train = X_padded @ B_ridge

linear_pred_test = test_features_padded @ B_linear
ridge_pred_test = test_features_padded @ B_ridge


linear_train_error_2 = ((y -  linear_pred_train).T @ ( y -  linear_pred_train )) / ( 2 * ( X_padded.shape[0]))
ridge_train_error_2 =   ((y -  ridge_pred_train).T @ ( y -  ridge_pred_train )) / ( 2 * ( X_padded.shape[0]))

linear_test_error_2 = ((test_labels -  linear_pred_test).T @ ( test_labels -  linear_pred_test )) / ( 2 * ( test_features_pad
ridge_test_error_2 = ((test_labels -  ridge_pred_test).T @ ( test_labels -  ridge_pred_test )) /  (2 * ( test_features_padded



print("Linear Regression Training Error:", linear_train_error_2)
print("Ridge Regression Training Error:", ridge_train_error_2)
print("Linear Regression Testing Error:", linear_test_error_2)
print("Ridge Regression Testing Error:", ridge_test_error_2)
```

```
Linear Regression Training Error: 0.049144184358431243
Ridge Regression Training Error: 0.04914418603031624
Linear Regression Testing Error: 0.04739264015617485
Ridge Regression Testing Error: 0.04739197713322119
```

```
import matplotlib.pyplot as plt
import numpy as np

error_types = ['Linear Regression Training', 'Ridge Regression Training', 'Linear Regression Testing', 'Ridge Regression Test
num_error_types = len(error_types)

cases = ['Case 1', 'Case 2']
error_values = np.array([
    [linear_train_error_1, ridge_train_error_1, linear_test_error_1, ridge_test_error_1],
    [linear_train_error_2, ridge_train_error_2, linear_test_error_2, ridge_test_error_2]
])

# Define colors for the bars for each case
colors = ['r', 'y']  # Blue for Case 1, Green for Case 2

bar_width = 0.35

x = np.arange(num_error_types)

fig, ax = plt.subplots(figsize=(10, 6))

for i, case in enumerate(cases):
    ax.bar(x + i * bar_width, error_values[i], bar_width, label=case, color=colors[i])

ax.set_xlabel('Error Types')
ax.set_ylabel('Error Value')
ax.set_title('Comparison of Errors for case 1 (didnt normalise/stnd) and Case 2 ( noramlize / stnd)')
ax.set_xticks(x + bar_width / 2)
ax.set_xticklabels(error_types)
ax.legend()

plt.tight_layout()
plt.show()
```

2) Logistic Regression



$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

```
data = pd.read_csv("/content/drive/My Drive/ISL/banknote_authentication.csv")
```

```
data.head(5)
```

	variance	skewness	curtosis	entropy	class
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

```
data = data.values
```

```
split_ratio = 0.8


total_samples = len(data)
train_samples = int(total_samples * split_ratio)
test_samples = total_samples - train_samples


train_data = data[:train_samples]
test_data = data[train_samples:]
```

```
X = train_data[ : , : -1]
y = train_data[ : , -1]

test_X = test_data[ : , : -1]
test_y = test_data[ : , -1]
```

```
# padding the train data
ones = np.ones((X.shape[0] , 1))
X_padded = np.hstack((X , ones))
X_padded
```

```
array([[ 3.6216 ,  8.6661 , -2.8073 , -0.44699,  1.     ],
       [ 4.5459 ,  8.1674 , -2.4586 , -1.4621 ,  1.     ],
       [ 3.866  , -2.6383 ,  1.9242 ,  0.10645,  1.     ],
       ...,
       [ 2.0177 ,  1.7982 , -2.9581 ,  0.2099 ,  1.     ],
       [ 1.164  ,  3.913  , -4.5544 , -3.8672 ,  1.     ],
       [-4.3667 ,  6.0692 ,  0.57208, -5.4668 ,  1.     ]])
```

```
def sigm( x , theta):
  z = np.dot( x , theta)
  ep = np.exp(-z)
  return ( (1) / ( 1 + ep))
```

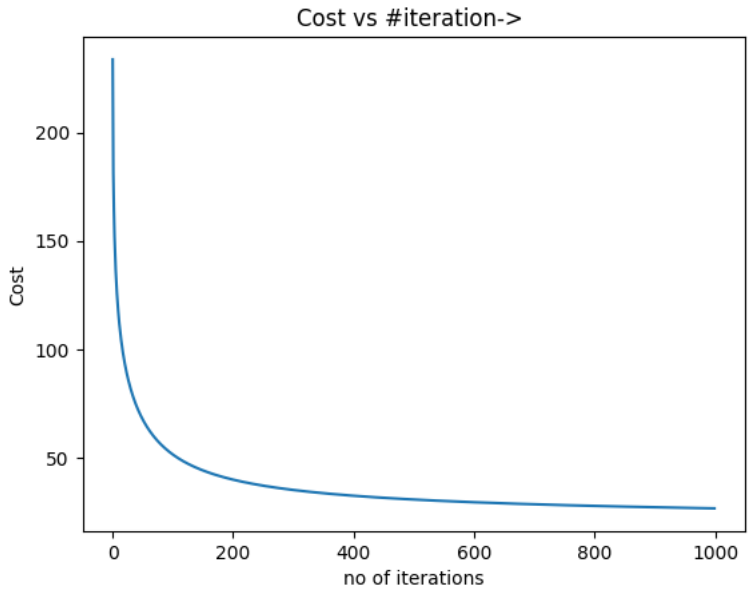
```
  return ( (1) / ( 1 + ep))
```

```
# padding test-X
ones  = np.ones((test_X.shape[0] , 1))
test_X_padded = np.hstack((test_X ,ones))
```

```
theta = [0 , 0 ,0 ,0 ,0 ]
```

```
# gradient descent for the optimal parameters
alpha = 0.5
costs = []

iters = 1000


for i in range(iters):
  ptheta = theta
  for j in range(len(theta)):
    temp = 0;
    for xx, yy in zip(X_padded , y ) :
      fx = sigm(xx, ptheta)
      temp = temp + (( fx - yy ) * (xx[j]))
    temp /= (X_padded.shape[0])
    temp *= alpha
    theta[j] -= temp


  st = X_padded @ theta
  prob = ( (1) /( 1 + np.exp(-st)) )


  cost = 0
  for predd , yy in zip( prob , y ):
    if yy == 1:
      cost += -(np.log(predd))
    else:
      cost += -(np.log(1- predd))
  costs.append(cost)
```

```
plt.plot(costs)
plt.title("Cost vs #iteration-> ")
plt.xlabel("no of iterations")
plt.ylabel("Cost")
plt.show()
```




```
theta
```

```
[-3.426170739063408,
 -1.8817312945138627,
```

```
       -2.3006951893166594,
       -0.09374573346865879,
       3.214515797551809]
```

```
st = test_X_padded @ theta

prob = ( (1) /( 1 + np.exp(-st)) )
pred = ( prob >= 0.5).astype(int)


test_cost = 0 ;
for predd , yy in zip( prob , test_y ):
  if yy == 1:
    test_cost += -(np.log(predd))
  else:
    test_cost += -(np.log(1- predd))




print(test_cost)
```

```
    7.765000364905097
```

```
# defining accuracy as  correctly identified / total test

num = ( pred == test_y)
# num = np.array(num)

accuracy = np.count_nonzero(num) / (test_y.shape[0])
print(f"{accuracy} is the test accuracy")



st = X_padded @ theta

prob = ( (1) /( 1 + np.exp(-st)) )
pred_train = ( prob >= 0.5).astype(int)

num = ( pred_train == y)

accuracy = np.count_nonzero(num) / (y.shape[0])
print(f"{accuracy} is the train accuracy")



train_cost = 0 ;
for predd , yy in zip( prob , y ):
  if yy == 1:
    train_cost += -(np.log(predd))
  else:
    train_cost += -(np.log(1- predd))

print()
print(f"{train_cost} is the training cost" )
print(f"{test_cost} is the testing cost" )
```

```
    0.9927272727272727 is the test accuracy
    0.9863263445761167 is the train accuracy

    26.7434716825555 is the training cost
    7.765000364905097 is the testing cost
```

As we can see the accuracy that we are getting is > 98 %

also the train error and train error are comparable hence it is a good and generalized model

3) Understanding Bias Variance Trade off

1. Load the data and plot the stock value Vs time.

```
data = pd.read_excel("/content/drive/My Drive/ISL/Apple_stock_data.xlsx")
data.head(5)
```

	Date	Close/Last	Volume	Open	High	Low
0	09/29/2023	$171.21	51861080	$172.02	$173.07	$170.341
1	09/28/2023	$170.69	56294420	$169.34	$172.03	$167.62
2	09/27/2023	$170.43	66921810	$172.62	$173.04	$169.05
3	09/26/2023	$171.96	64588950	$174.82	$175.20	$171.66
4	09/25/2023	$176.08	46172740	$174.20	$176.97	$174.15

```
df = data
df['Close/Last'].dtype
```

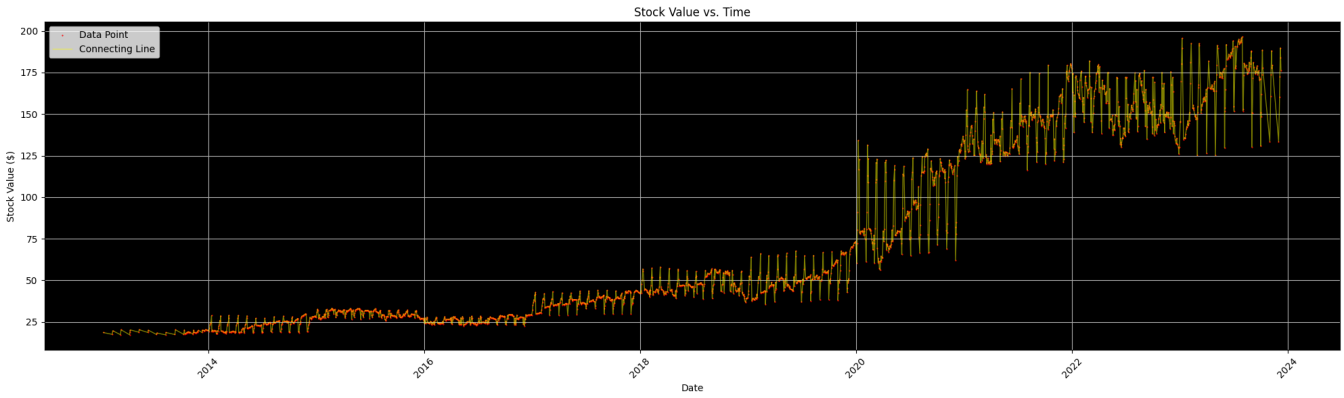
```
dtype('O')
```

```
df['Date'] = pd.to_datetime(df['Date'])

df['Close/Last'] = df['Close/Last'].str.replace('$', '').astype(float)
```

```
<ipython-input-296-b53857874181>:3: FutureWarning: The default value of regex will change from True to False in a future
  df['Close/Last'] = df['Close/Last'].str.replace('$', '').astype(float)
```

```
df.sort_values(by='Date', inplace=True)
```

```
plt.figure(figsize=(20, 6))
plt.gca().set_facecolor('black')
plt.scatter(df['Date'], df['Close/Last'], marker='o', label='Data Point', color='red' , s = 0.3)

plt.plot(df['Date'], df['Close/Last'], linestyle='-', linewidth=1, color='yellow', alpha=0.5, label='Connecting Line')

plt.title('Stock Value vs. Time')
plt.xlabel('Date')
plt.ylabel('Stock Value ($)')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.legend()

plt.show()
```



2. Split the data into train and test. (data upto 2022, consider as training data, remaining data can be used as testing data)

```
train_data = df[df['Date'] <= '2022-12-31']  # Data up to the end of 2022
test_data = df[df['Date'] > '2022-12-31']    # Data from 2023 onwards

train_data.shape
```

```
(2329, 6)
```

```
train_data.reset_index(drop=True, inplace=True)
test_data.reset_index(drop=True, inplace=True)

train_data.shape
```

```
(2329, 6)
```

```
df.head(5)
```

	Date	Close/Last	Volume	Open	High	Low
2493	2013-01-11	18.5725	274097882	$18.715	$18.7429	$18.4229
2515	2013-02-10	17.4843	287196445	$17.3439	$17.5643	$17.2768
2473	2013-02-12	19.6868	471672691	$19.9286	$20.1546	$19.6722
2514	2013-03-10	17.2646	315209314	$17.5182	$17.5839	$17.1692
2472	2013-03-12	20.2258	449352780	$19.9393	$20.2279	$19.9172

3. Apply polynomial fit to predict the current stock value from the past values.

$$Y_t = \beta_{1,t-1}Y_{t-1} + \beta_{2,t-1}Y_{t-1}^2 + - - - - - + \beta_{p-1,t-k}Y_{t-k}^{p-1} + \beta_{p,t-k}Y_{t-k}^{p}$$

4. Add more features using the higher powers of the past stock values in the dataset. We have two parameters in hand i) k- past window length ii)p-order of the polynomial. Play with $k = 1, 2, 3$ and $p = 1, 2, 3$. k and p should be atleast 3.

5. Computer the prediction accuracy for the test data and plot the predicted stock value and original stock value Vs time on a single plot. (For all values of k and p). i) Observe the change in parameter values β w.r.t p and k ii) comment on bias and variance.

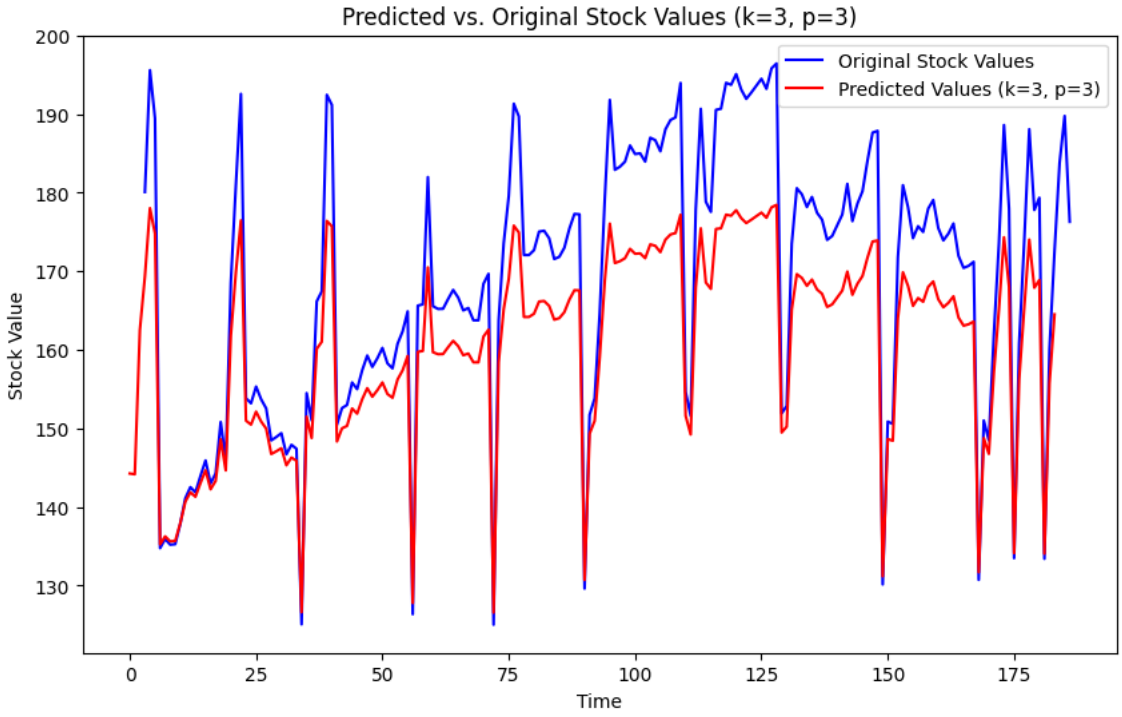
```
def plotfig( y , pred , k , p ):
    plt.figure(figsize=(10, 6))
    plt.plot(y, label='Original Stock Values', color='blue')

    plt.plot(pred, label=f'Predicted Values (k={k}, p={p})', color='red')

    plt.xlabel('Time')
    plt.ylabel('Stock Value')
    plt.title(f'Predicted vs. Original Stock Values (k={k}, p={p})')
    plt.legend()
    plt.show()
```

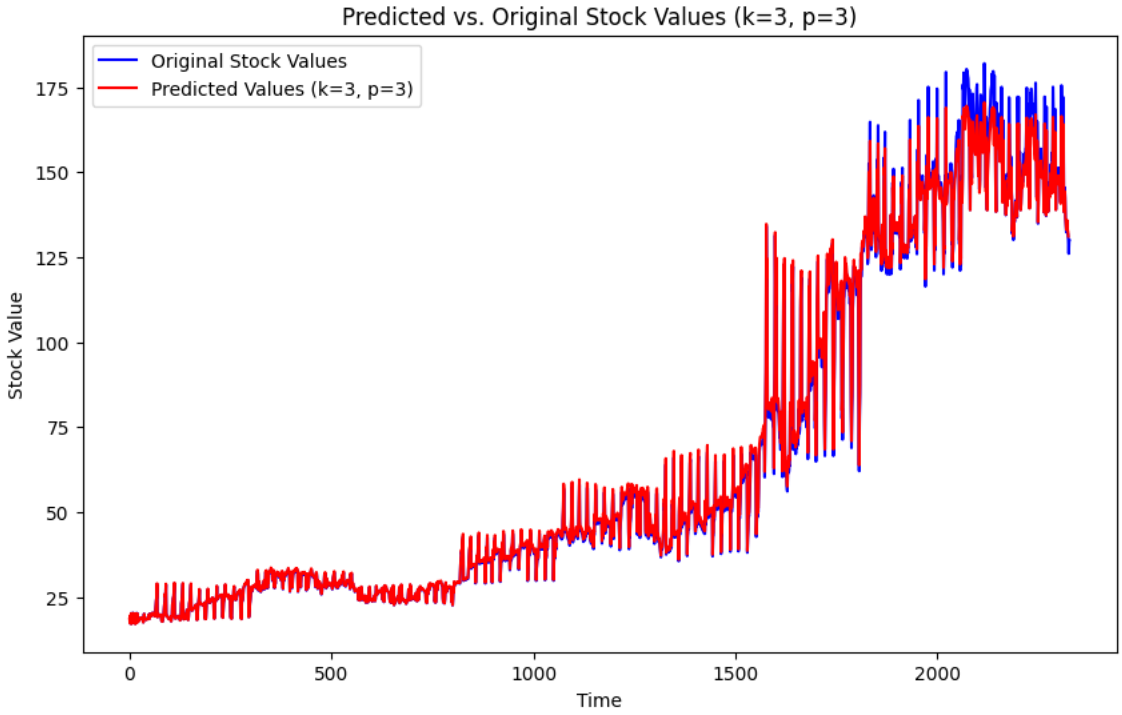
```
import numpy as np
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt



# Define ks and ps
```

```
ks = [3, 4, 5, 7, 9, 10]
ps = [3, 4, 5, 6, 8, 10]

# taking three cases
# k = 3 , p = 3 i.e high bias
# k = 5 , p= 5 i.e. generalized
# k = 10 , p = 10 , i.e. high variance

k3p3_test = np.array([])
k3p3_train = np.array([])
k5p5_test = np.array([])
k5p5_train = np.array([])
k10p10_test = np.array([])
k10p10_train = np.array([])

te =[]
tse = []

pred_error_p = []
pred_error_k = []



thetas = [ [] ]

test_errors = []
train_errors = []



for k in ks:
    for p in ps:

        X_train = np.zeros((len(train_data) - k, p))
        y_train = train_data['Close/Last'][k:]

        X_test = np.zeros((len(test_data) - k, p))
        y_test = test_data['Close/Last'][k:]

        for kk in range(k):
            for pp   in range(p):
                X_train[:, pp] = train_data['Close/Last'][k - kk - 1:-kk - 1] ** (pp + 1)
                X_test[:, pp] = test_data['Close/Last'][k - kk - 1:-kk - 1] ** (pp + 1)

