

# TSSL Lab 1 - Autoregressive models

We load a few packages that are useful for solving this lab assignment.

```
In [1]: import pandas as pd # Loading data / handling data frames
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model as lm # Used for solving Linear regression problems
from sklearn.neural_network import MLPRegressor # Used for NAR model

from tssltools_lab1 import acf, acfplot # Module available in LISAM - Used for plotting
```

## 1.1 Loading, plotting and detrending data

In this lab we will build autoregressive models for a data set corresponding to the Global Mean Sea Level (GMSL) over the past few decades. The data is taken from <https://climate.nasa.gov/vital-signs/sea-level/> and is available on LISAM in the file `sealevel.csv`.

**Q1:** Load the data and plot the GMSL versus time. How many observations are there in total in this data set?

*Hint:* With pandas you can use the function `pandas.read_csv` to read the csv file into a data frame. Plotting the time series can be done using `pyplot`. Note that the sea level data is stored in the 'GMSL' column and the time when each data point was recorded is stored in the column 'Year'.

**A1:** There are 1119 observations in this data set

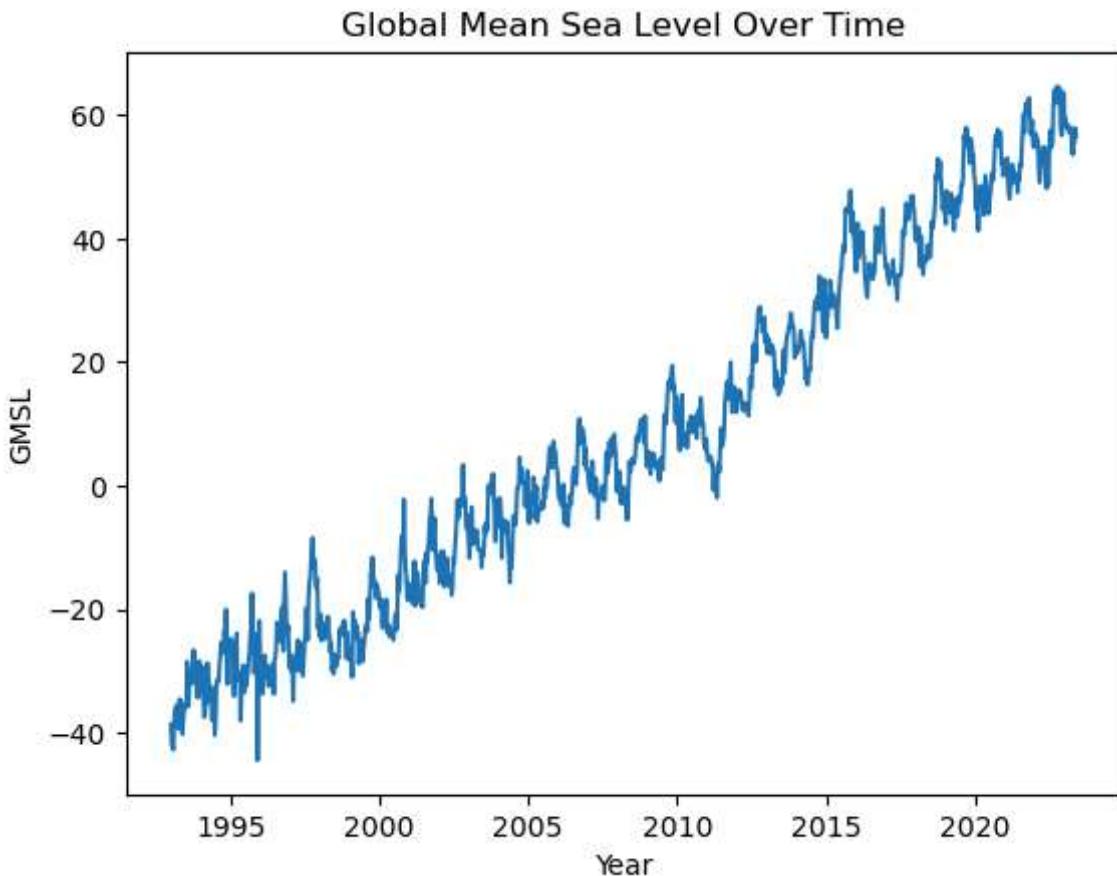
```
In [2]: data = pd.read_csv("sealevel.csv")

# Print number of observations
print("There are", np.shape(data)[0], "observations in this data set")

# Plot the GMSL vs time
plt.plot(data['Year'], data['GMSL'], label='GMSL')
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.title('Global Mean Sea Level Over Time')
```

There are 1119 observations in this data set

```
Out[2]: Text(0.5, 1.0, 'Global Mean Sea Level Over Time')
```



**Q2:** The data has a clear upward trend. Before fitting an AR model to this data need to remove this trend. Explain, using one or two sentences, why this is necessary.

**A2:** Before fitting an AR model to this data, we need to remove this trend because AR model assume that data is stationary which means properties of statical distributions(mean, median etc.) are not changing over time. If we don't remove this trend, AR model might be fail to represent the time-based(temporal) relationships and will cause to poor predictions.

**Q3** Detrend the data following these steps:

1. Fit a straight line,  $\mu_t = \theta_0 + \theta_1 u_t$  to the data based on the method of least squares.

Here,  $u_t$  is the time point when obervation  $t$  was recorded.

*Hint:* You can use `lm.LinearRegression().fit(...)` from scikit-learn. Note that the inputs need to be passed as a 2D array.

Before going on to the next step, plot your fitted line and the data in one figure.

2. Subtract the fitted line from  $y_t$  for the whole data series and plot the deviations from the straight line.

**From now, we will use the detrended data in all parts of the lab.**

*Note:* The GMSL data is recorded at regular time intervals, so that  $u_{t+1} - u_t = \text{const.}$  Therefore, you can just as well use  $t$  directly in the linear regression function if you prefer,  $\mu_t = \theta_0 + \theta_1 t$ .