        # Fit the model
        theta = np.linalg.inv(X_train.T @ X_train) @ (X_train.T @ y_train)


        pred_train = X_train @ theta
        pred_test = X_test @ theta

        # errors
        squared_diff_train = (y_train - pred_train) ** 2
        train_error = np.mean(squared_diff_train)

        squared_diff_test = (y_test - pred_test) ** 2
        test_error = np.mean(squared_diff_test)

        train_errors.append(train_error)
        test_errors.append(test_error)

        print(f"k = {k}, p = {p}")
        print(f"Training error: {train_error}")
        print(f"Testing error: {test_error}")

        plotfig(y_test , pred_test , k , p  )
        plotfig(y_train , pred_train , k , p)

        if k == 3 and p == 3 :
          k3p3_test = pred_test
          k3p3_train = pred_train
          te.append(train_error)
          tse.append(test_error)

        if k == 5 and p == 5:
          k5p5_test = pred_test
          k5p5_train = pred_train
          te.append(train_error)
          tse.append(test_error)


        if k == 10 and p == 10:
```

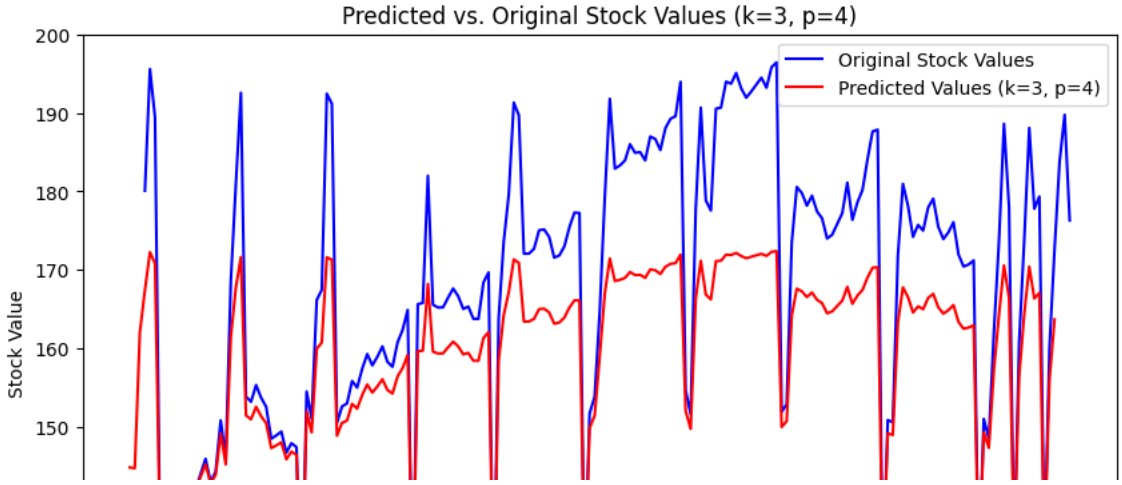
```
      k10p10_test = pred_test
      k10p10_train = pred_train
      te.append(train_error)
      tse.append(test_error)


    thetas.append(theta)

    if k == 5:
      pred_error_p.append(test_error)

    if p == 5:
      pred_error_k.append(test_error)
```

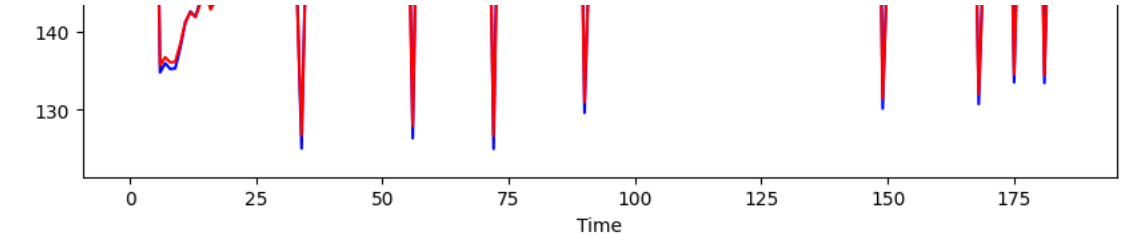
```
k = 3, p = 3
Training error: 104.35061822387826
Testing error: 418.2981912312131
```

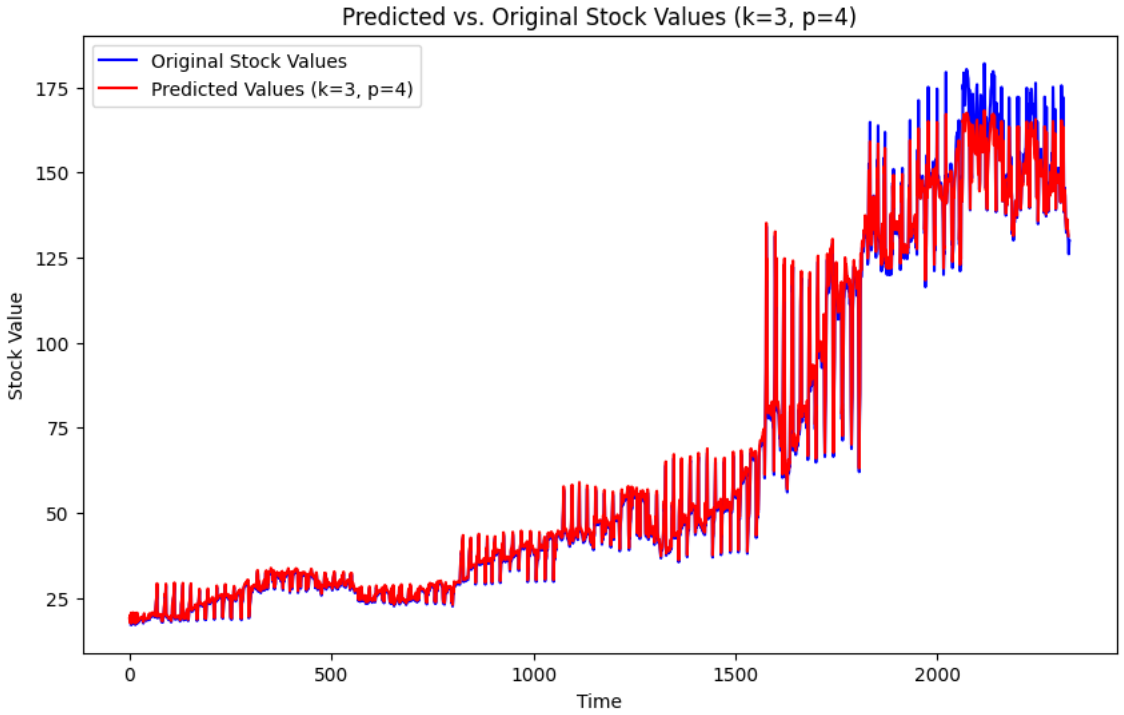




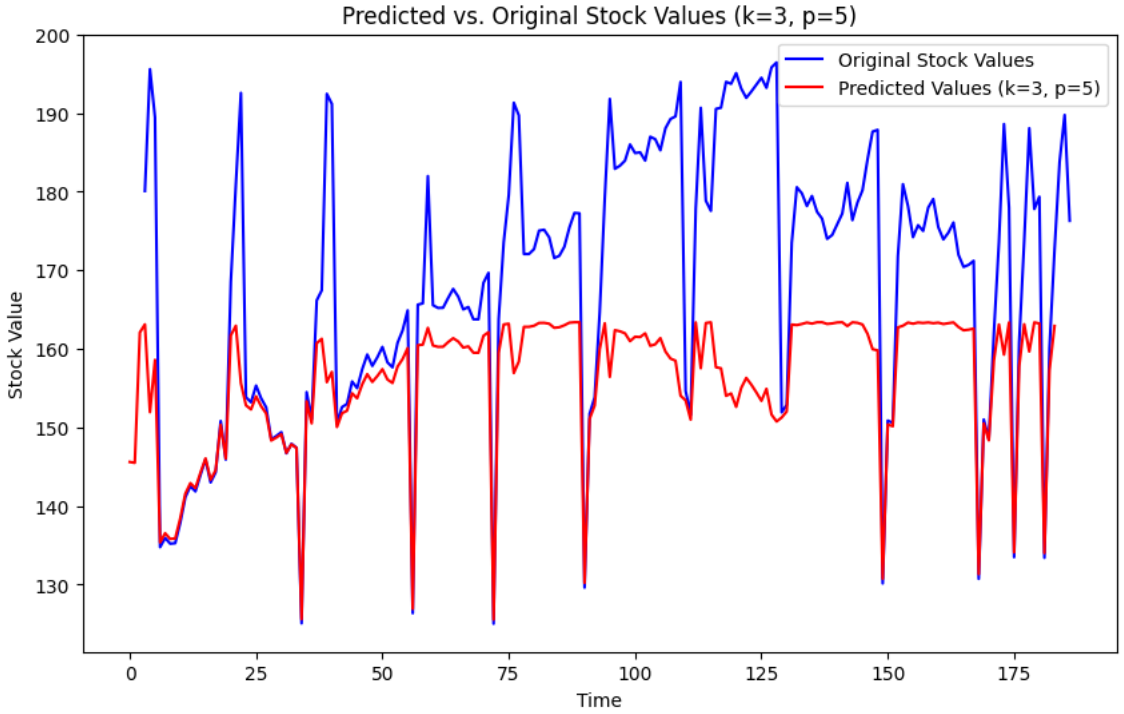

```
k = 3, p = 4
Training error: 104.156439182689
Testing error: 419.93692937472593
```

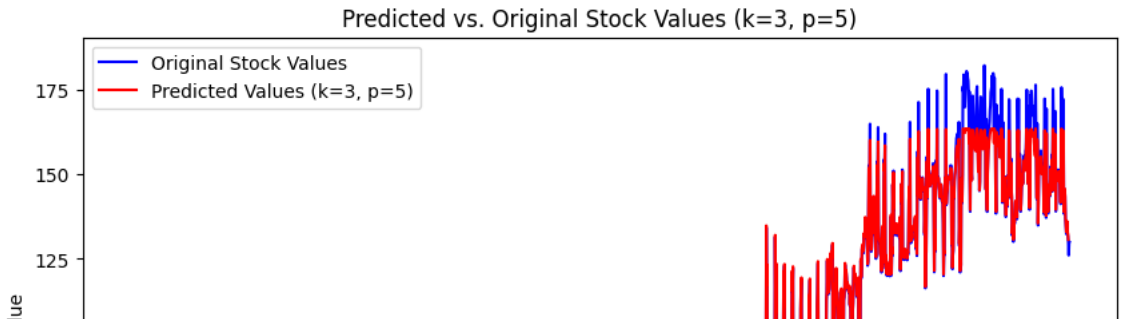


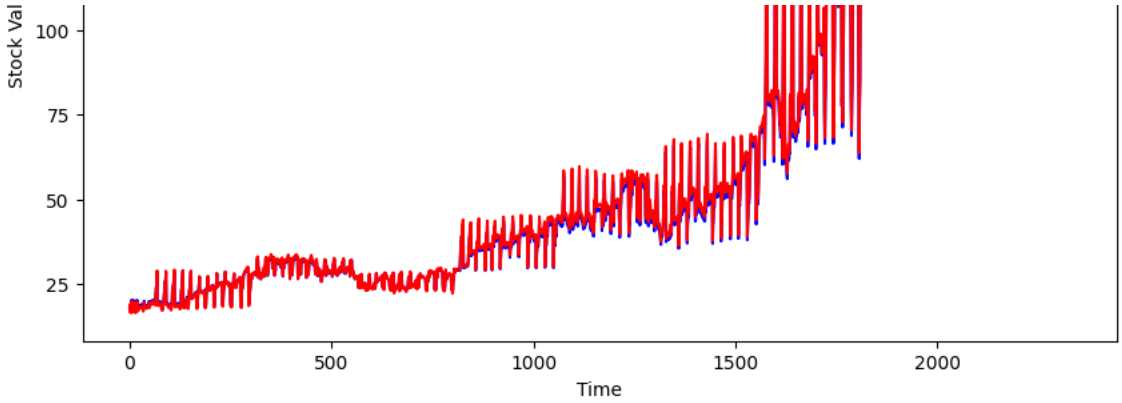




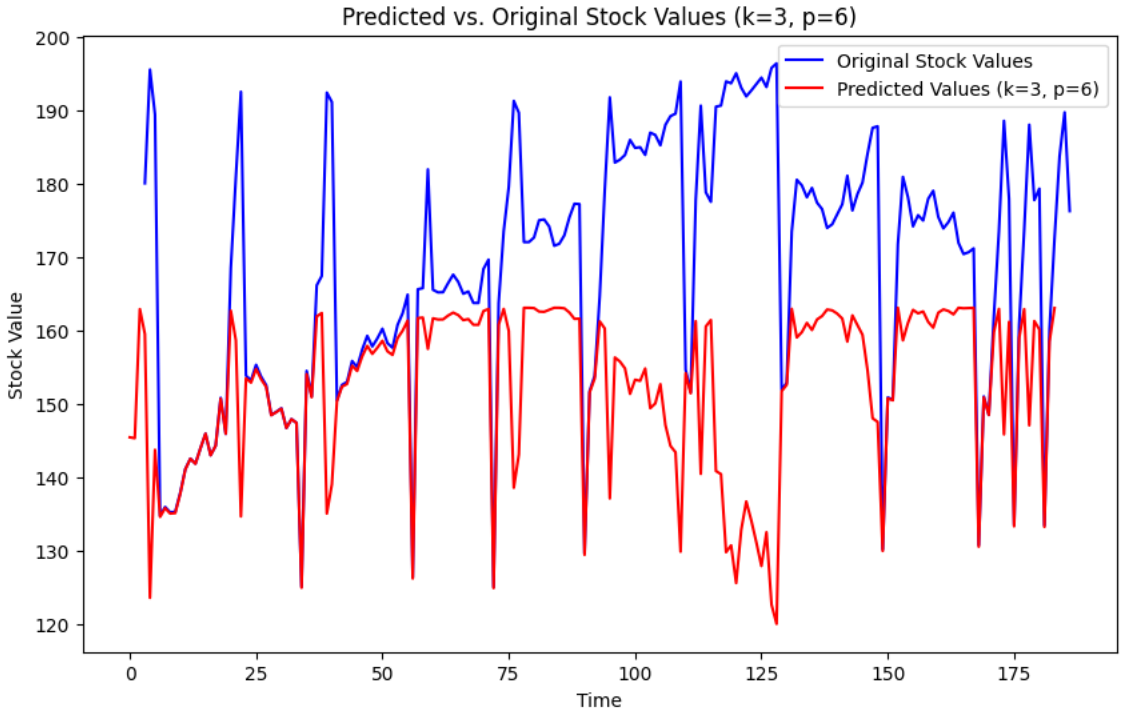

```
k = 3, p = 5
Training error: 103.50715707006074
Testing error: 485.1309157313537
```

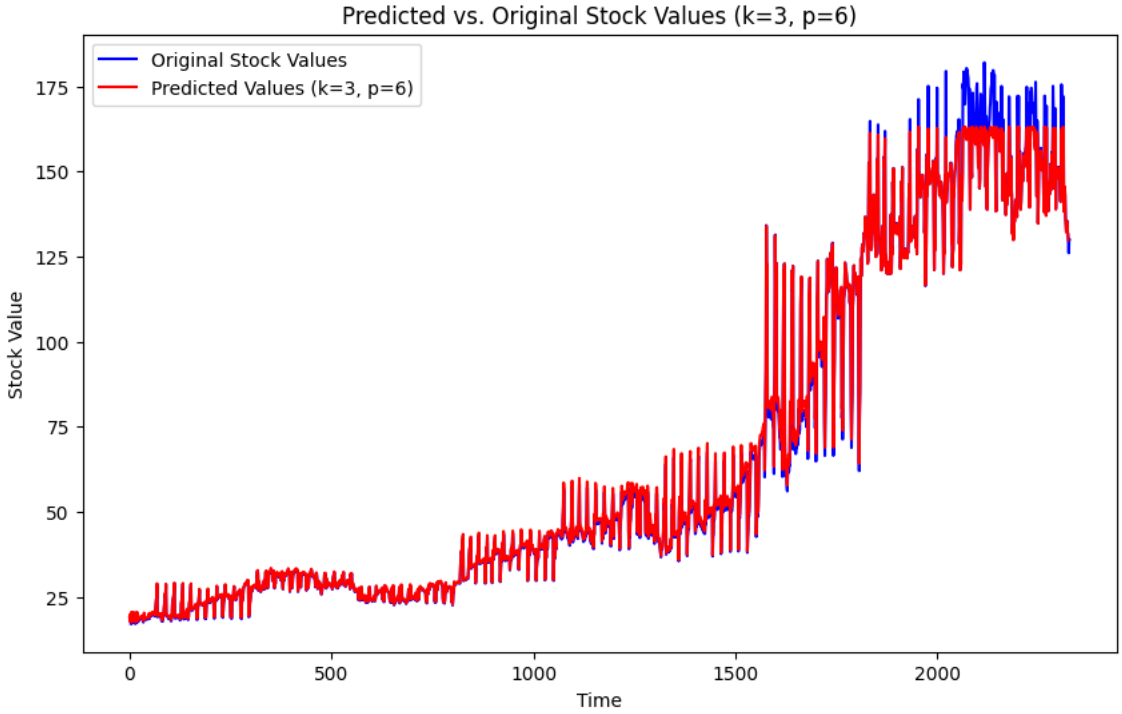




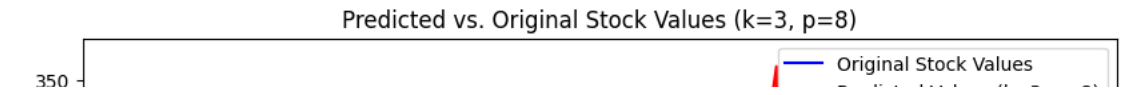


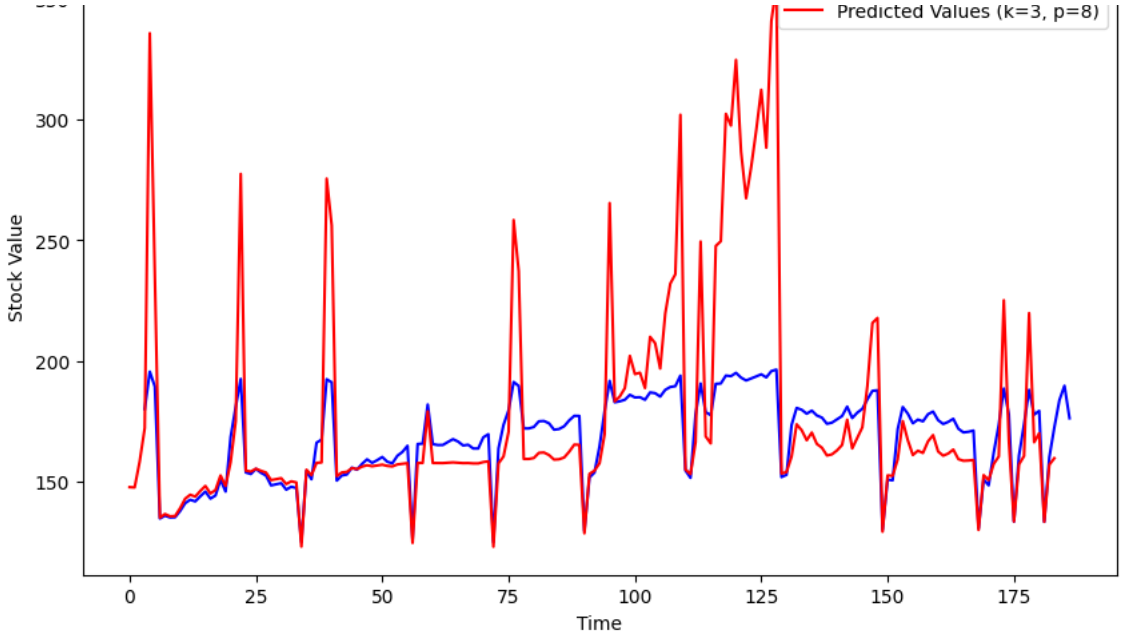