### A3:

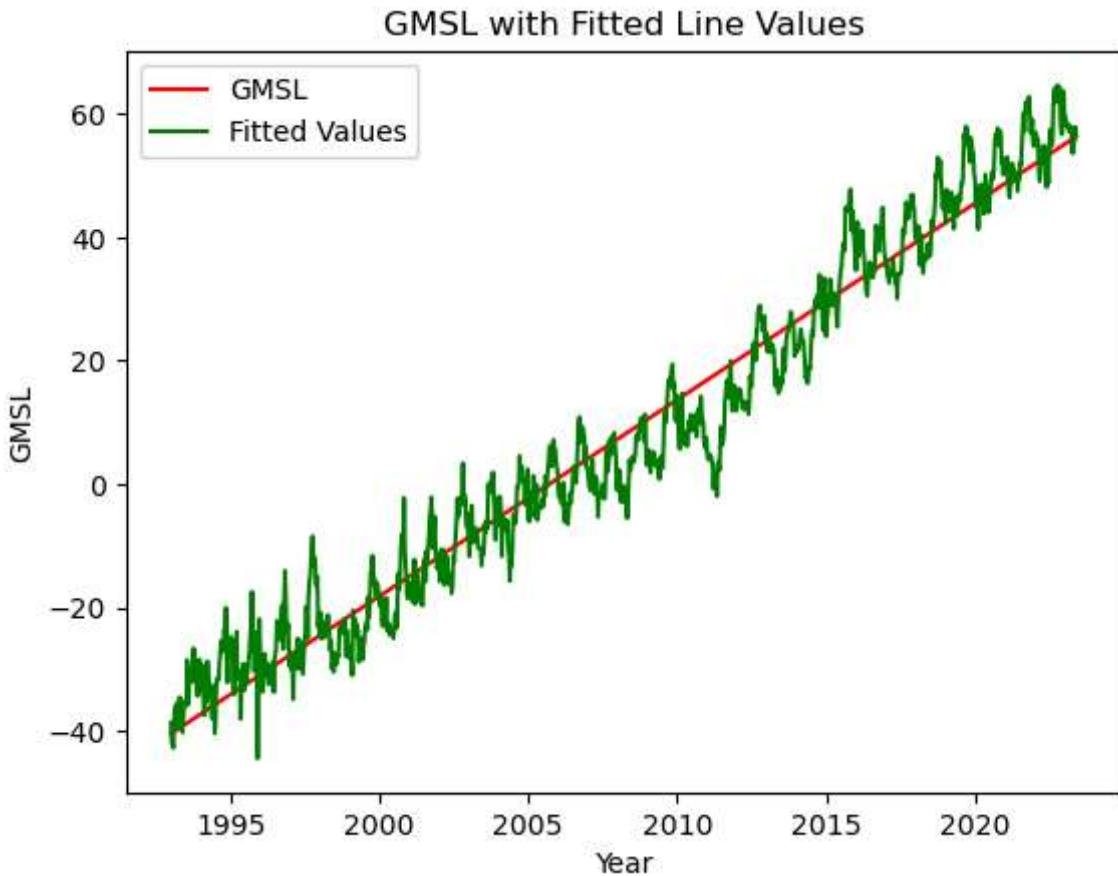
```
In [3]: # Prepare the data
X = np.asarray(data["Year"]).reshape(-1,1)
y = data["GMSL"]

# Create the model
model = lm.LinearRegression().fit(X,y)

# Prediction
model_pred = model.predict(X)

# Plot fitted Line and the data in one figure
plt.plot(data["Year"], model_pred, label='GMSL', color='red')
plt.plot(data["Year"], data["GMSL"], label='Fitted Values', color='green')
plt.legend()
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.title('GMSL with Fitted Line Values')
```

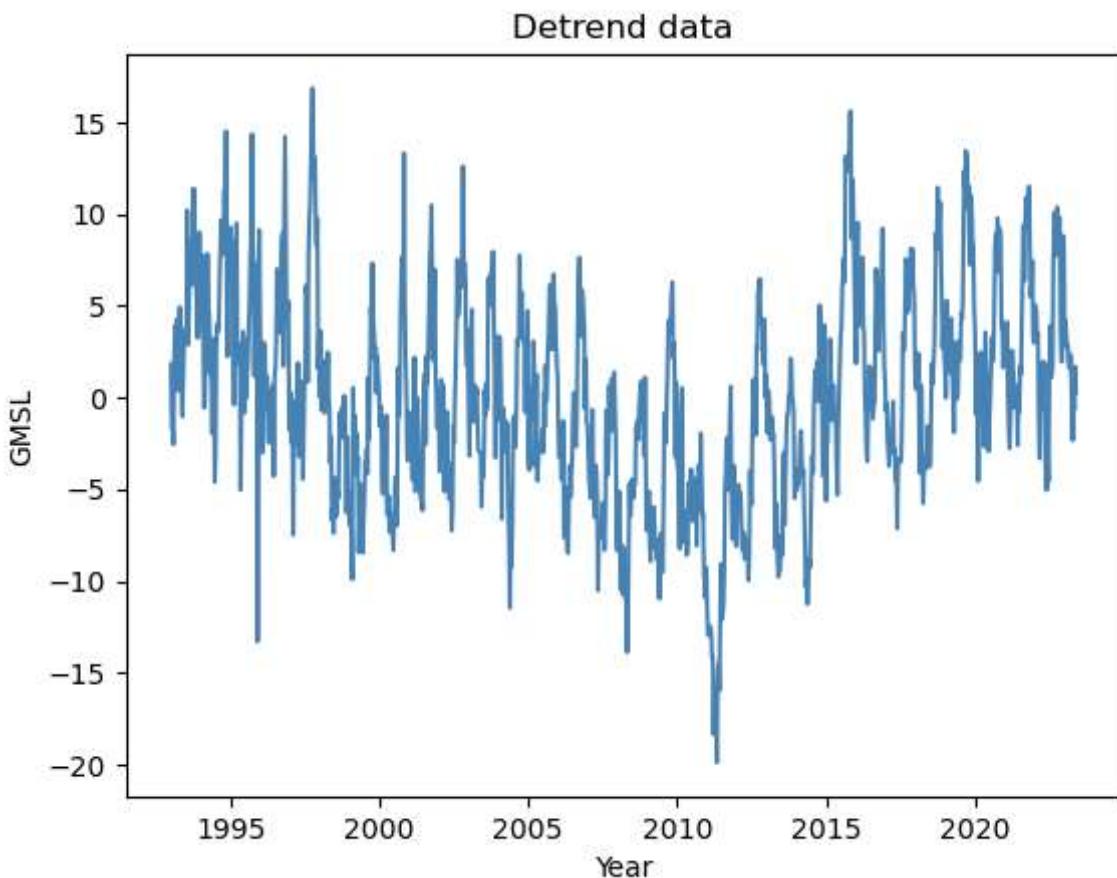
Out[3]: Text(0.5, 1.0, 'GMSL with Fitted Line Values')



```
In [4]: # Subtract the fitted line from y_t fro the whole data series
data2 = y - model_pred
# Plot the deviations from the straight line
plt.plot(data["Year"], data2, color='steelblue')

plt.xlabel('Year')
plt.ylabel('GMSL')
plt.title('Detrend data')
```

```
Out[4]: Text(0.5, 1.0, 'Detrend data')
```



**Q4:** Split the (detrended) time series into training and validation sets. Use the values from the beginning up to the 700th time point (i.e.  $y_t$  for  $t = 1$  to  $t = 700$ ) as your training data, and the rest of the values as your validation data. Plot the two data sets.

*Note:* In the above, we have allowed ourselves to use all the available data (train + validation) when detrending. An alternative would be to use only the training data also when detrending the model. The latter approach is more suitable if, either:

- we view the linear detrending as part of the model choice. Perhaps we wish to compare different polynomial trend models, and evaluate their performance on the validation data, or
- we wish to use the second chunk of observations to estimate the performance of the final model on unseen data (in that case it is often referred to as "test data" instead of "validation data"), in which case we should not use these observations when fitting the model, including the detrending step.

In this laboration we consider the linear detrending as a predetermined preprocessing step and therefore allow ourselves to use the validation data when computing the linear trend.

**A4:**

```
In [5]: # Split the data into train and val
X_train = X[:700]
y_train = data2[:700]
```

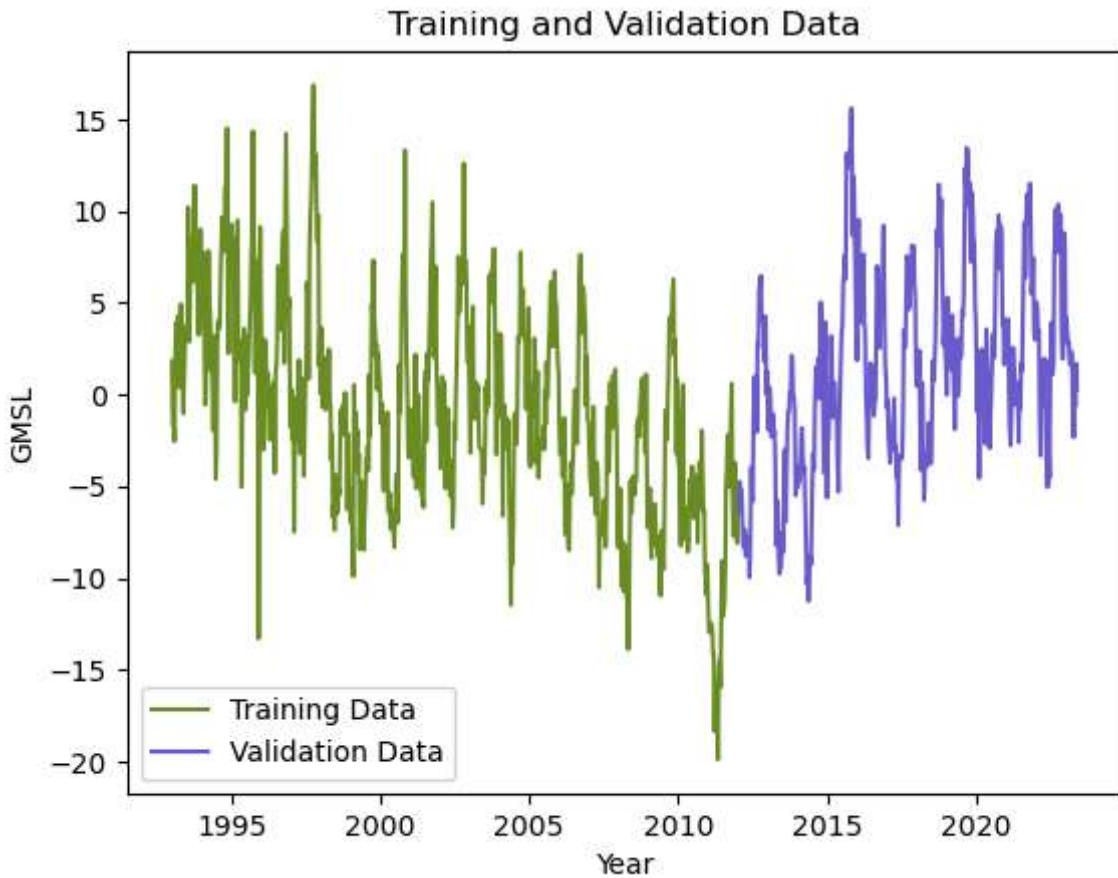
```

X_valid = X[700:]
y_valid = data2[700:]

# Plot train and val datas
plt.plot(X_train, y_train, label='Training Data', color='olivedrab')
plt.plot(X_valid, y_valid, label='Validation Data', color='slateblue')
plt.legend()
plt.xlabel('Year')
plt.ylabel('GMSL')
plt.title('Training and Validation Data')

```

Out[5]: Text(0.5, 1.0, 'Training and Validation Data')



## 1.2 Fit an autoregressive model

We will now fit an  $\text{AR}(p)$  model to the training data for a given value of the model order  $p$ .

**Q5:** Create a function that fits an  $\text{AR}(p)$  model for an arbitrary value of  $p$ . Use this function to fit a model of order  $p = 10$  to the training data and write out (or plot) the coefficients.

*Hint:* Since fitting an AR model is essentially just a standard linear regression we can make use of `lm.LinearRegression().fit(...)` similarly to above. You may use the template below and simply fill in the missing code.