```
k = 3, p = 6
Training error: 103.1004733736073
Testing error: 699.3396580141628
```

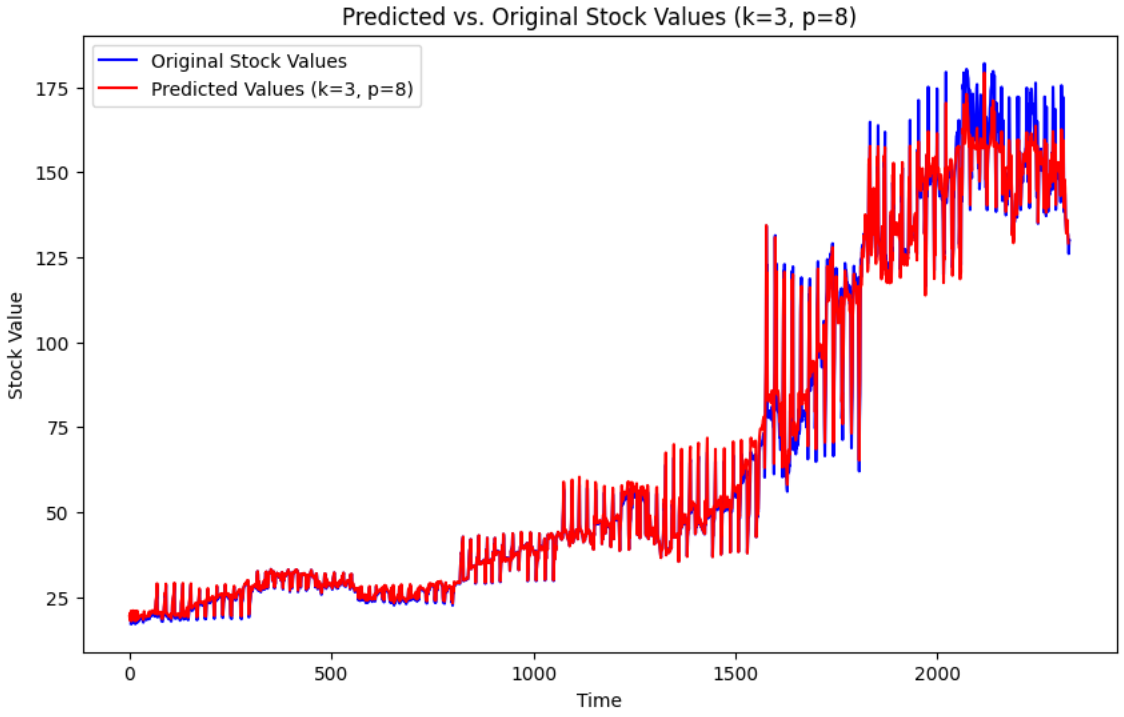




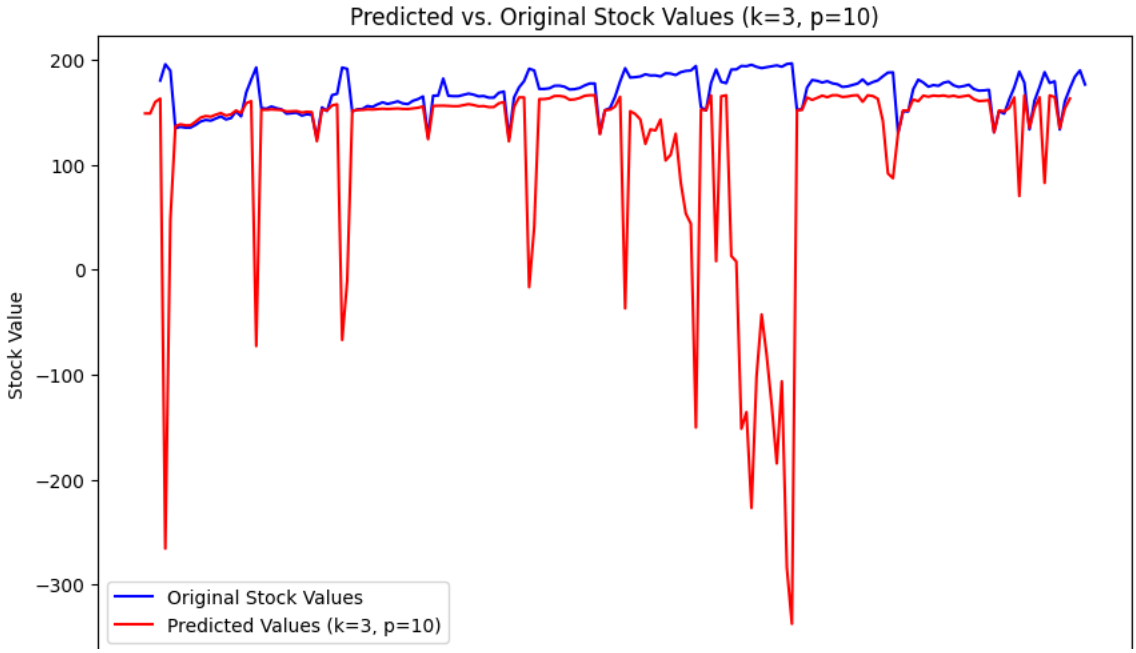

```
k = 3, p = 8
Training error: 100.90855893819274
Testing error: 2200.7199025502746
```

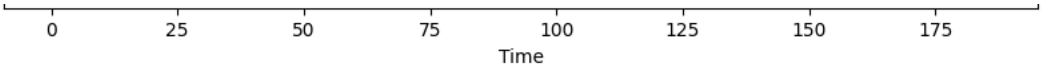


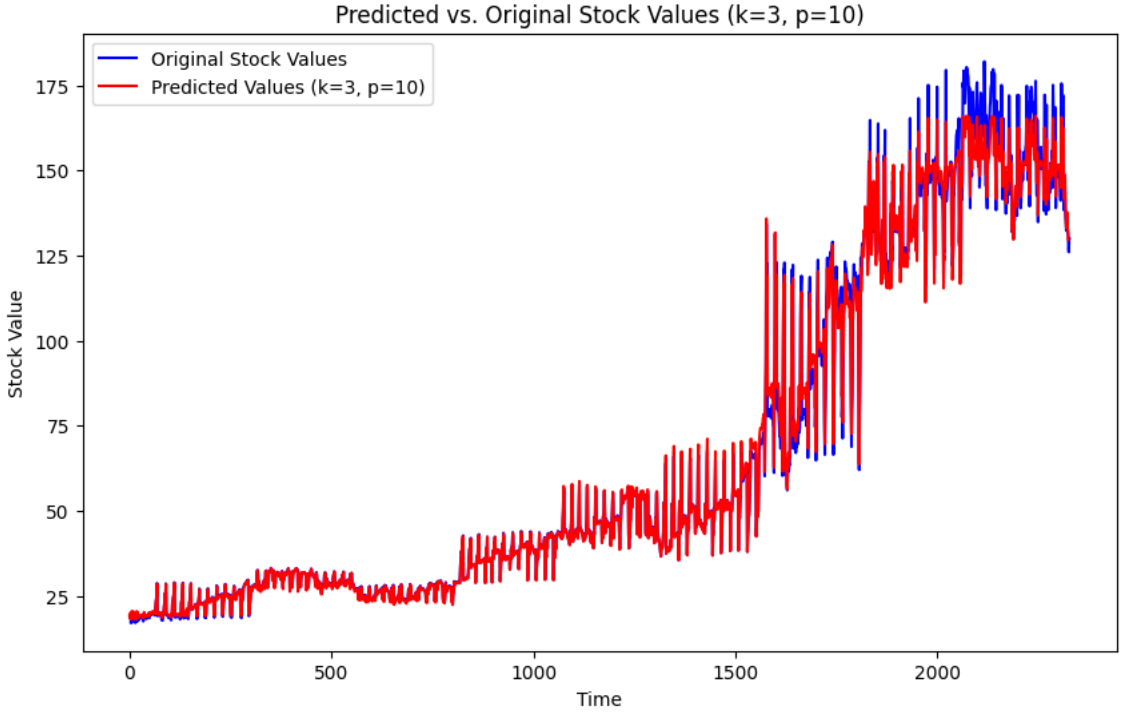




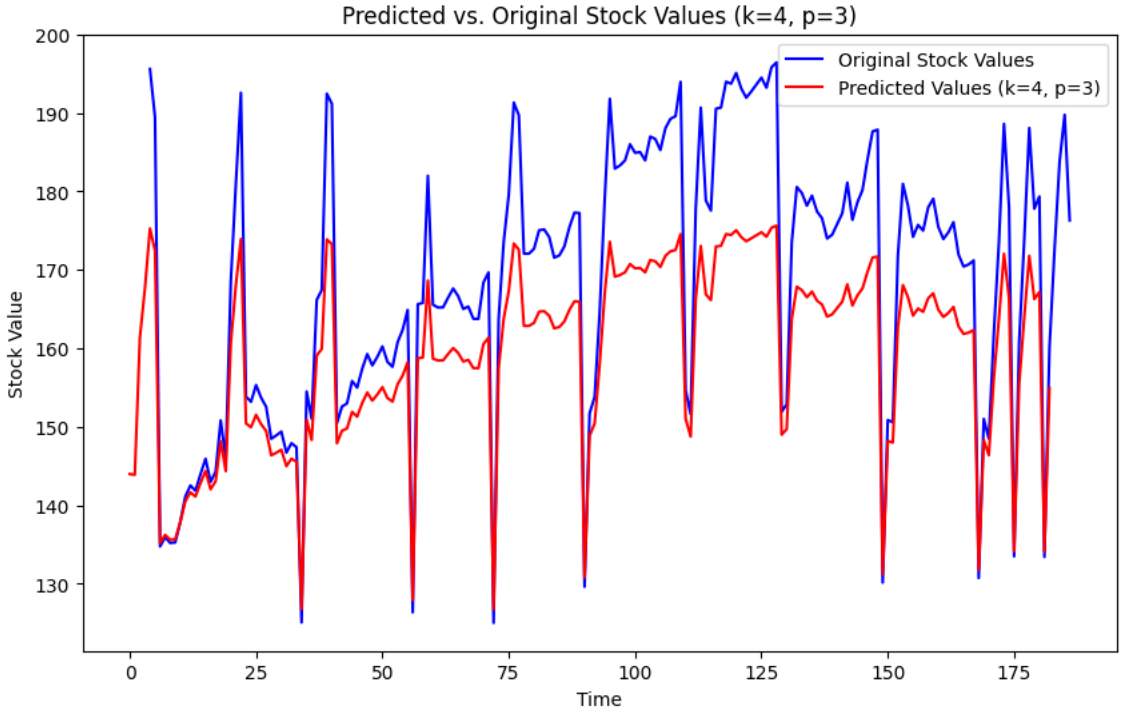

```
k = 3, p = 10
Training error: 99.56244740681957
Testing error: 11659.97826341857
```

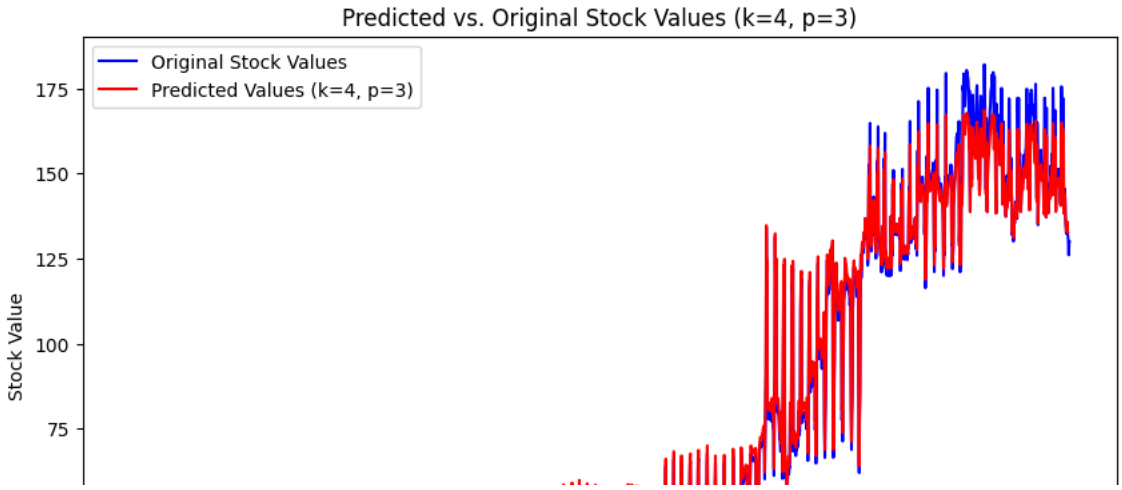


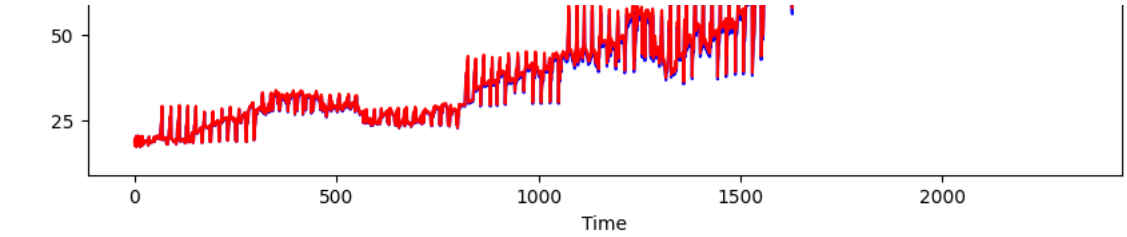




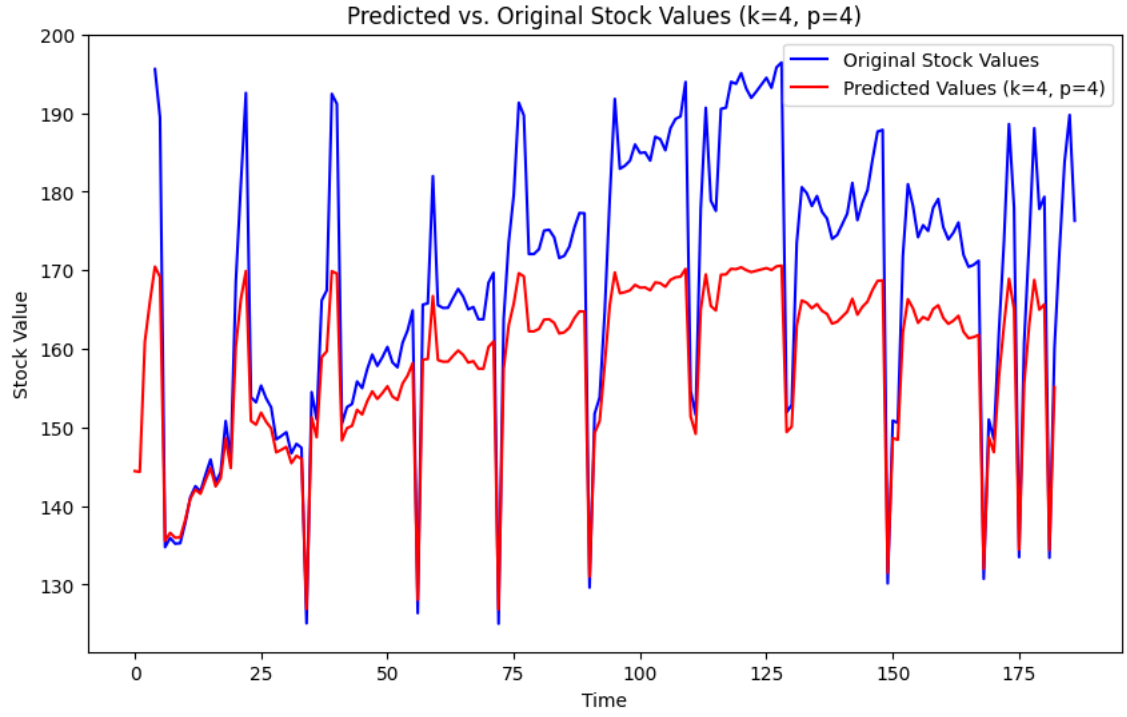

```
k = 4, p = 3
Training error: 125.49687651021092
Testing error: 437.69698368342637
```

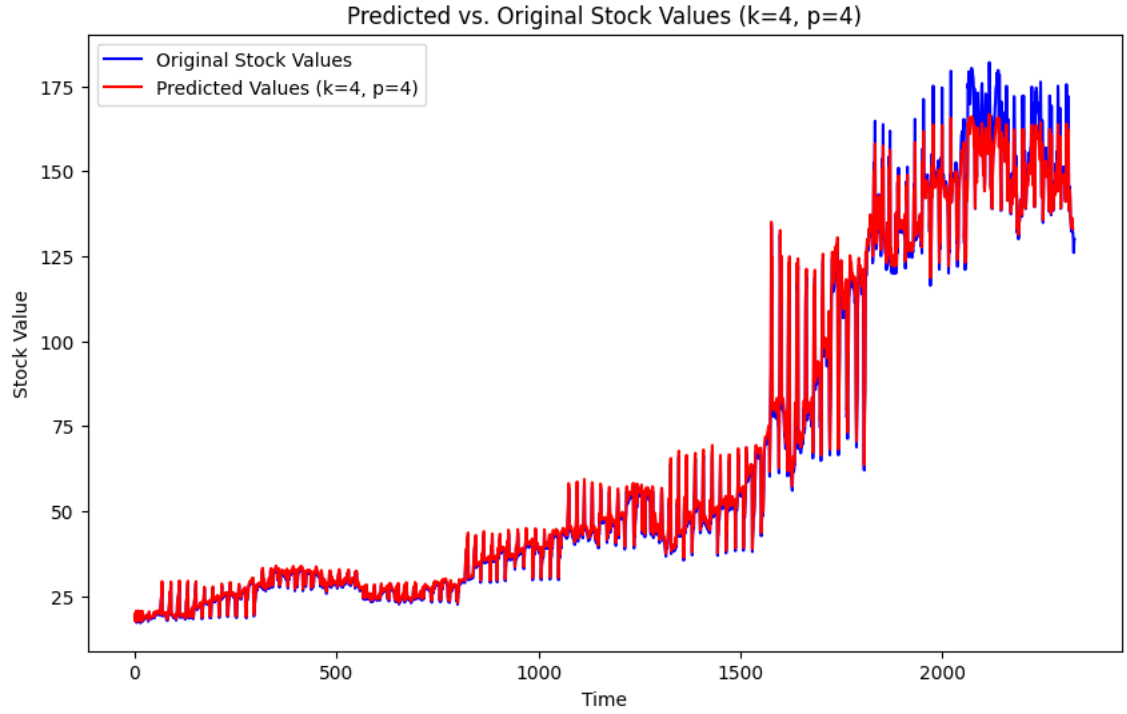




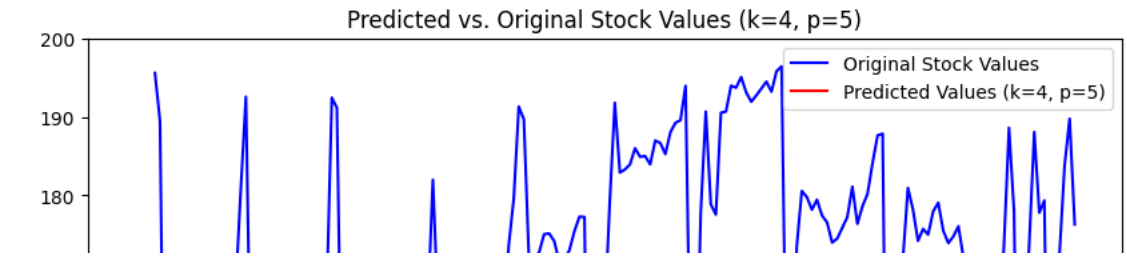


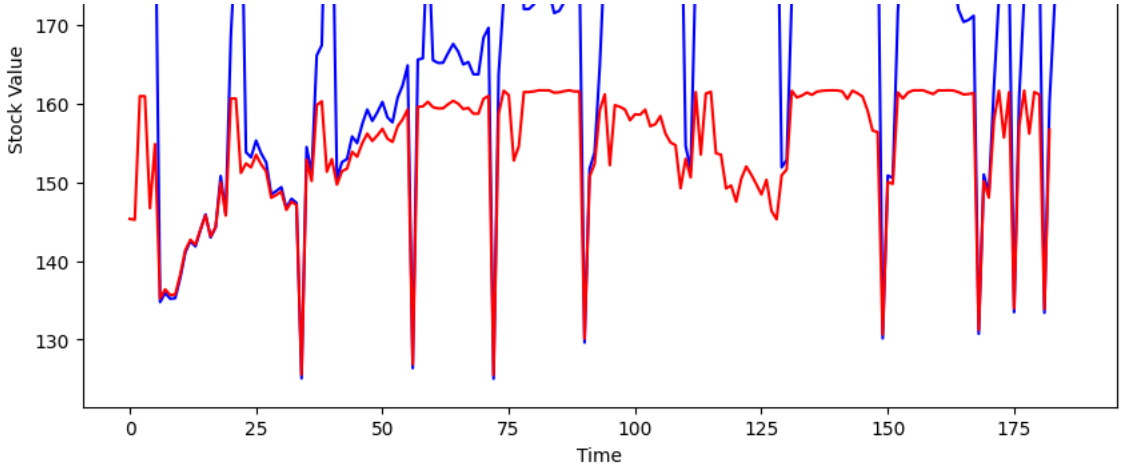