**A5:**

```
In [6]: def fit_ar(y, p):
    """Fits an AR(p) model. The loss function is the sum of squared errors from
    :param y: array (n,), training data points
    :param p: int, AR model order
    :return theta: array (p,), learnt AR coefficients
    """

    # Number of training data points
    n = len(y) # <COMPLETE THIS LINE>

    # Construct the regression matrix
    Phi = np.zeros((n-p, p)) # <COMPLETE THIS LINE>
    for j in range(p):
        Phi[:,j] = y[p - j - 1 : n - j - 1] # <COMPLETE THIS LINE>

    # Drop the first p values from the target vector y
    yy = y[p:] # yy = (y_{t+p+1}, ..., y_n)

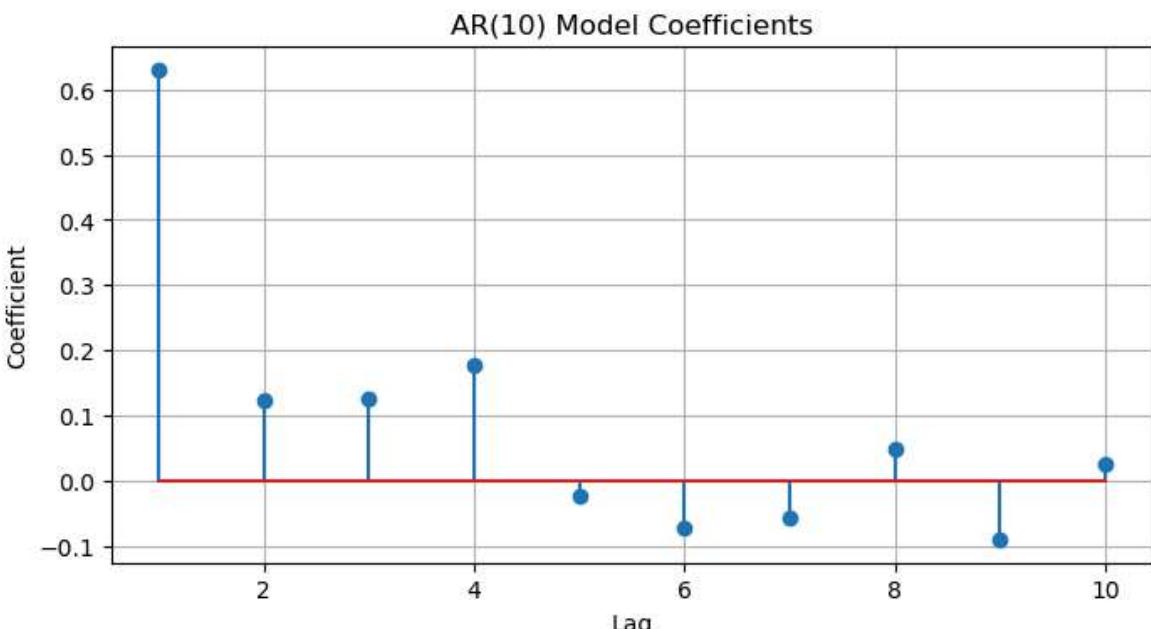
    # Here we use fit_intercept=False since we do not want to include an intercept
    regr = lm.LinearRegression(fit_intercept=False)
    regr.fit(Phi,yy)

    return regr.coef_
```

```
In [7]: p = 10
theta = fit_ar(y_train, p)
print(theta)
```

```
# Plot the AR coefficients
plt.figure(figsize=(8, 4))
plt.stem(np.arange(1, p+1), theta)
plt.grid(True)
plt.xlabel('Lag')
plt.ylabel('Coefficient')
plt.title(f'AR({p}) Model Coefficients')
plt.show()
```

```
[ 0.63068183  0.1231388   0.12558768  0.17683292 -0.02284342 -0.07140349
 -0.05693816  0.0479181  -0.0893176   0.0251526 ]
```



**Q6:** Next, write a function that computes the one-step-ahead prediction of your fitted model. 'One-step-ahead' here means that in order to predict  $y_t$  at  $t = t_0$ , we use the actual values of  $y_t$  for  $t < t_0$  from the data. Use your function to compute the predictions for both *training data* and *validation data*. Plot the predictions together with the data (you can plot both training and validation data in the same figure). Also plot the *residuals*.

*Hint:* It is enough to call the predict function once, for both training and validation data at the same time.

### A6:

```
In [8]: def predict_ar_1step(theta, y_target):
    """Predicts the value y_t for t = p+1, ..., n, for an AR(p) model, based on
    one-step-ahead prediction.

    :param theta: array (p,), AR coefficients, theta=(a1,a2,...,ap).
    :param y_target: array (n,), the data points used to compute the predictions
    :return y_pred: array (n-p,), the one-step predictions (\hat y_{p+1}, ....,
    """

    n = len(y_target)
    p = len(theta)

    # Number of steps in prediction
    m = n-p
    y_pred = np.zeros(m)

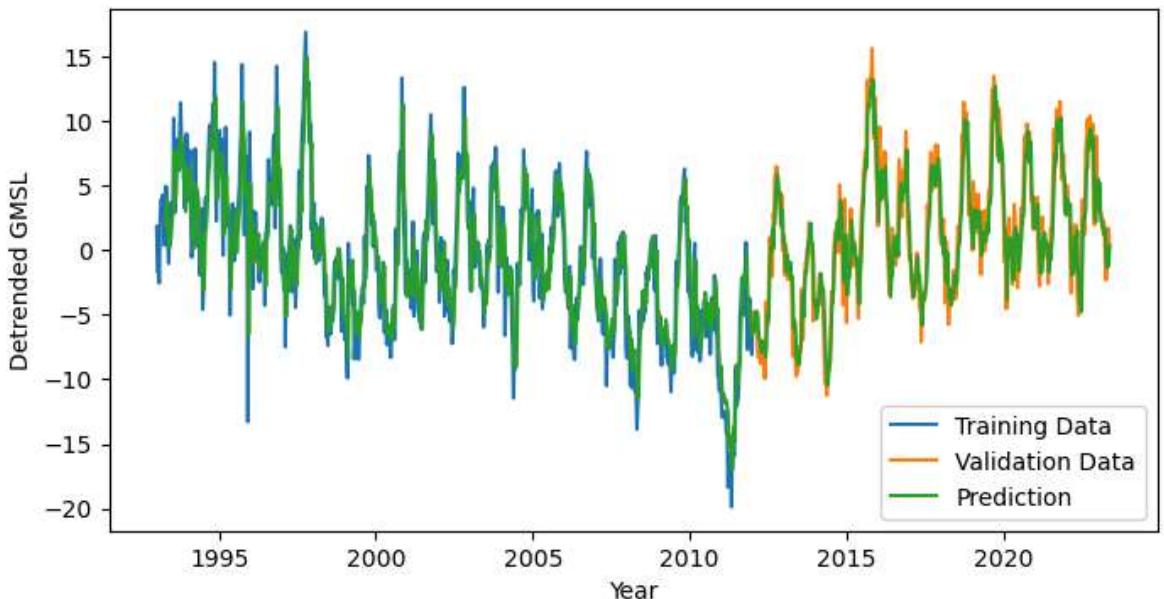
    for i in range(m):
        # <COMPLETE THIS CODE BLOCK>
        y_pred[i] = np.dot(theta, y_target[i + p - np.arange(1, p + 1)]) # <COMP

    return y_pred
```

<>>2: SyntaxWarning: invalid escape sequence '\h'  
<>>2: SyntaxWarning: invalid escape sequence '\h'  
C:\Users\SK\AppData\Local\Temp\ipykernel\_18288\2208616143.py:2: SyntaxWarning: invalid escape sequence '\h'  
 """Predicts the value y\_t for t = p+1, ..., n, for an AR(p) model, based on the  
 data in y\_target using

```
In [9]: pred_ar = predict_ar_1step(theta, data2)

plt.figure(figsize=(8, 4))
plt.plot(X_train, y_train, label = "Training Data")
plt.plot(X_valid, y_valid, label = "Validation Data")
plt.plot(X[p:], pred_ar, label = "Prediction" )
plt.xlabel("Year")
plt.ylabel("Detrended GMSL")
plt.legend()
plt.show()
```



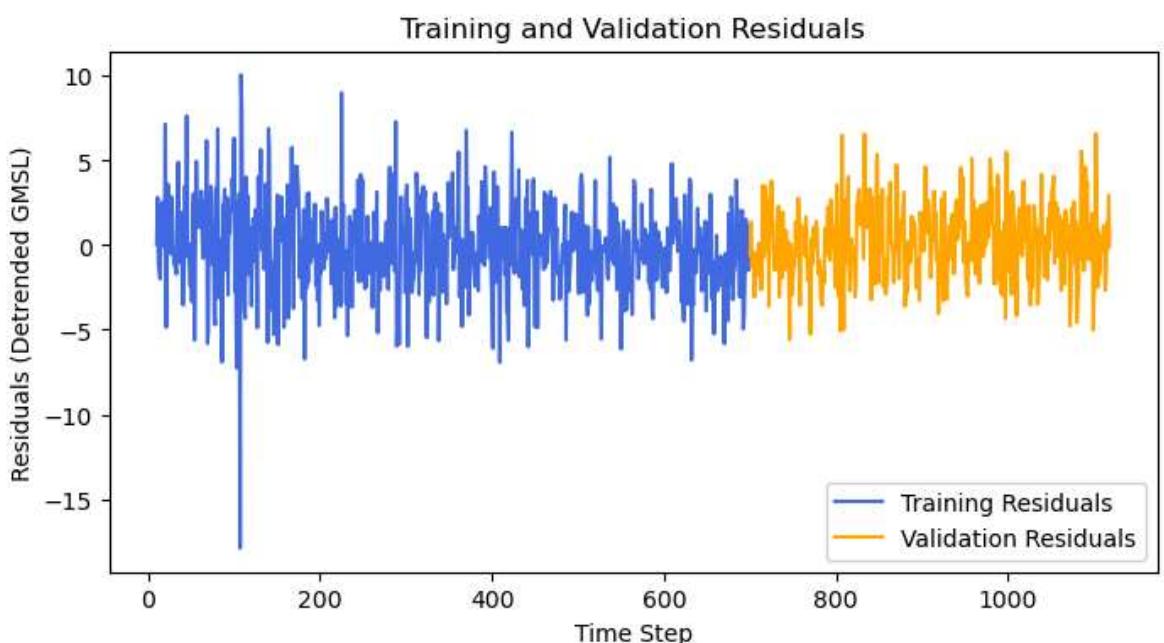
```
In [10]: # Calculate residuals
residuals = data2[p:] - pred_ar

train_residuals = residuals[:700-p] # Training residuals, from t=p+1 to t=700
val_residuals = residuals[700-p:] # Validation residuals

# Plot residuals
plt.figure(figsize=(8, 4))
plt.plot(np.arange(p, 700), train_residuals, label="Training Residuals", color='blue')
plt.plot(np.arange(700, len(data2)), val_residuals, label="Validation Residuals", color='orange')

# Add Labels, title, and Legend
plt.xlabel("Time Step")
plt.ylabel("Residuals (Detrended GMSL)")
plt.title("Training and Validation Residuals")
plt.legend()

plt.show()
```



**Q7:** Compute and plot the autocorrelation function (ACF) of the *residuals* only for the *validation data*. What conclusions can you draw from the ACF plot?

*Hint:* You can use the function `acfplot` from the `tssltools` module, available on the course web page.

### A7:

```
In [11]: help(acfplot)
```

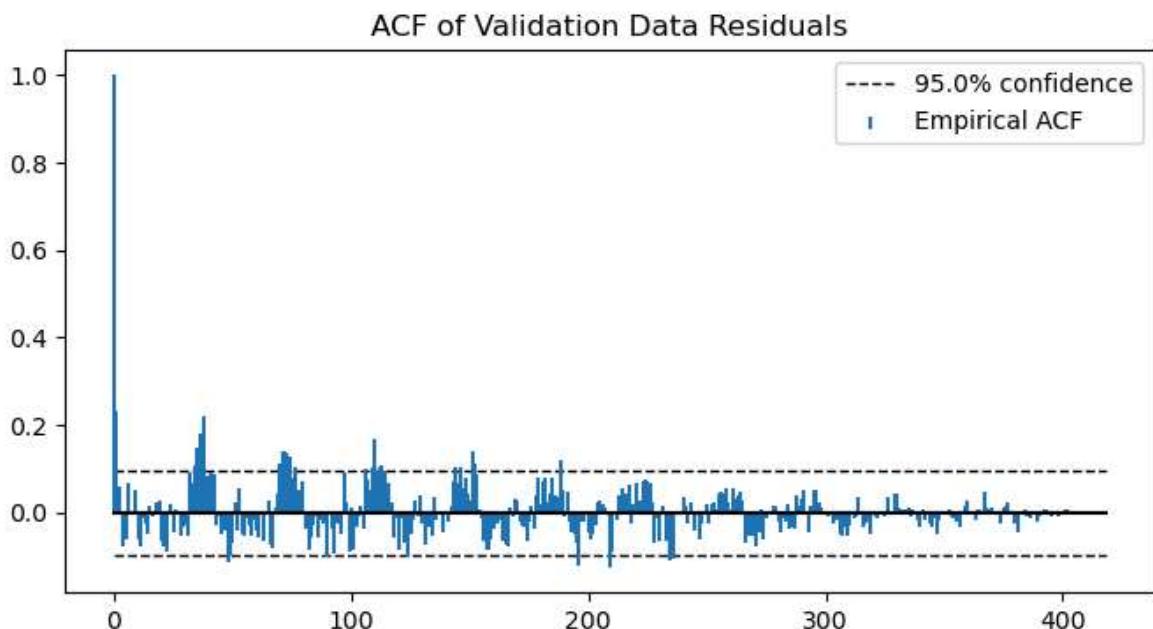
Help on function `acfplot` in module `tssltools_lab1`:

```
acfplot(x, lags=None, conf=0.95)
    Plots the empirical autocorrelation function.

    :param x: array (n,), sequence of data points
    :param lags: int, maximum lag to compute the ACF for. If None, this is set to
n-1. Default is None.
    :param conf: float, number in the interval [0,1] which specifies the confidence
level (based on a central limit
        theorem under a white noise assumption) for two dashed lines drawn in the plot. Default is 0.95.
    :return:
```

```
In [12]: from tssltools_lab1 import acfplot
```

```
# Plot the ACF of the residuals for the validation data
plt.figure(figsize=(8, 4))
acfplot(val_residuals)
plt.title("ACF of Validation Data Residuals")
plt.show()
```