```
k = 4, p = 4
Training error: 125.36182581561926
Testing error: 444.9018243045056
```

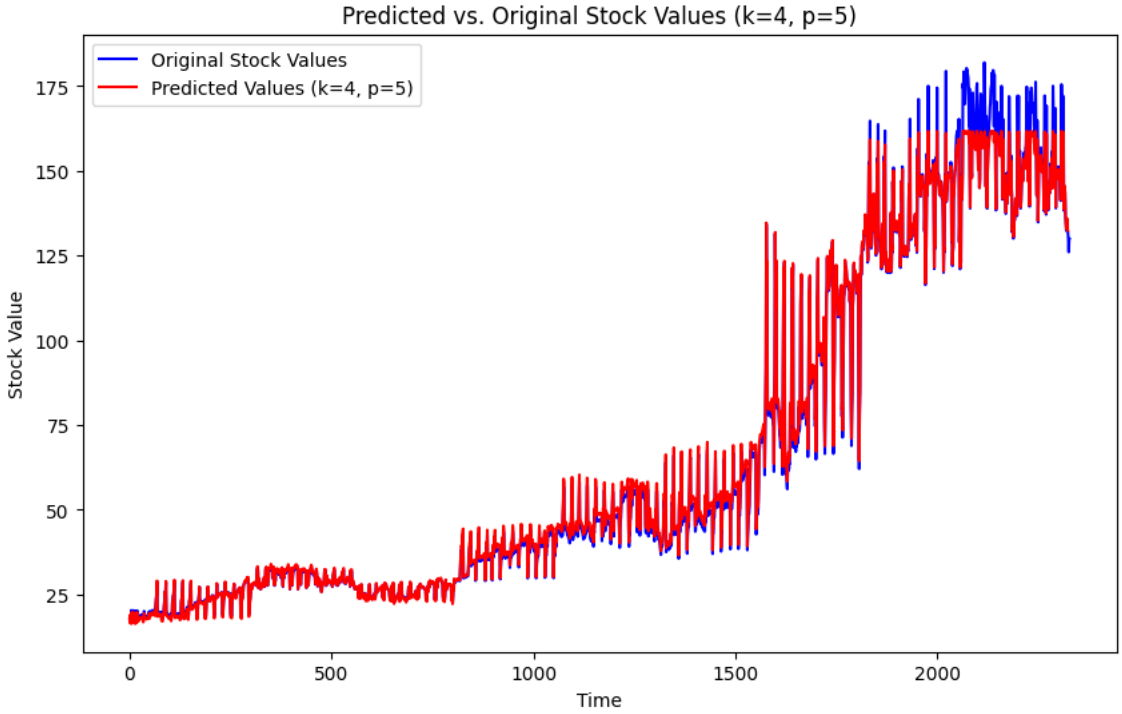




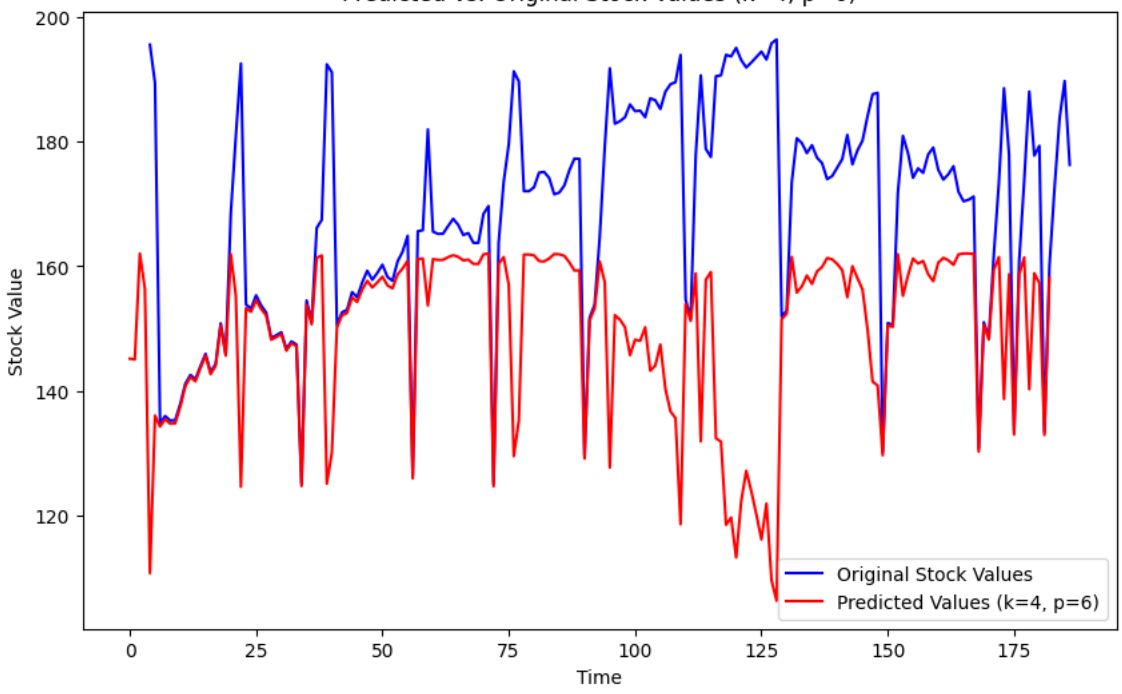

```
k = 4, p = 5
Training error: 124.47510591703555
Testing error: 555.075876148586
```

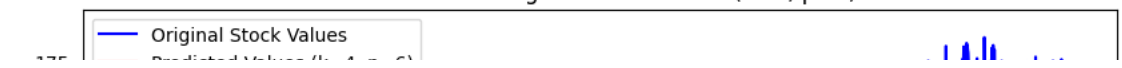


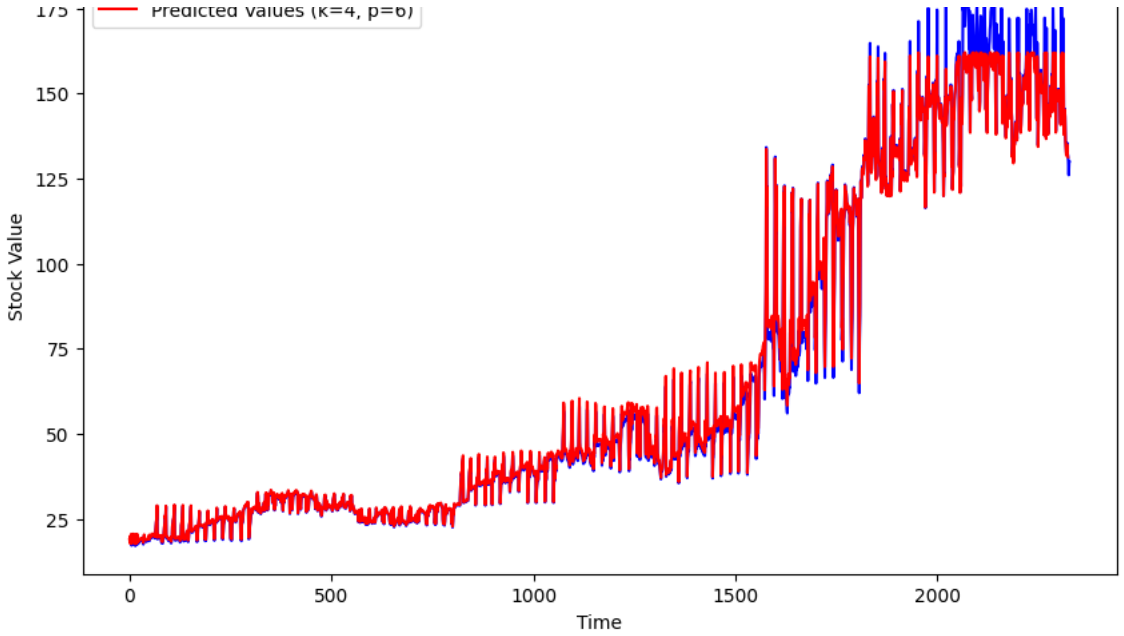




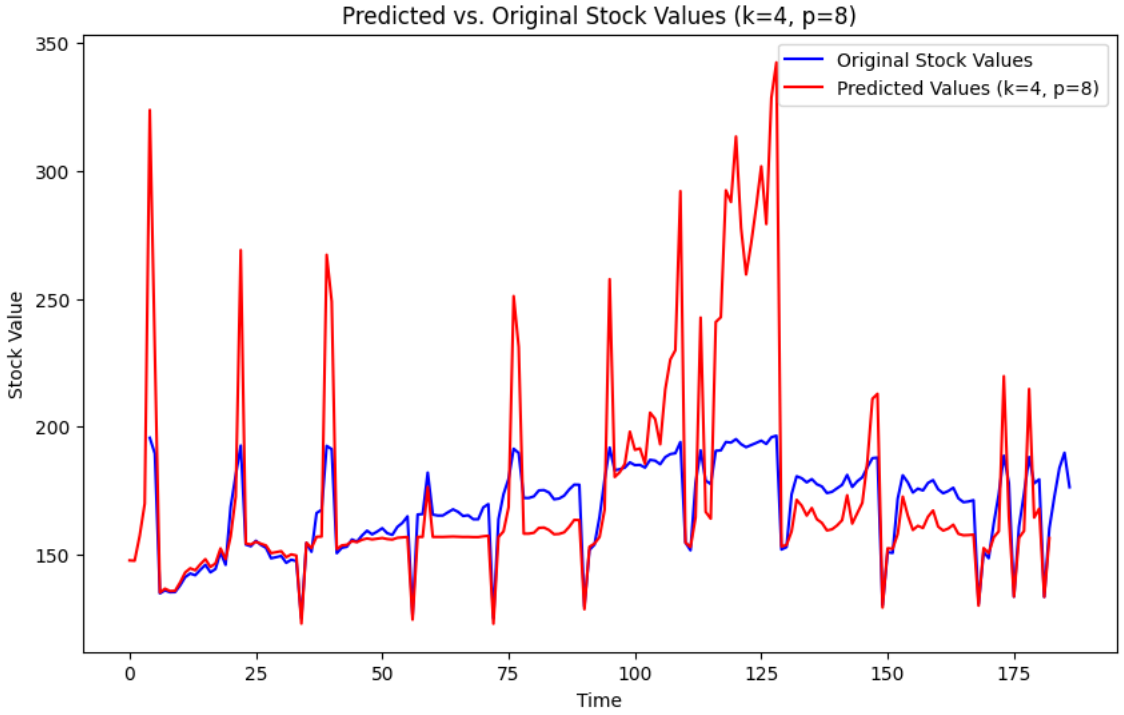

```
k = 4, p = 6
Training error: 123.8199834853926
Testing error: 912.1810208935289
```

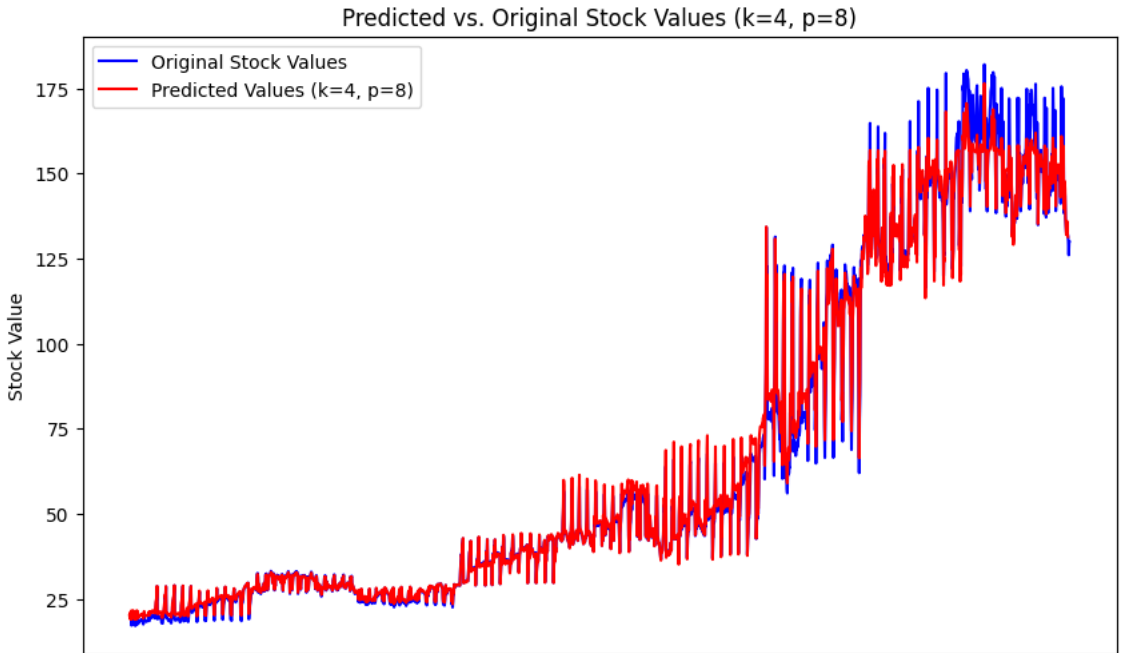


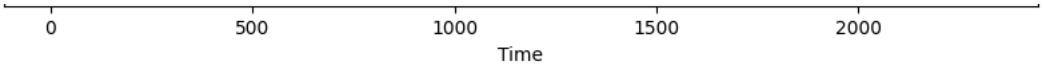




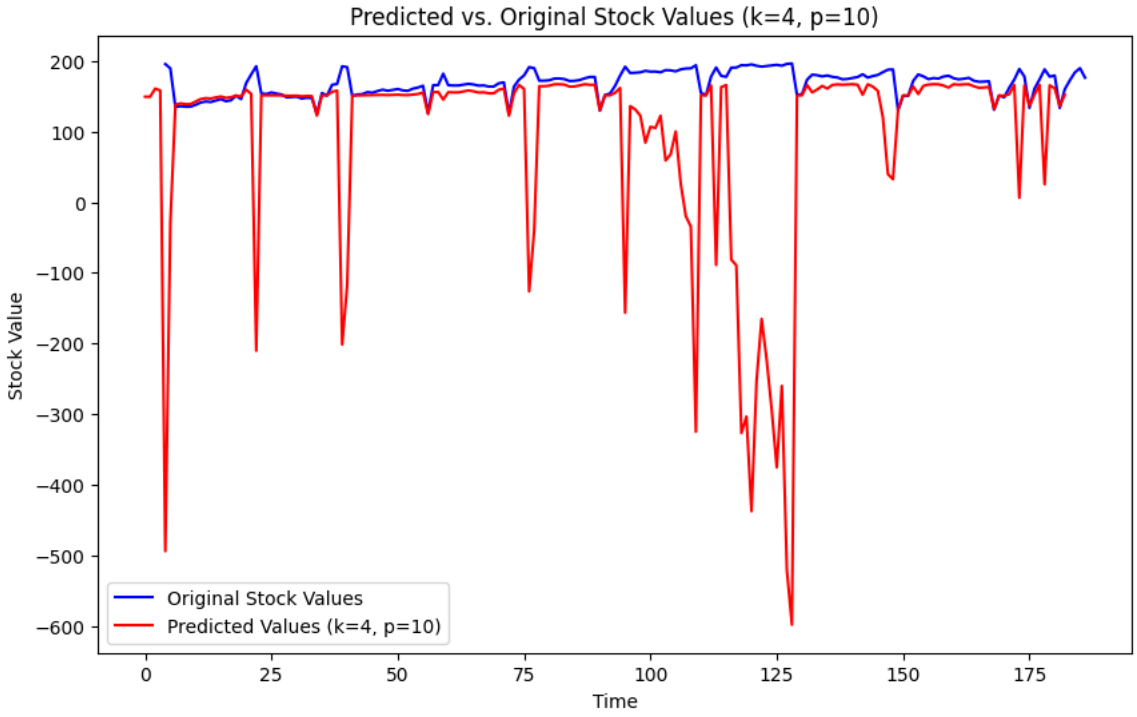

```
k = 4, p = 8
Training error: 121.15508625205123
Testing error: 1903.9779065431433
```

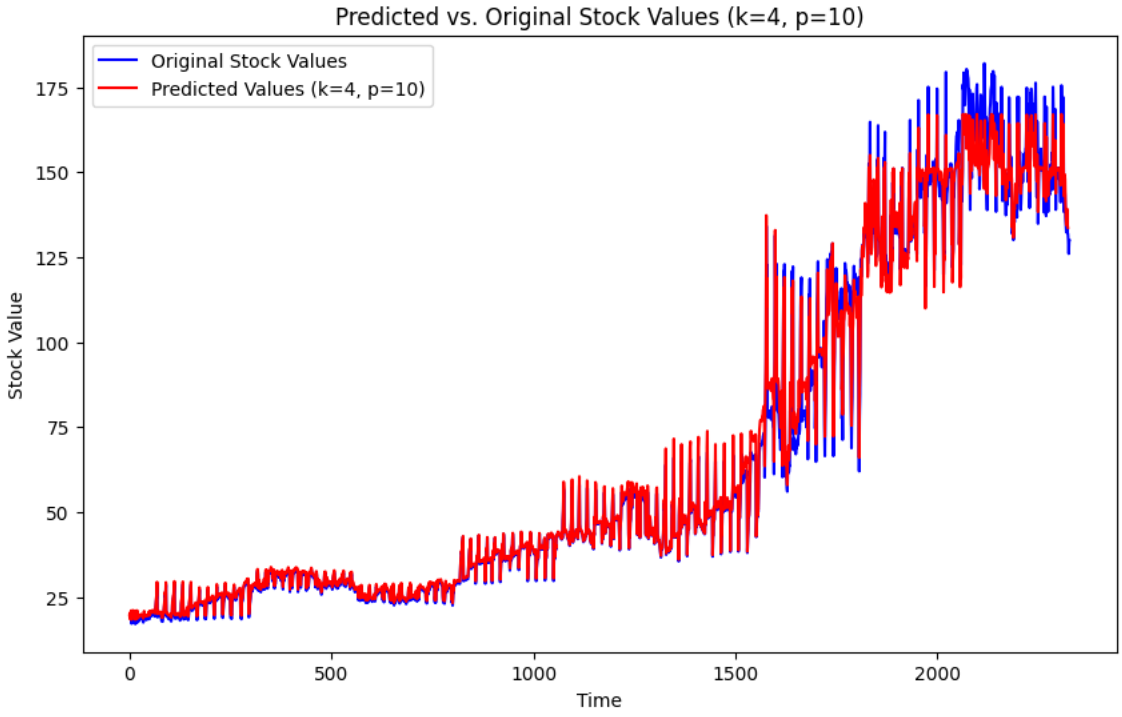




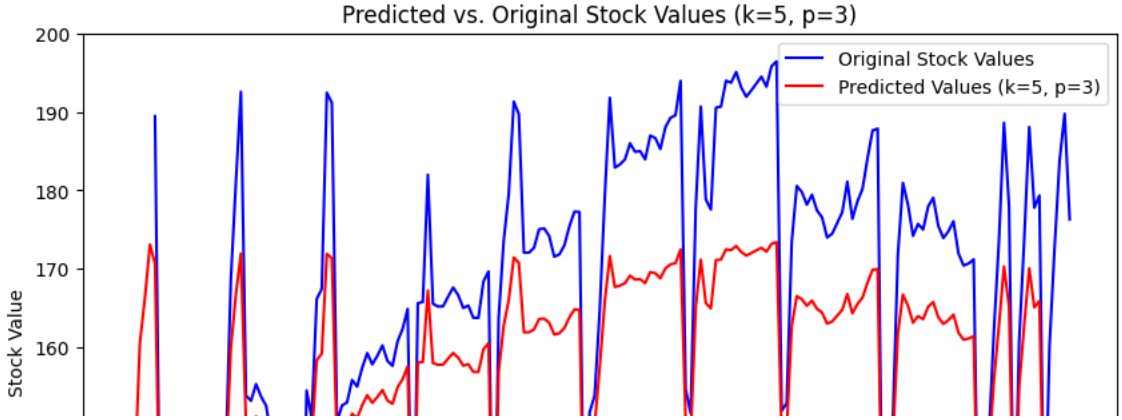


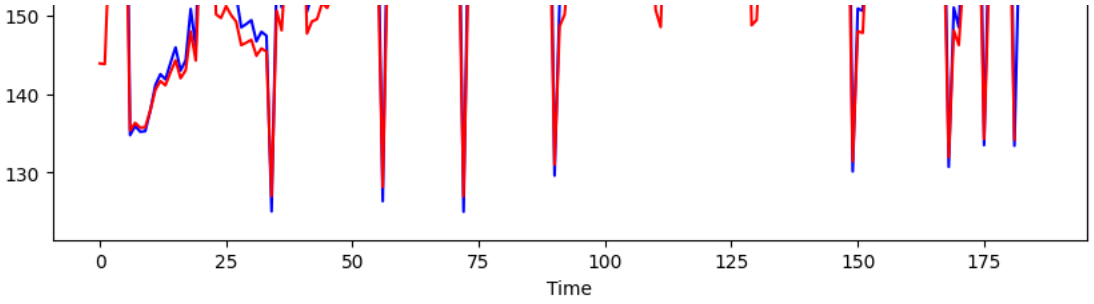

```
k = 4, p = 10
Training error: 117.84821895021562
Testing error: 27423.355120451946
```

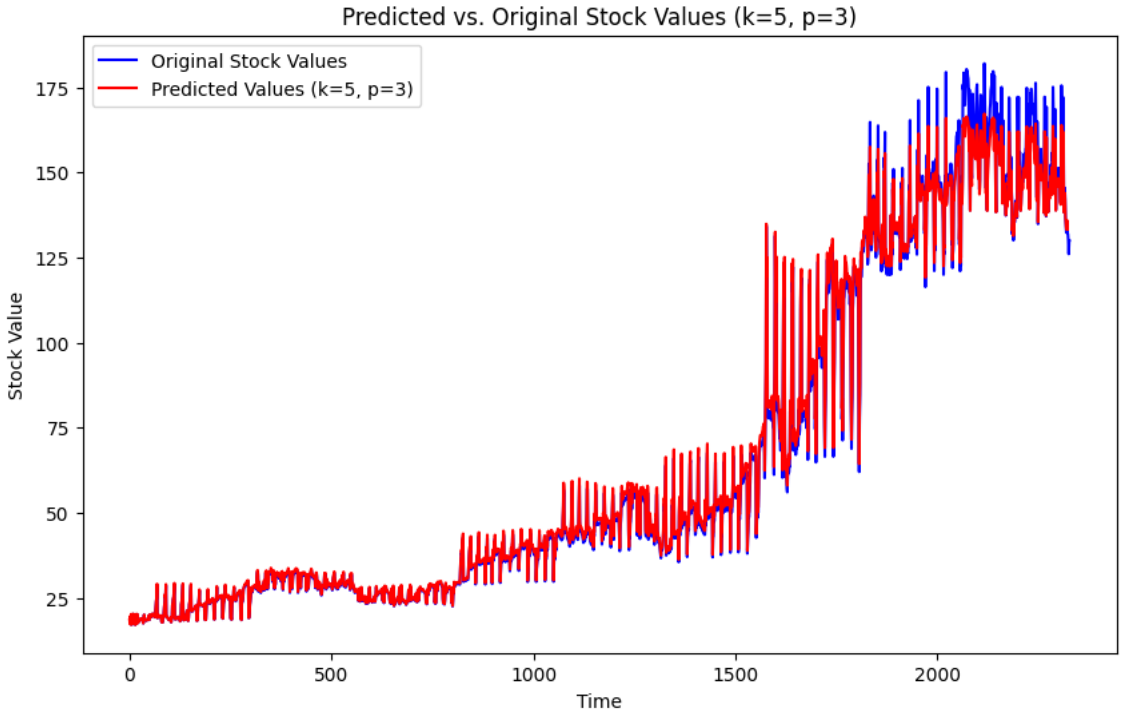




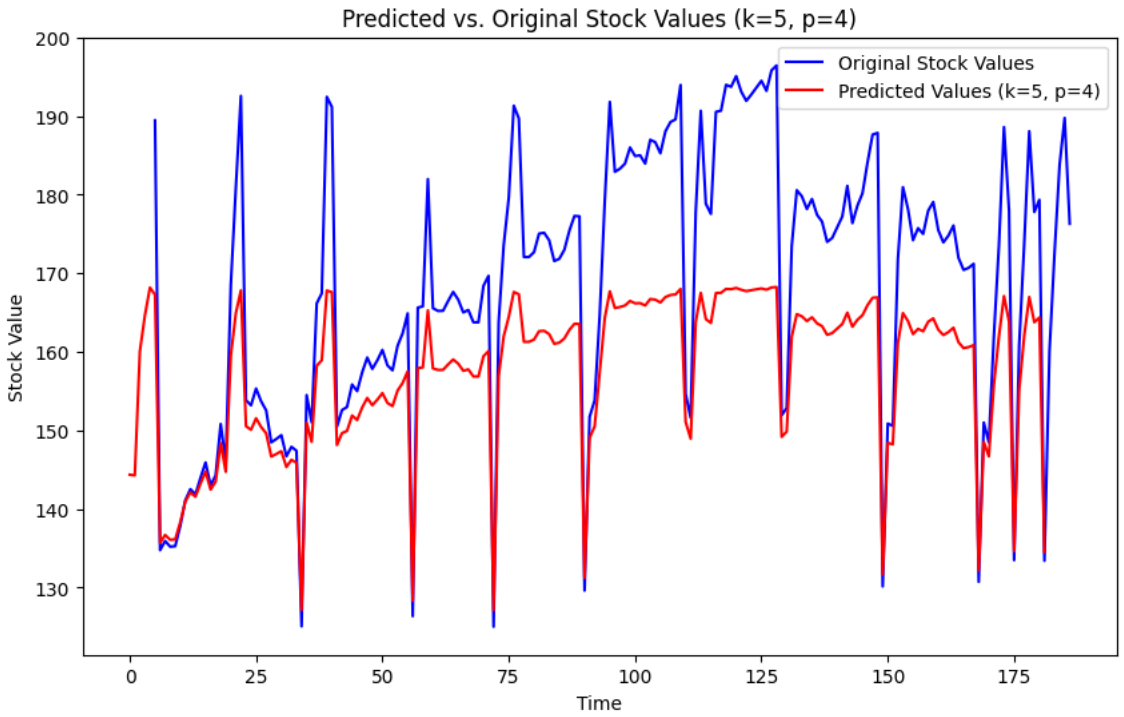

```
k = 5, p = 3
Training error: 137.05140210193744
Testing error: 432.42719489079866
```

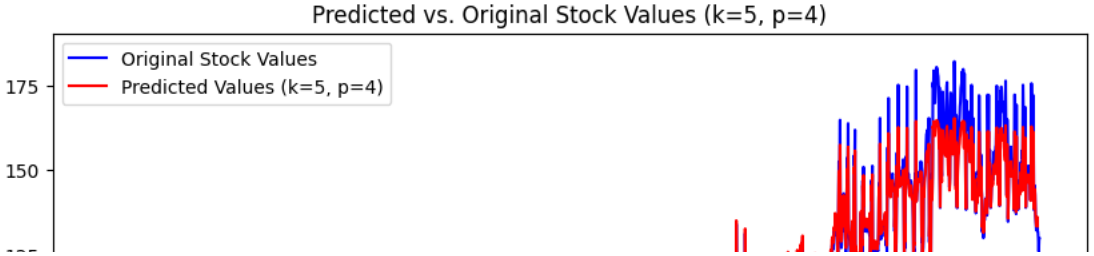


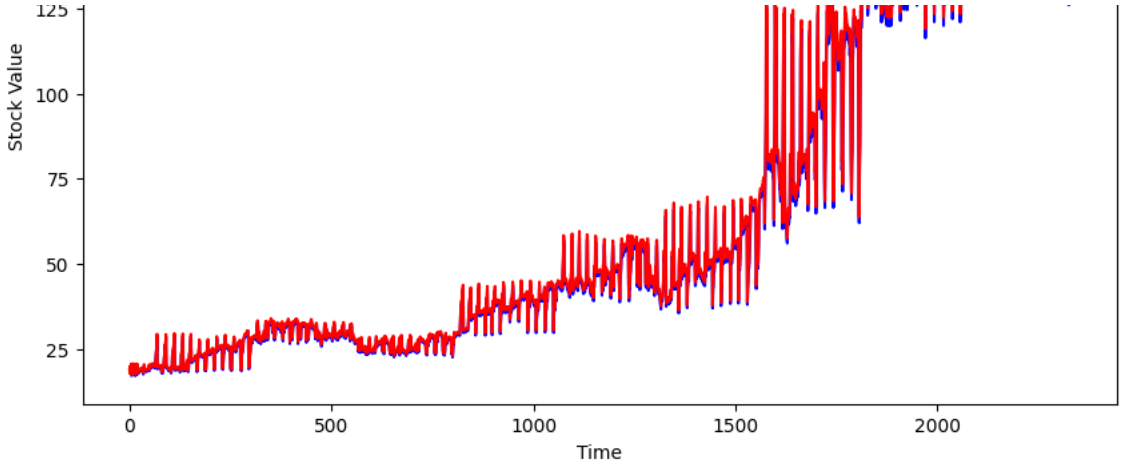




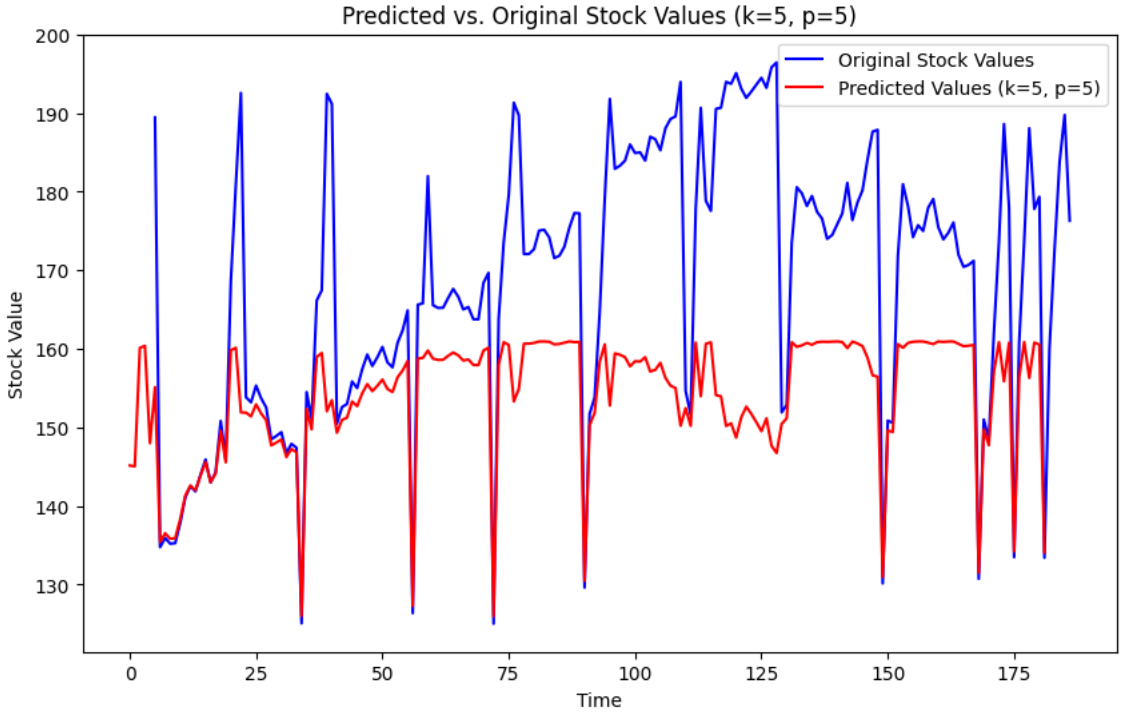

```
k = 5, p = 4
Training error: 136.91004580967368
Testing error: 446.03906288921814
```

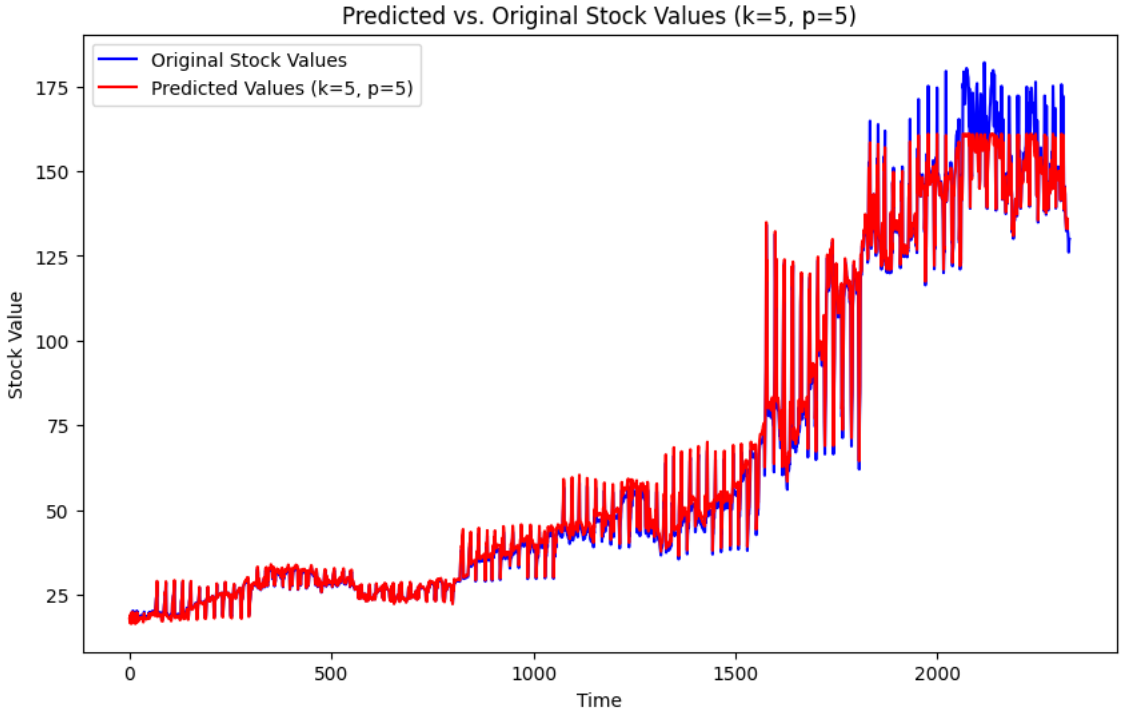




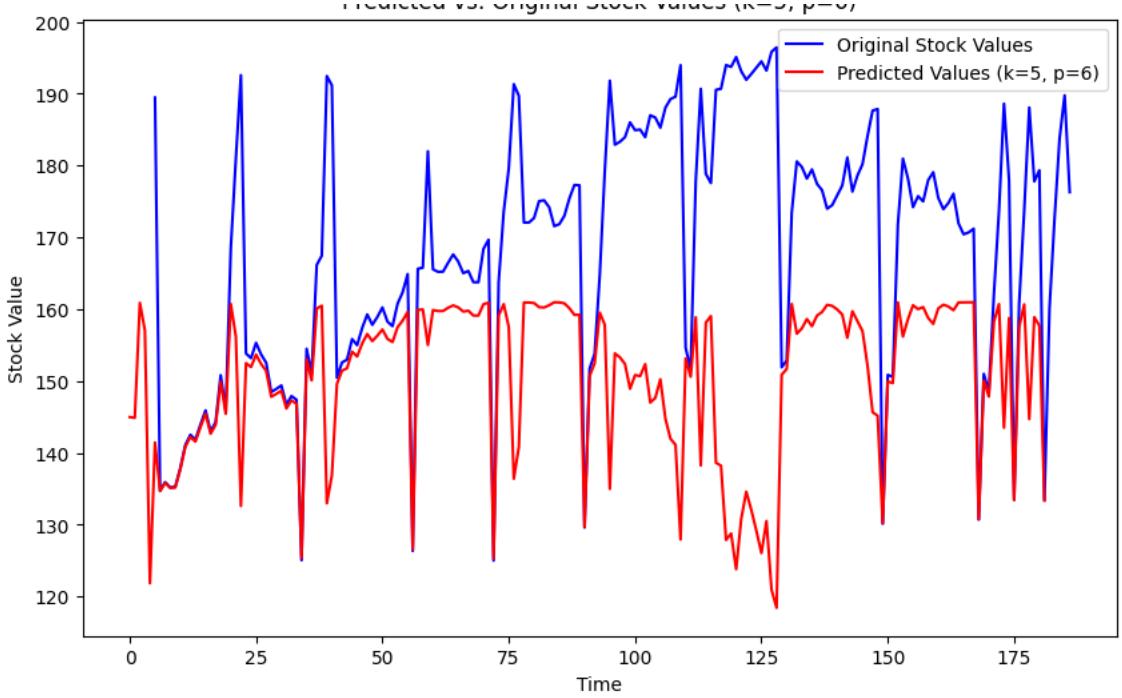


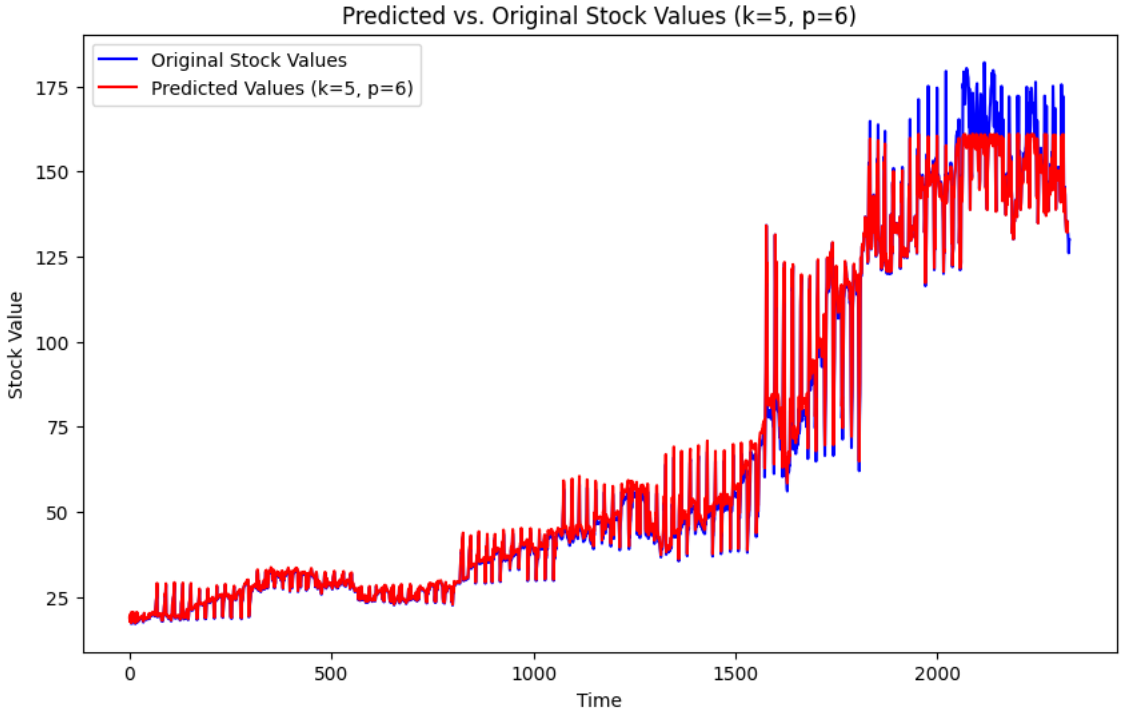

```
k = 5, p = 5
Training error: 136.2681077957716
Testing error: 553.1790501934859
```



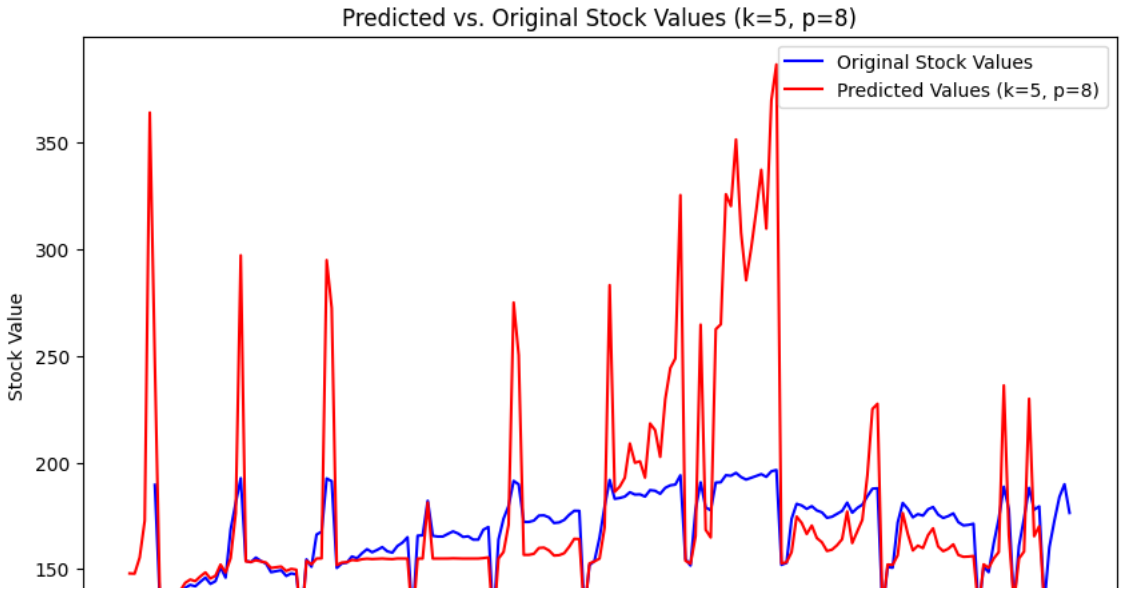


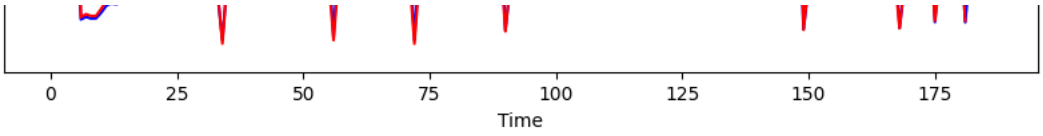