According to the ACF plot of validation data, we observed that AR(10) model doesn't fully fit the data. There is a noticeable deviation in values of the left side of the graph and many of them outside of the 95% confidence interval. For good model, the residuals

should resemble white noise, with ACF values close to 0, indicating that no remaining autocorrelation. This means that AR(10) is insufficient for this dataset.

## 1.3 Model validation and order selection

Above we set the model order  $p = 10$  quite arbitrarily. In this section we will try to find an appropriate order by validation.

**Q8:** Write a loop in which AR-models of orders from  $p = 2$  to  $p = 150$  are fitted to the data above. Plot the training and validation mean-squared errors for the one-step-ahead predictions versus the model order.

Based on your results:

- What is the main difference between the changes in training error and validation error as the order increases?
- Based on these results, which model order would you suggest to use and why?

*Note:* There is no obvious "correct answer" to the second question, but you still need to pick an order and motivate your choice!

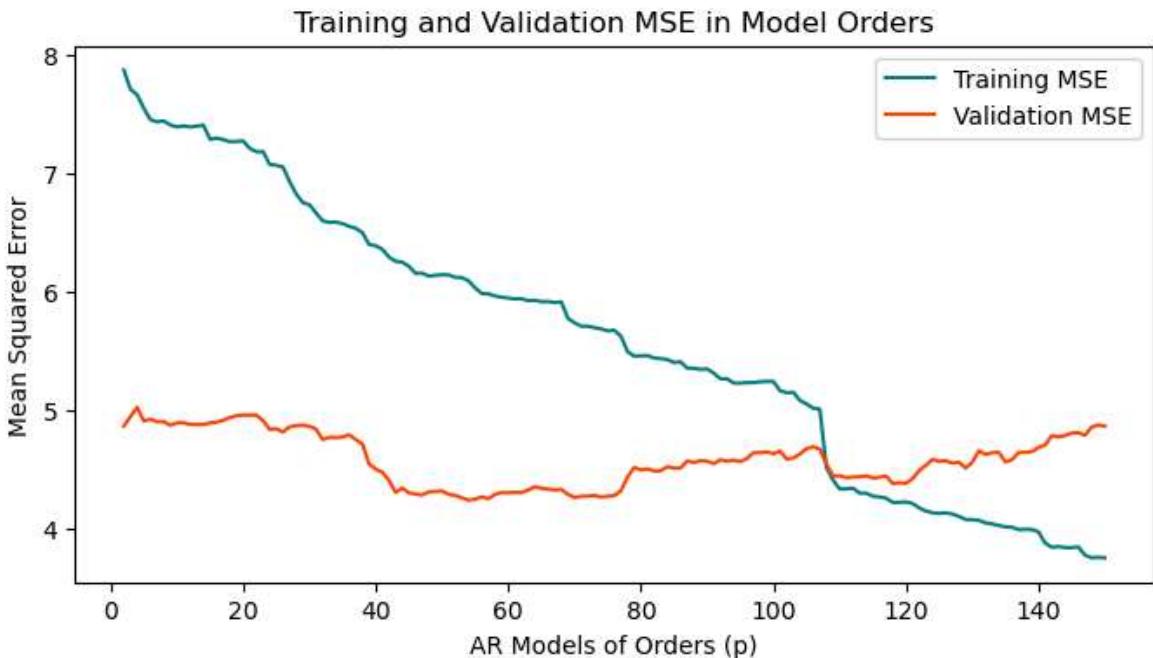
**A8:**

```
In [13]: # Initialize arrays for storing MSE
train_MSE = np.zeros(149)
valid_MSE = np.zeros(149)

# Loop over model orders from 2 to 150
for i in range(2,151):
    theta = fit_ar(y_train, i)
    pred_ar = predict_ar_1step(theta, data2)

    # Compute training and validation MSE
    loop = np.squeeze(data2[i:]) - pred_ar
    valid_MSE[i-2] = np.sum((loop[701-i:]**2)) / len(loop[701-i:])
    train_MSE[i-2] = np.sum((loop[:701-i]**2)) / len(loop[:701-i])

# Plot the Training and Validation MSE in Model Orders
plt.figure(figsize=(8, 4))
plt.plot(range(2,151), train_MSE, label = "Training MSE", color='teal')
plt.plot(range(2,151), valid_MSE, label = "Validation MSE", color='orange')
plt.xlabel('AR Models of Orders (p)')
plt.ylabel('Mean Squared Error')
plt.title('Training and Validation MSE in Model Orders')
plt.legend()
plt.show()
```



```
In [14]: # Find the index of the minimum validation MSE
best_order = np.argmin(train_MSE) + 2 # Add 2 because the array starts at order 1
min_mse = np.min(train_MSE)

best_order2 = np.argmin(valid_MSE) + 2
min_mse2 = np.min(valid_MSE)

# Print the best order
print(f'The best model order is: {best_order} with a training MSE of: {min_mse}')
print(f'The best model order is: {best_order2} with a validation MSE of: {min_mse2}'
```

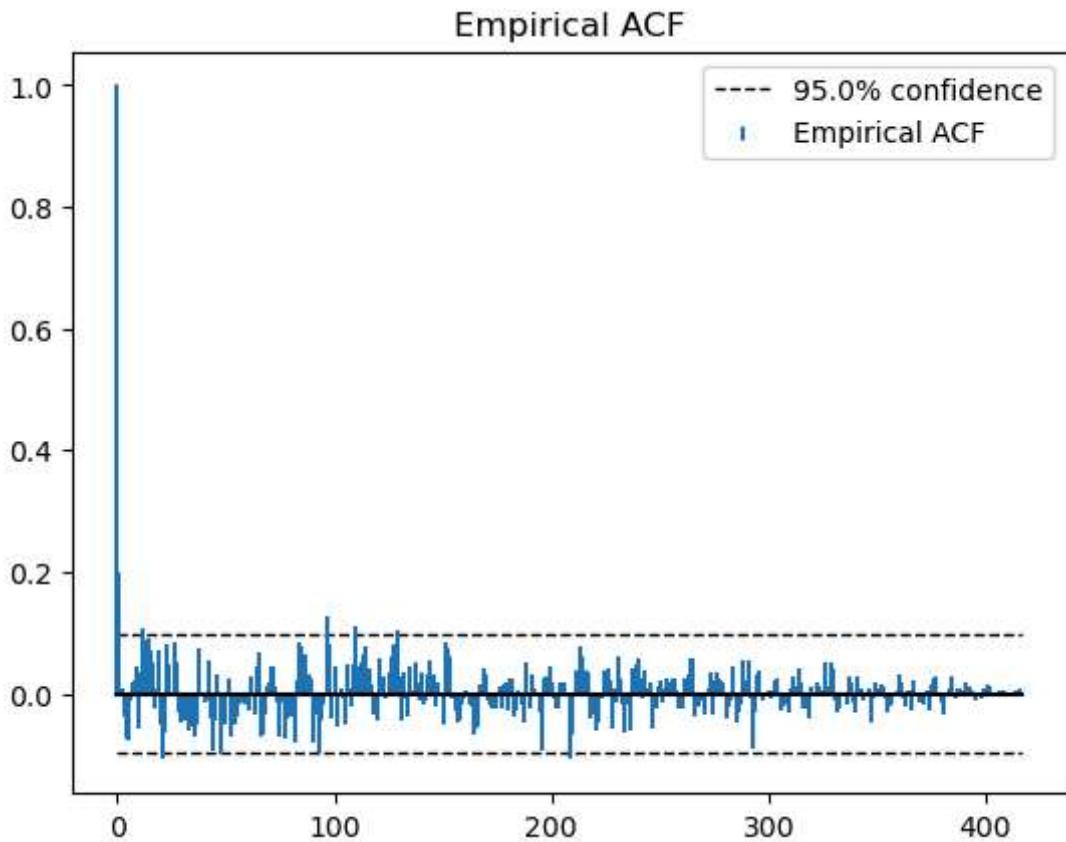
The best model order is: 150 with a training MSE of: 3.7531645210158633  
The best model order is: 54 with a validation MSE of: 4.239951178217028

When order increase, Training MSE is decreasing while Validation MSE is increasing.

According to graph and also check with code, best model is AR(54) because it has a lowest error.

**Q9:** Based on the chosen model order, compute the residuals of the one-step-ahead predictions on the *validation data*. Plot the autocorrelation function of the residuals. What conclusions can you draw? Compare to the ACF plot generated above for p=10.

```
In [15]: p_new = 54 # Best model order
theta_new = fit_ar(y_train, p_new)
pred_ar_new = predict_ar_1step(theta_new, data2)
loop_new = np.squeeze(data2[p_new:]) - pred_ar_new
val_residuals_new = loop_new[701-p_new:]
acfplot(val_residuals_new)
```



There is a great improvement in the AR(54) model when compare to the AR(10) model. As seen, most of the values are inside of the 95% confidence interval and also it converges more quickly. It is not best model but it is better than AR(10).

## 1.4 Long-range predictions

So far we have only considered one-step-ahead predictions. However, in many practical applications it is of interest to use the model to predict further into the future. For instance, for the sea level data studied in this laboration, it is more interesting to predict the level one year from now, and not just 10 days ahead (10 days = 1 time step in this data).

**Q10:** Write a function that simulates the value of an AR( $p$ ) model  $m$  steps into the future, conditionally on an initial sequence of data points. Specifically, given  $y_{1:n}$  with  $n \geq p$  the function/code should predict the values

$$\hat{y}_{t|n} = \mathbb{E}[y_t | y_{1:n}], \quad t = n + 1, \dots, n + m. \quad (1)$$

Use this to predict the values for the validation data ( $y_{701:997}$ ) conditionally on the training data ( $y_{1:700}$ ) and plot the result.

*Hint:* Use the pseudo-code derived at the first pen-and-paper session.