```
k = 5, p = 6
Training error: 135.92434576648154
Testing error: 791.3744405195356
```

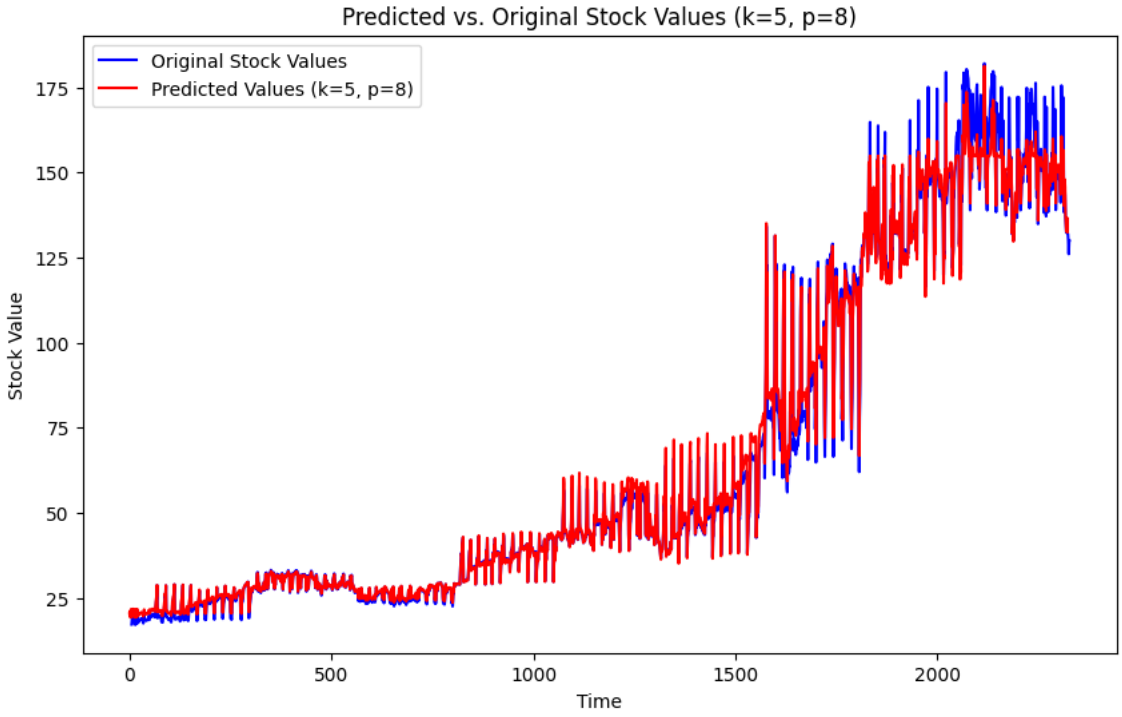




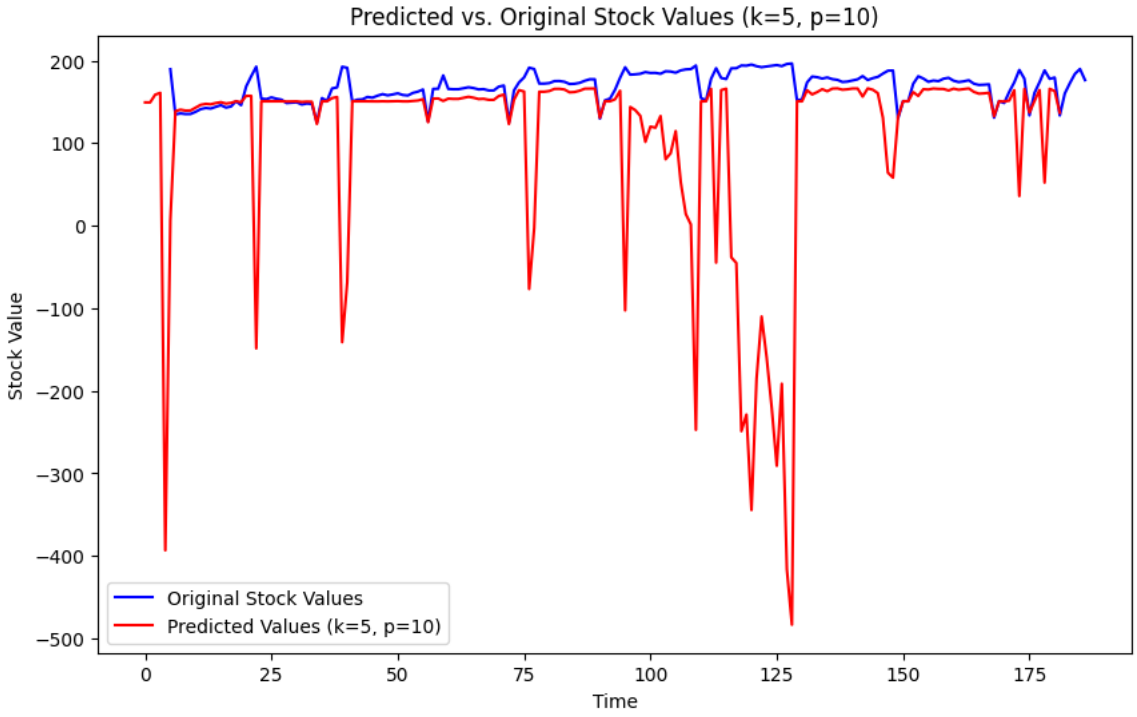

```
k = 5, p = 8
Training error: 132.23932793646625
Testing error: 2891.089882410244
```

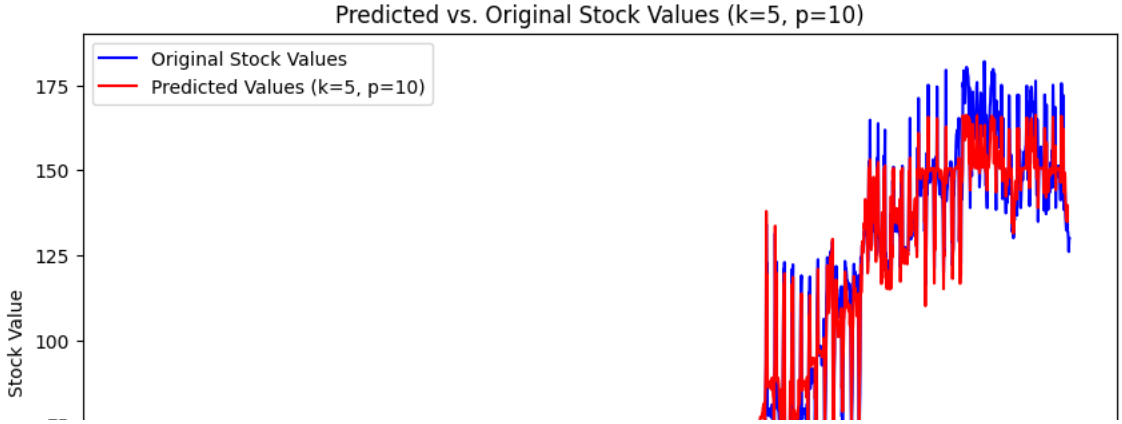


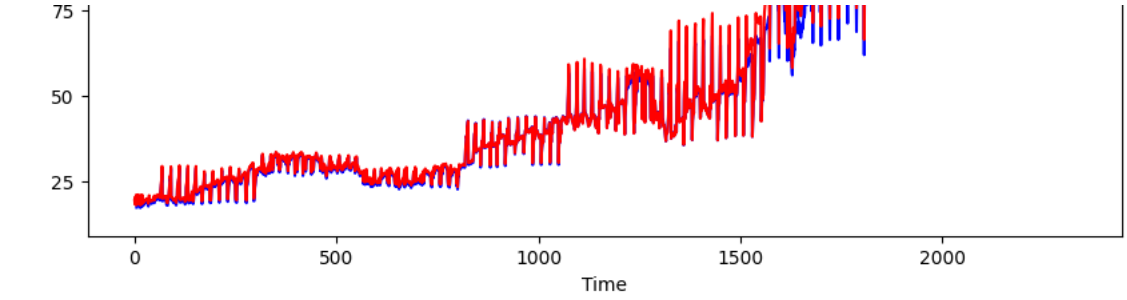




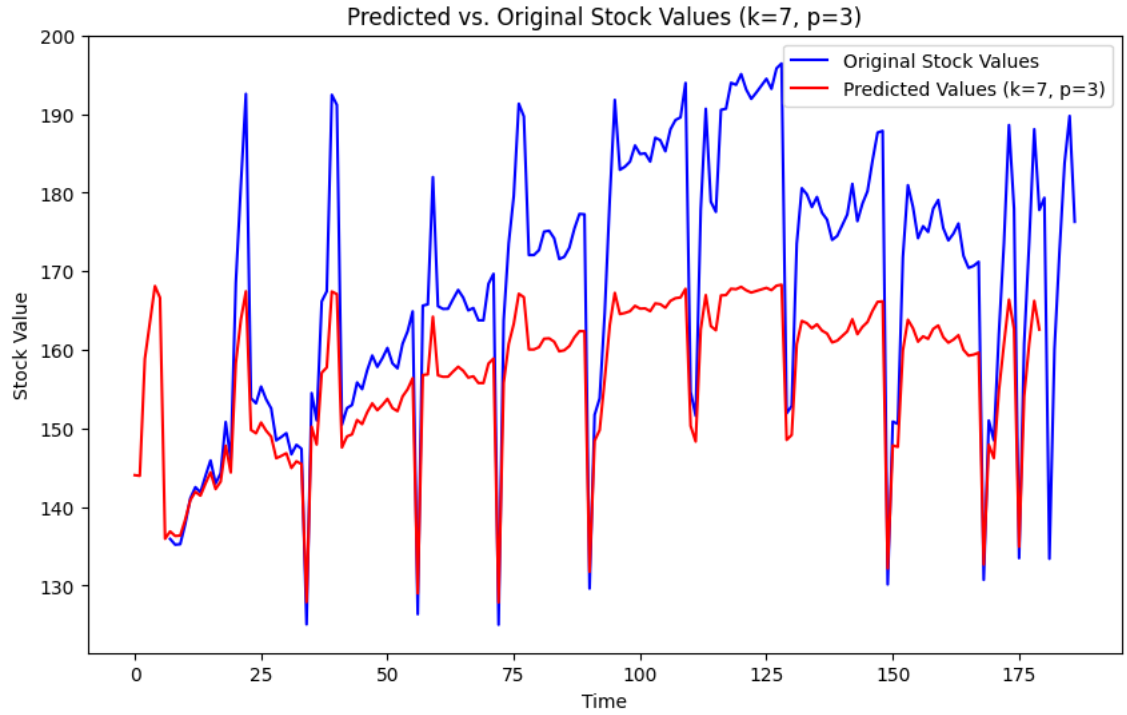

```
k = 5, p = 10
Training error: 129.057335481319
Testing error: 19761.10105816004
```

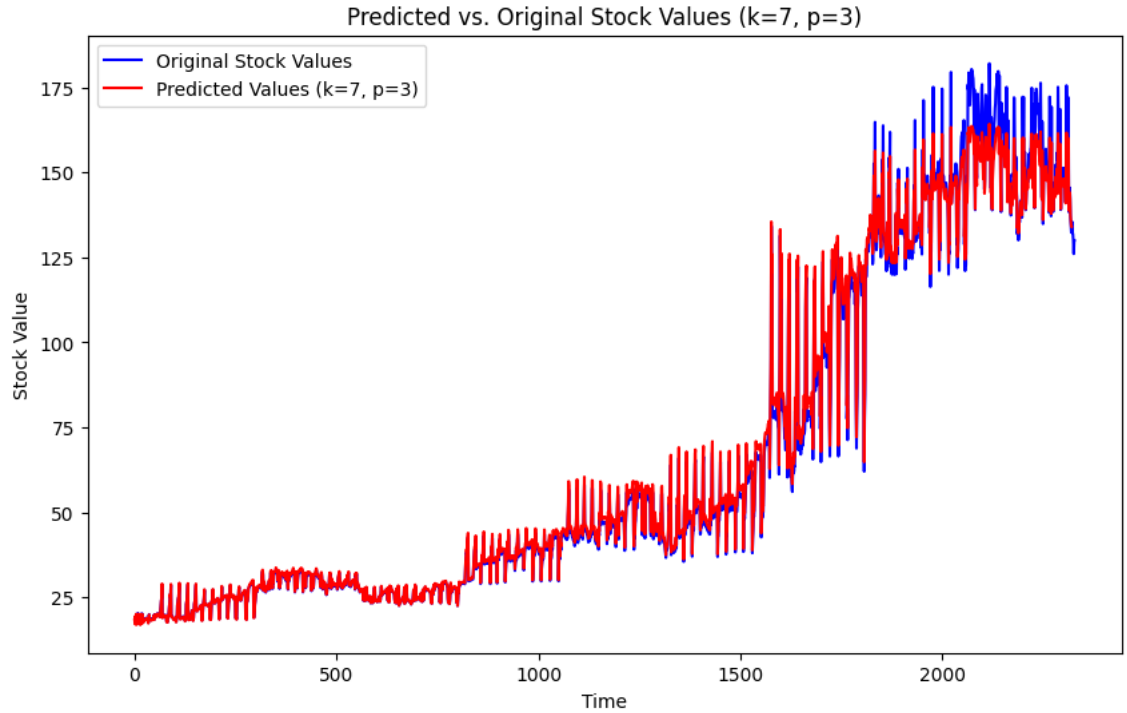




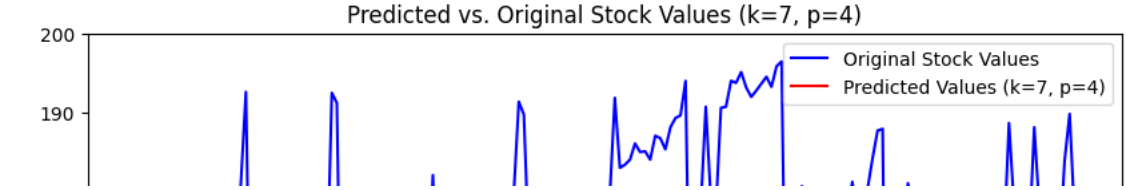


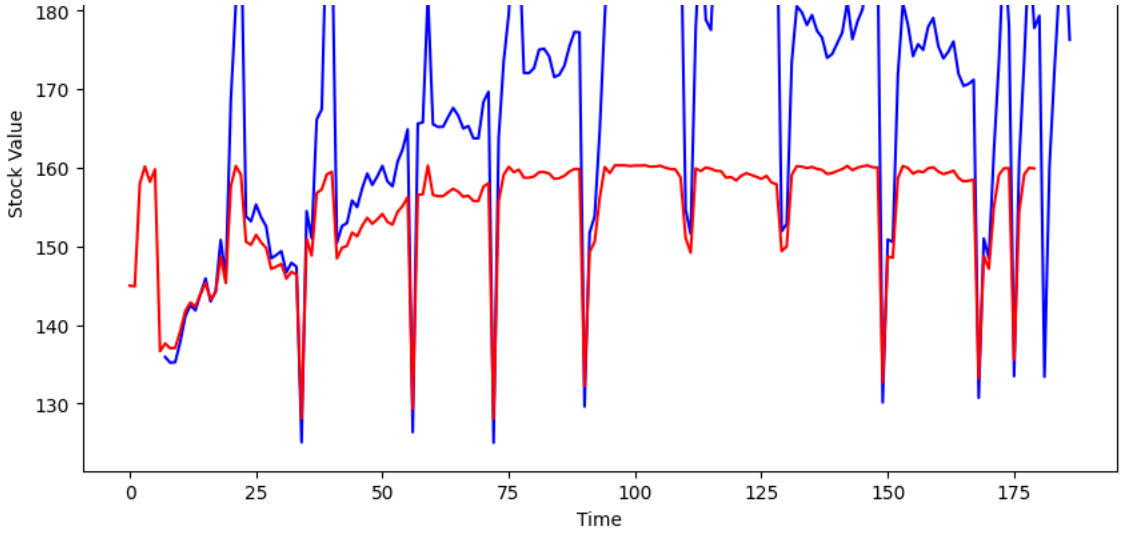

```
k = 7, p = 3
Training error: 147.08306452605268
Testing error: 424.16167464666813
```

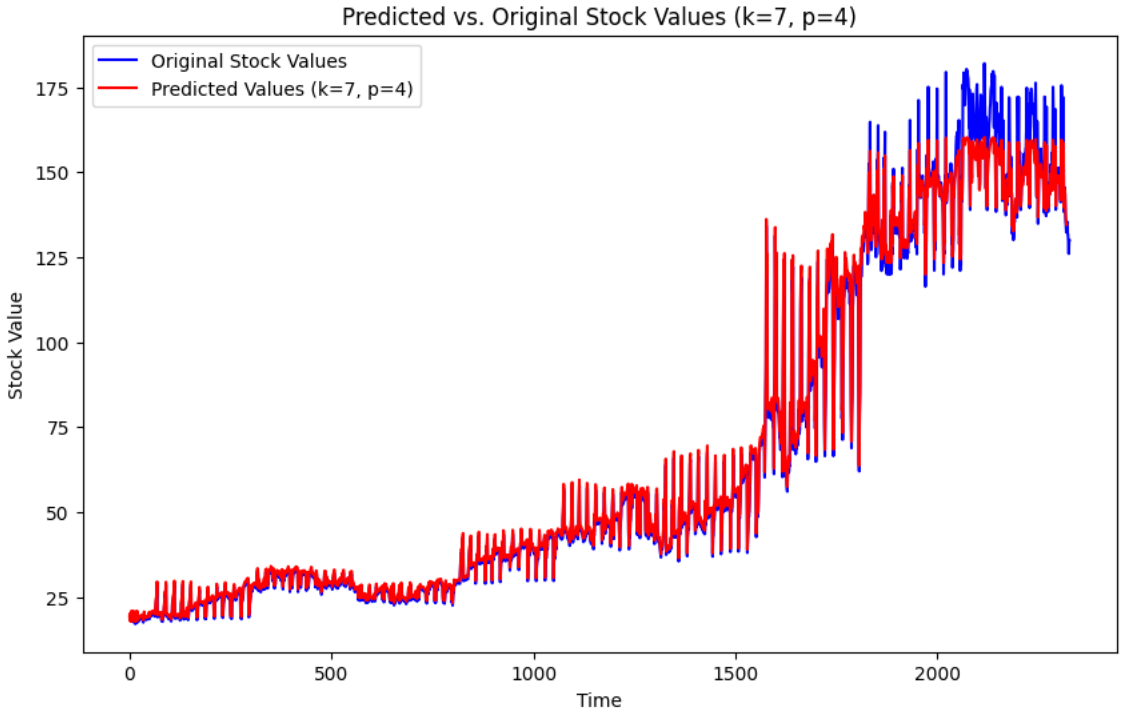




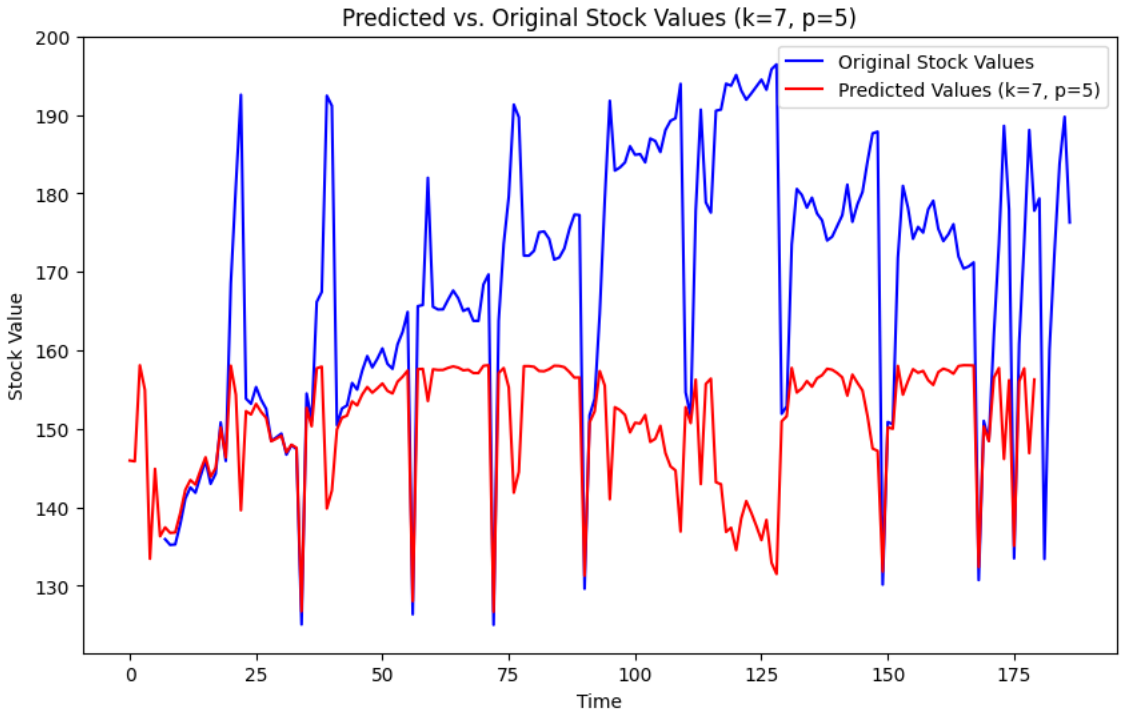

```
k = 7, p = 4
Training error: 146.51220375981543
Testing error: 474.3940527130634
```



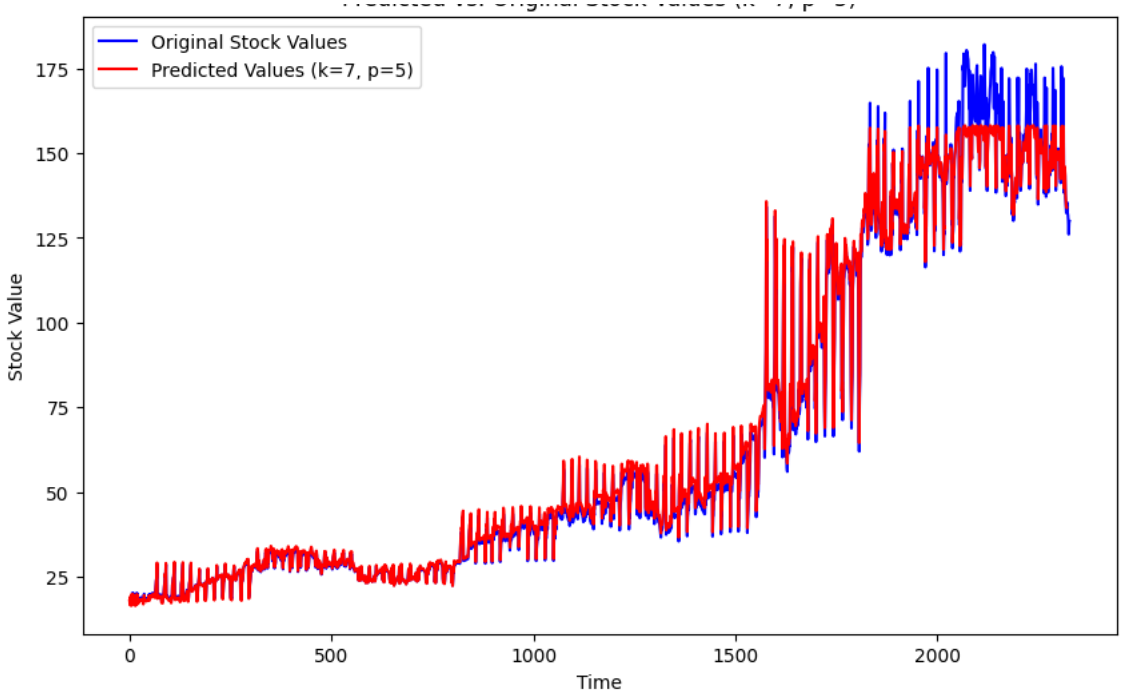




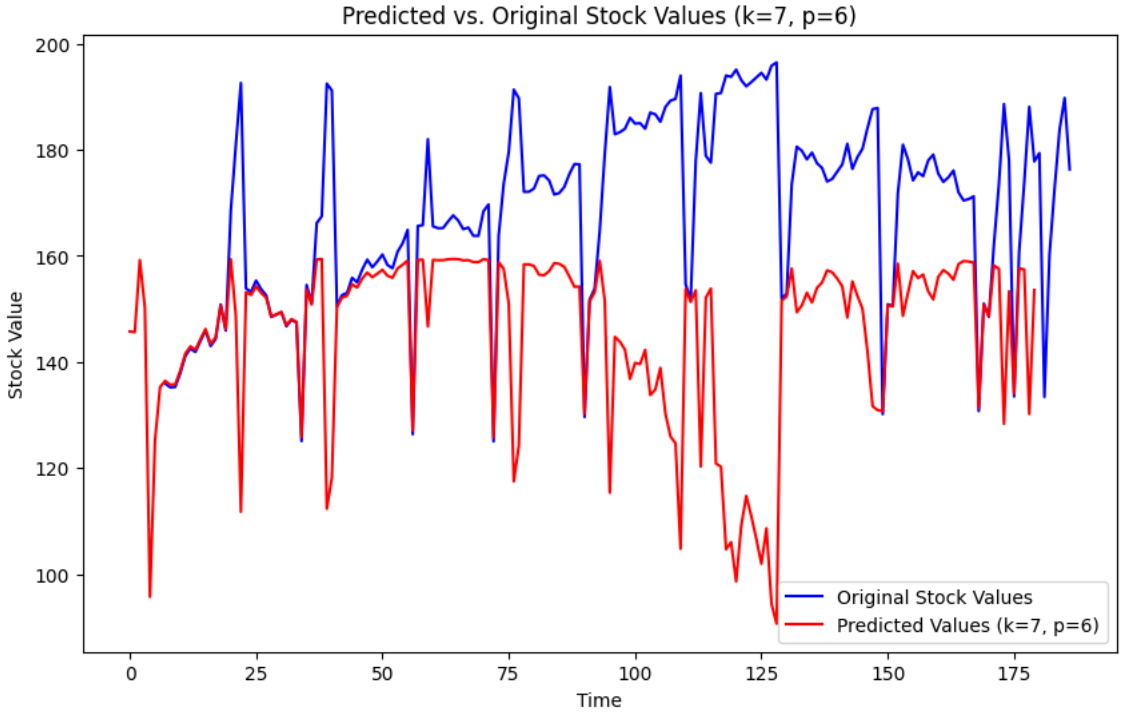

```
k = 7, p = 5
Training error: 145.54301588529881
Testing error: 679.1821123362824
```

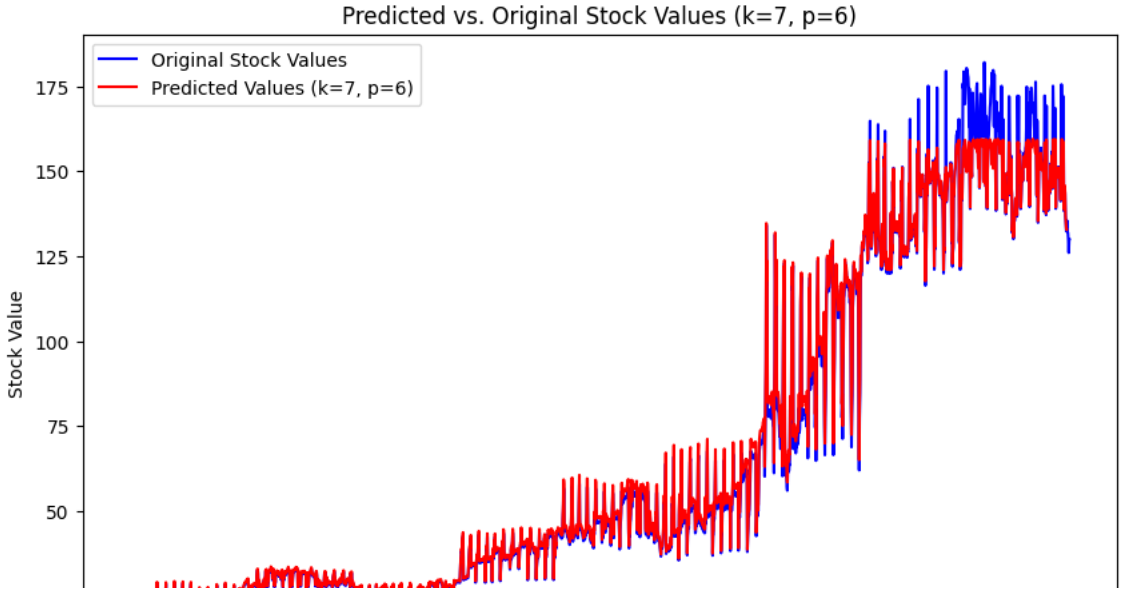


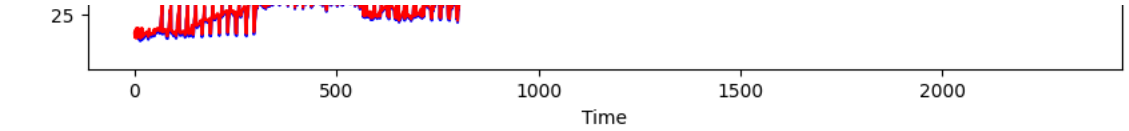
Predicted vs. Original Stock Values (k=7, p=5)



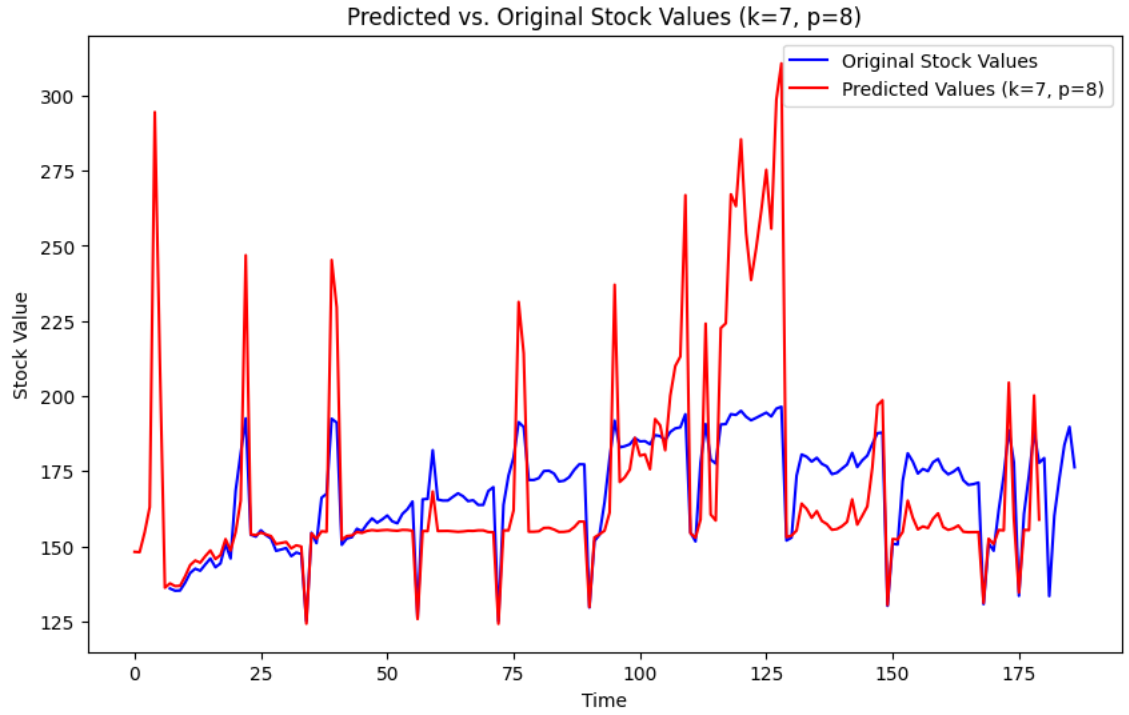

```
k = 7, p = 6
Training error: 144.82661952889802
Testing error: 1185.0629165045127
```

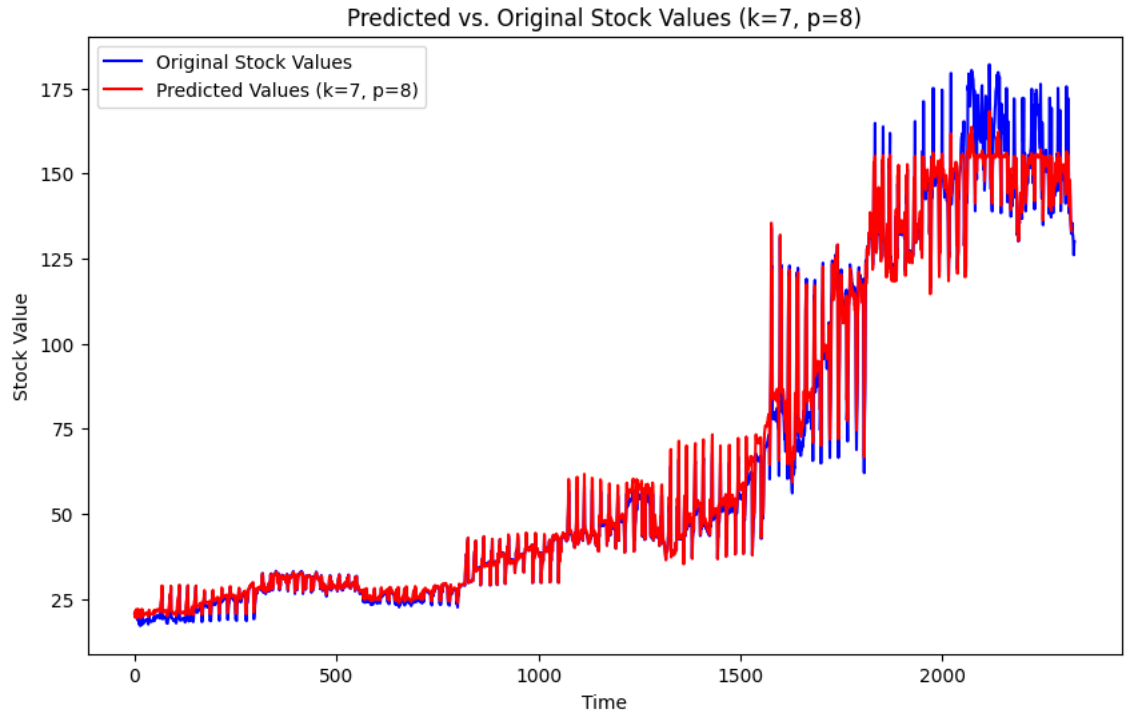




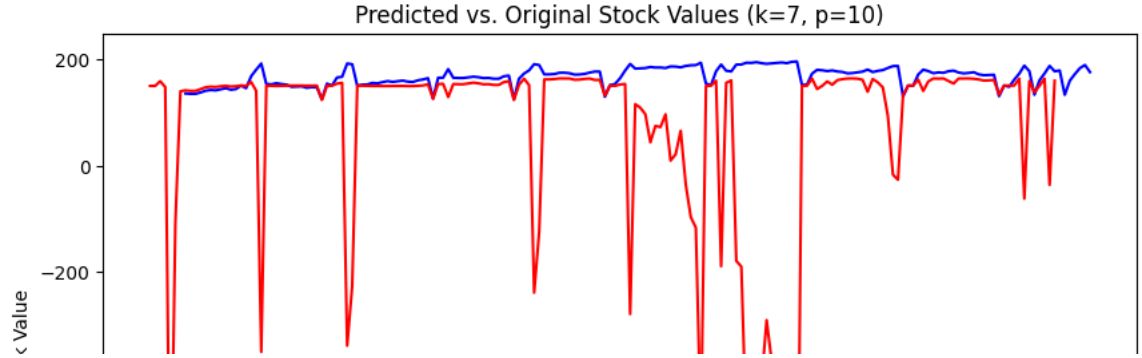


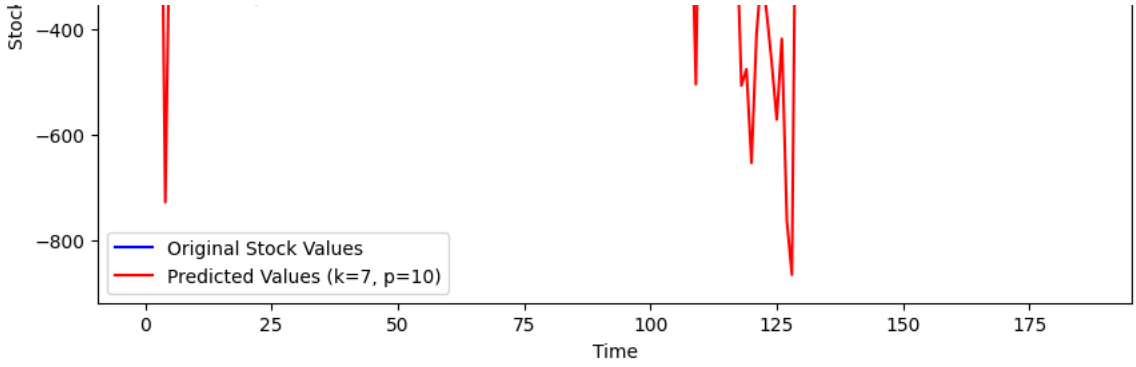

```
k = 7, p = 8
Training error: 142.28493841076704
Testing error: 1282.4525167145798
```

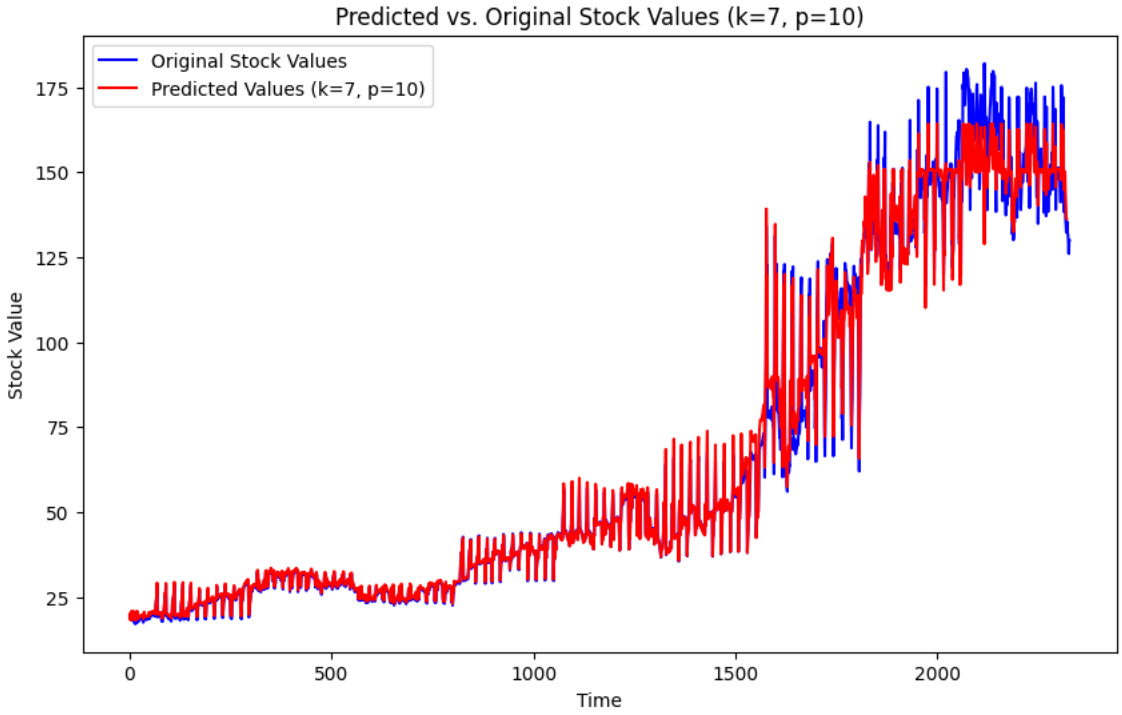




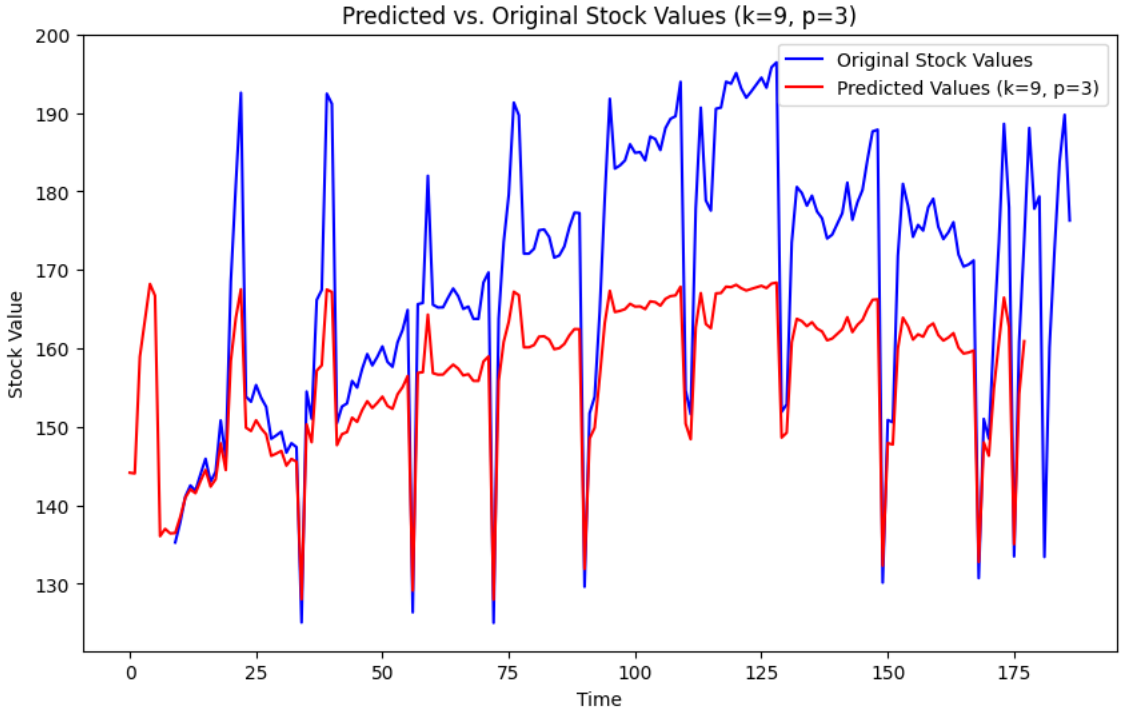

```
k = 7, p = 10
Training error: 137.0845242016022
Testing error: 51447.62352476746
```

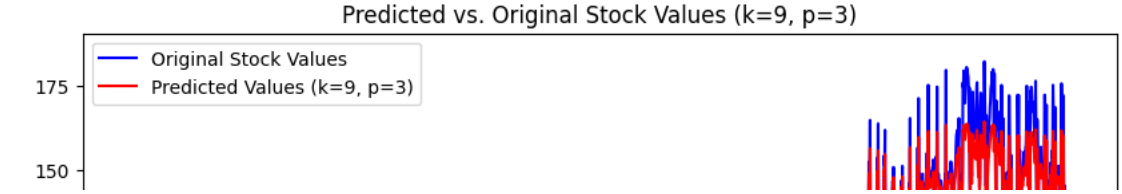


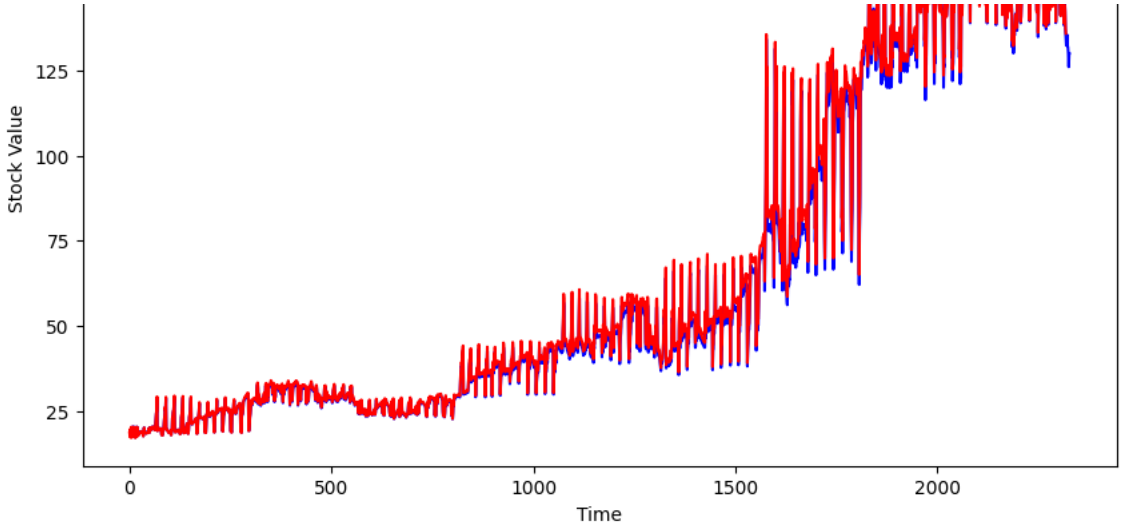




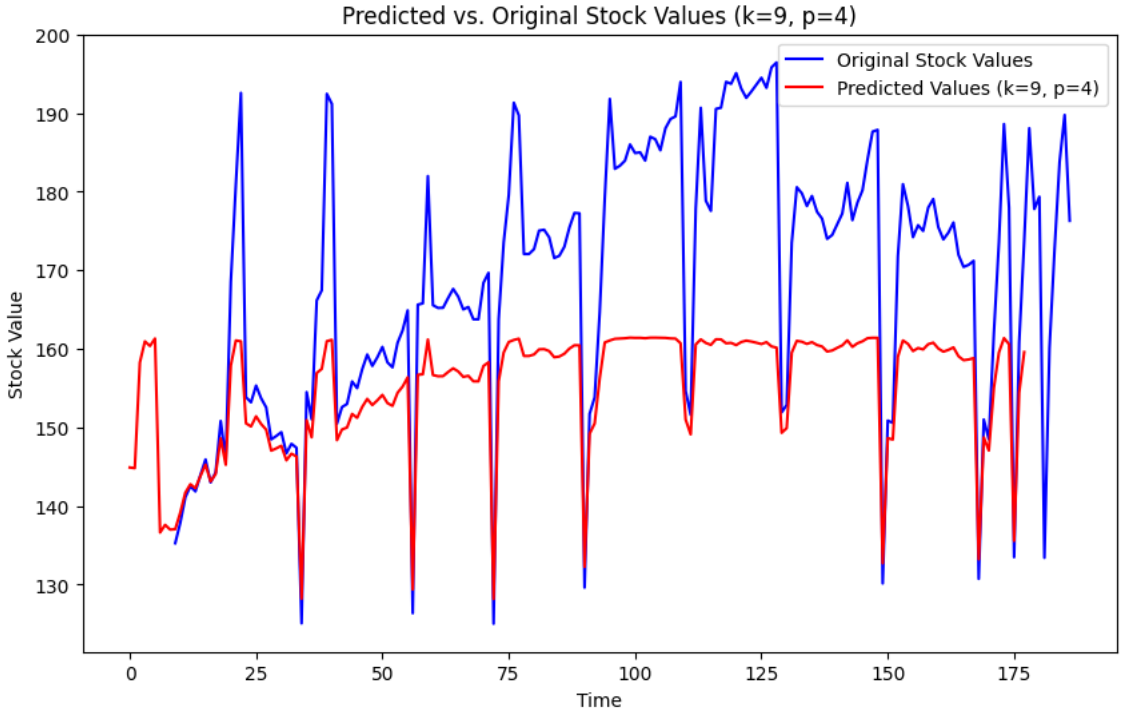

```
k = 9, p = 3
Training error: 147.51041657593643
Testing error: 462.86853007685636
```

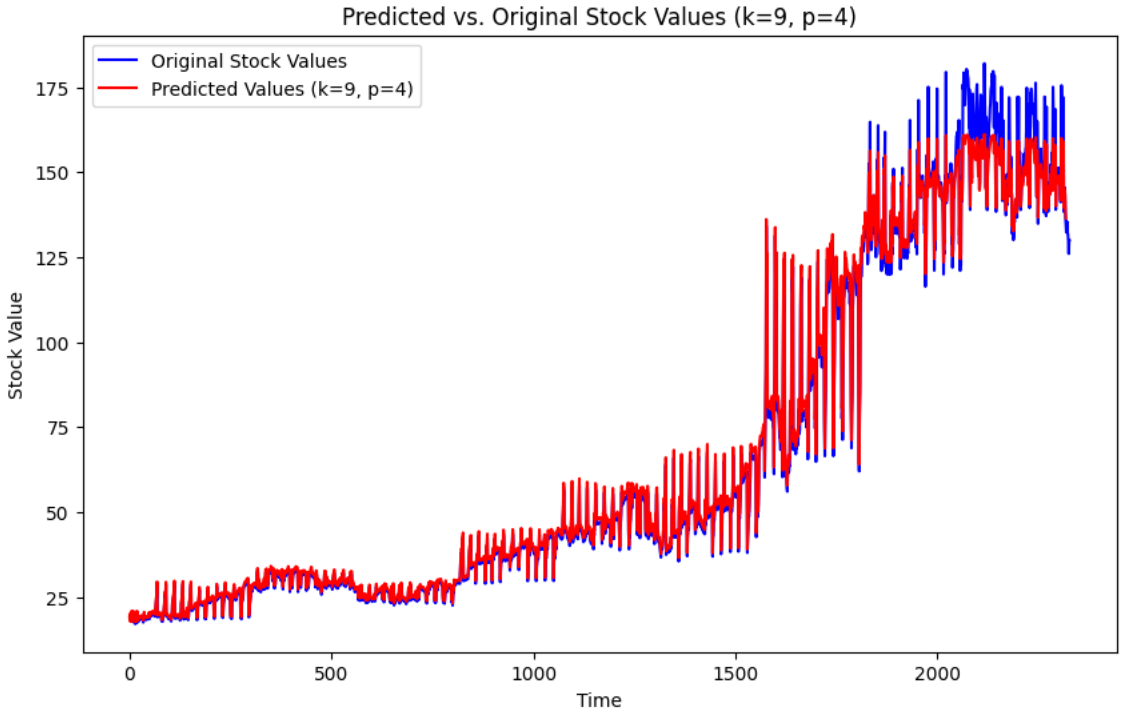






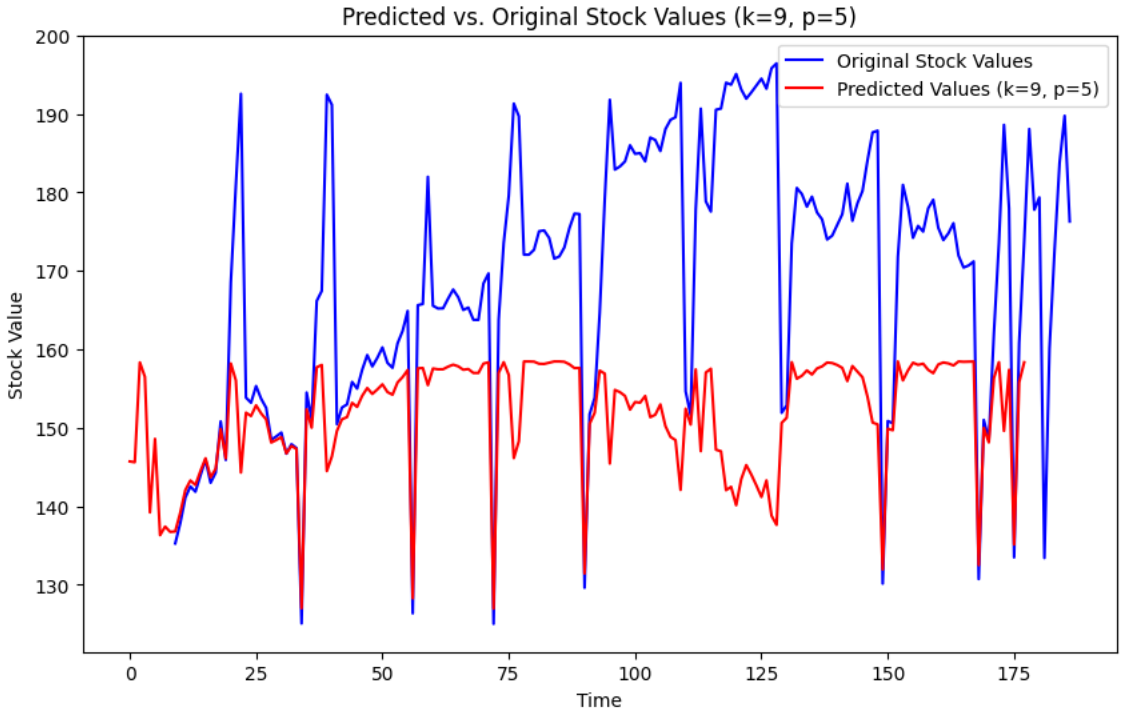

```
k = 9, p = 4
Training error: 147.15224721941155
Testing error: 494.31711262227753
```

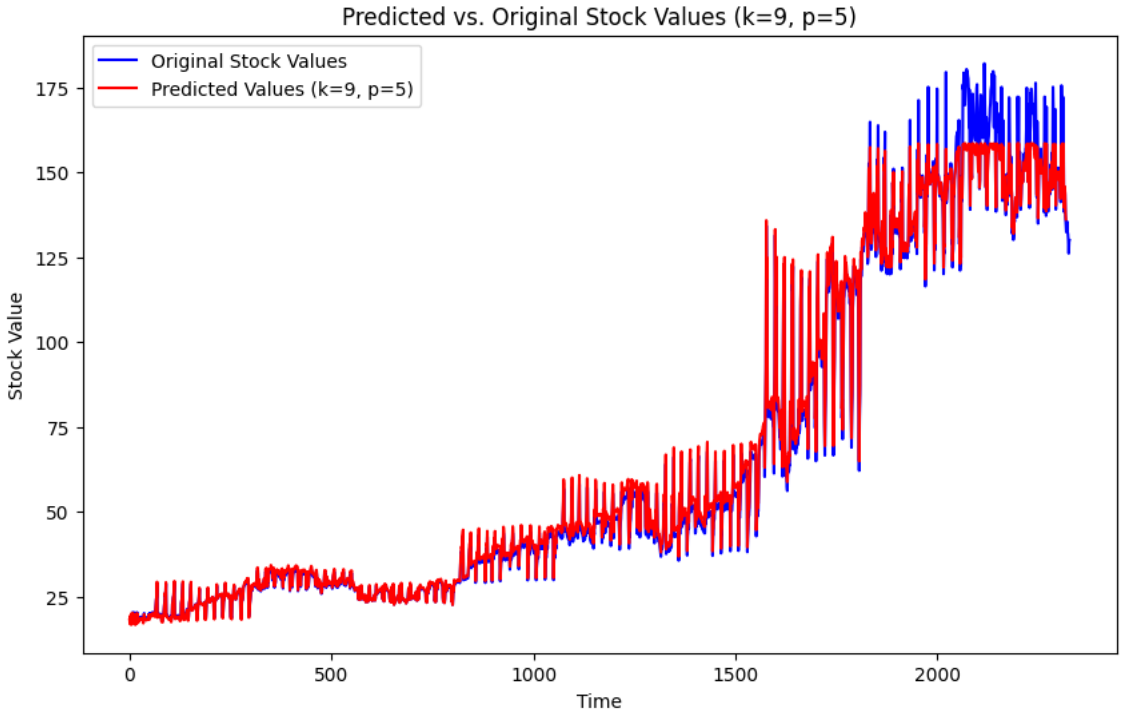




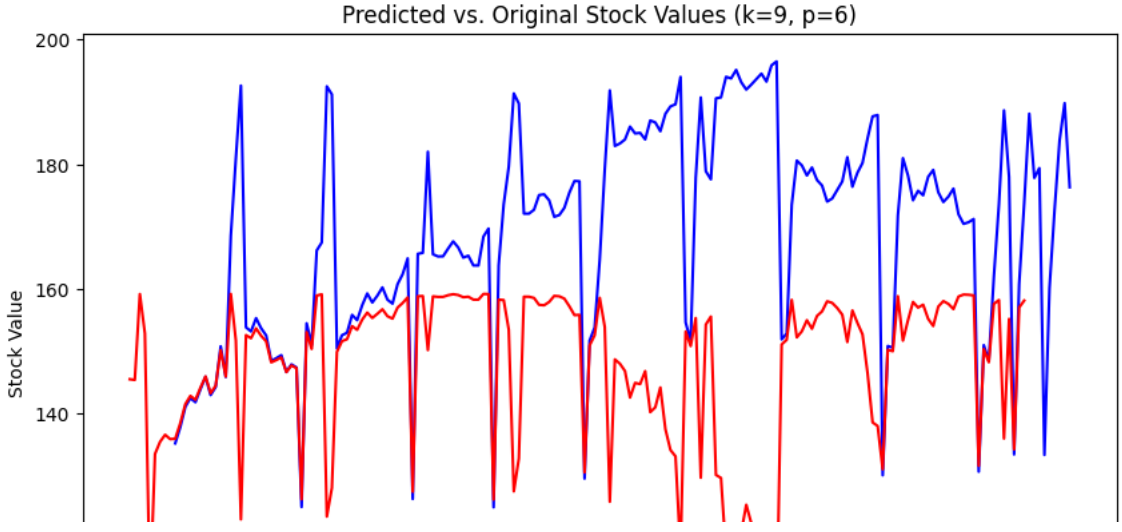

```
k = 9, p = 5
Training error: 146.4465457178094
```

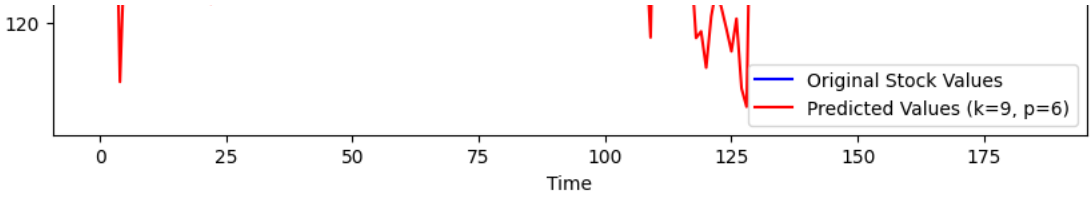
```
Training error: 140.44654371786094
Testing error: 637.2384949844766
```

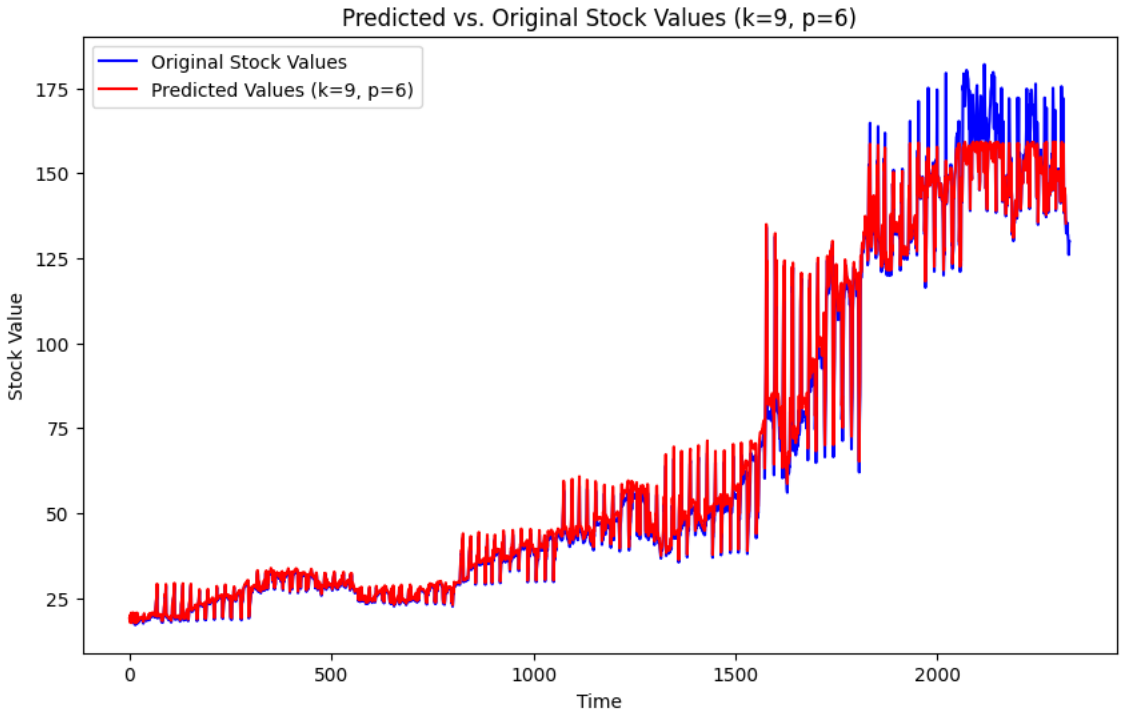





```
k = 9, p = 6
Training error: 146.0315229583377
Testing error: 949.9641165345181
```








```
k = 9, p = 8
Training error: 143.06111176411676
Testing error: 1764.893181985691
```