**A10:**

```
In [16]: def simulate_ar(y, theta, m):
    """Simulates an AR(p) model for m steps, with initial condition given by the
```

```

:param y: array (n,) with n>=p. The last p values are used to initialize the
:param theta: array (p,). AR model parameters,
:param m: int, number of time steps to simulate the model for.
"""

p = len(theta)
y_sim = np.zeros(m)
phi = np.flip(y[-p:].copy()) # (y_{n-1}, ..., y_{n-p})^T - note that y[ntrain]

for i in range(m):
    y_sim[i] = np.dot(theta, phi) # <COMPLETE THIS LINE>
    # <COMPLETE THIS CODE BLOCK>
    phi = np.concatenate([y_sim[i:i+1], phi[:-1]])

return y_sim

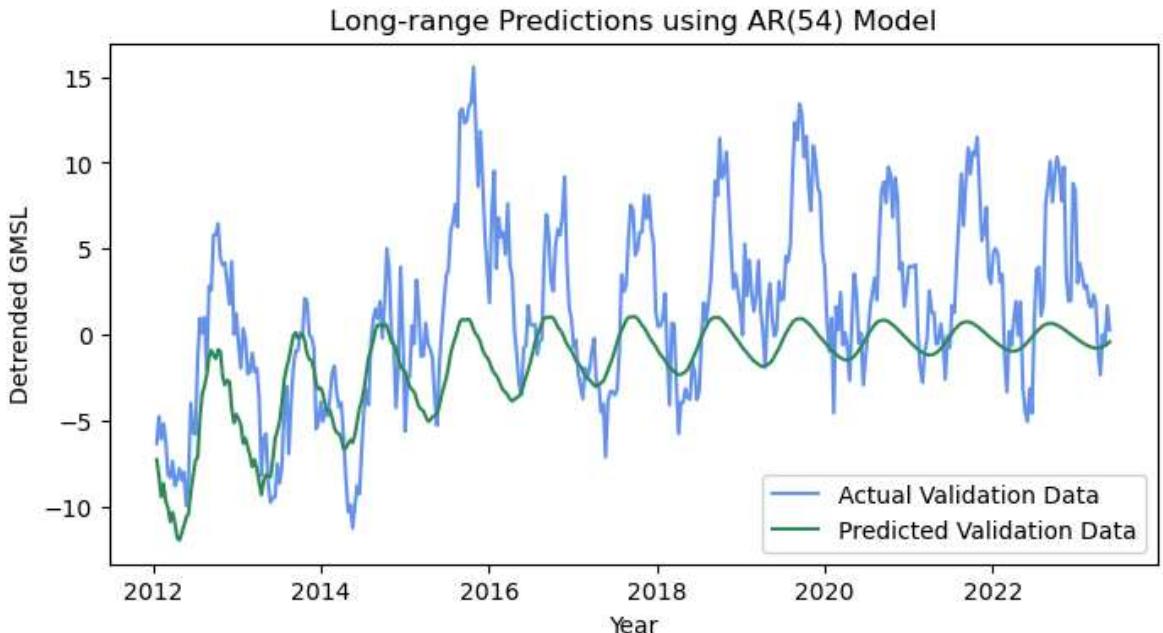
```

```

In [17]: p_new = 54
theta_new = fit_ar(y_train, p_new)
m_steps = len(y_valid)
pred_sim_ar = simulate_ar(y_train, theta_new, m_steps)

# Plot the actual validation data and the predicted validation data values
plt.figure(figsize=(8, 4))
plt.plot(X_valid, y_valid, label='Actual Validation Data', color='cornflowerblue')
plt.plot(X_valid, pred_sim_ar, label='Predicted Validation Data', color='seagreen')
plt.xlabel('Year')
plt.ylabel('Detrended GMSL')
plt.title(f'Long-range Predictions using AR({p_new}) Model')
plt.legend()
plt.show()

```

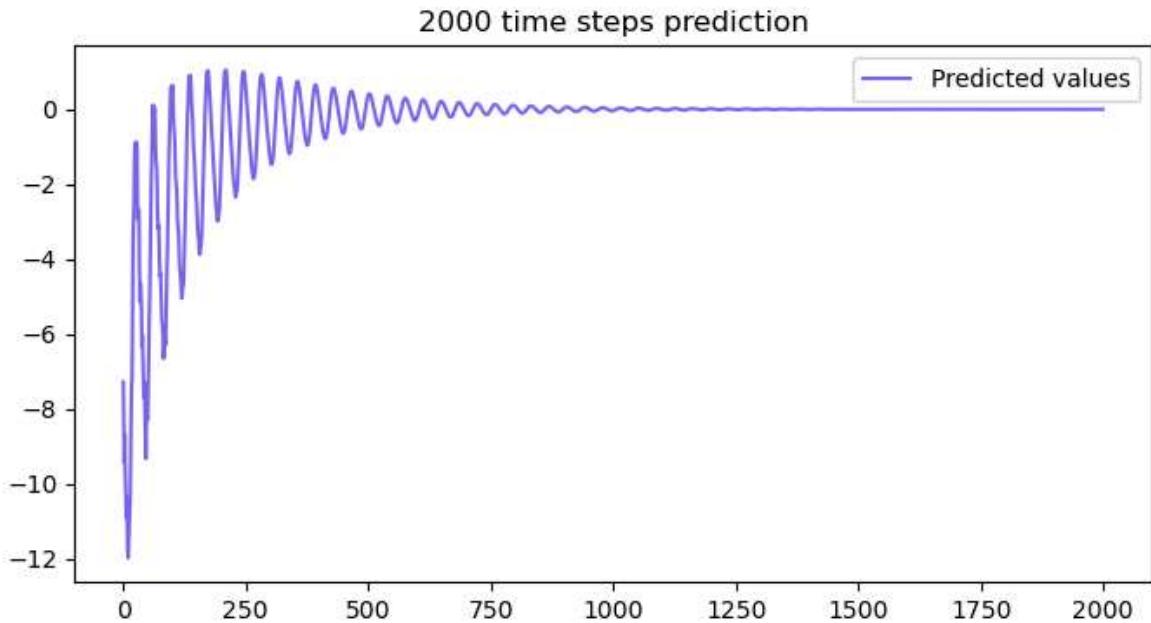


**Q11:** Using the same function as above, try to simulate the process for a large number of time steps (say,  $m = 2000$ ). You should see that the predicted values eventually converge to a constant prediction of zero. Is this something that you would expect to see in general? Explain the result.

### A11:

```
In [18]: # 2000-step predictions
m_steps_2000 = 2000
pred_sim_ar_2000 = simulate_ar(y_train, theta_new, m_steps_2000)

# Plot the 2000 time steps prediction values
plt.figure(figsize=(8, 4))
plt.plot(range(2000), pred_sim_ar_2000, label='Predicted values', color= "mediumblue")
plt.title("2000 time steps prediction")
plt.legend()
plt.show()
```



As shown in the graph, the predicted values gradually approach zero after several steps, which is typical for a stationary process without added noise, especially in many AR models when they are stable. After detrending, the data has a mean of zero, and the AR model's coefficients cause the predictions to decay toward that mean over time.

## 1.5 Nonlinear AR model

In this part, we switch to a nonlinear autoregressive (NAR) model, which is based on a feedforward neural network. This means that in this model the recursive equation for making predictions is still in the form  $\hat{y}_t = f_\theta(y_{t-1}, \dots, y_{t-p})$ , but this time  $f$  is a nonlinear function learned by the neural network. Fortunately almost all of the work for implementing the neural network and training it is handled by the `scikit-learn` package with a few lines of code, and we just need to choose the right structure, and prepare the input-output data.

**Q12:** Construct a NAR( $p$ ) model with a feedforward (MLP) network, by using the `MLPRegressor` class from `scikit-learn`. Set  $p$  to the same value as you chose for the linear AR model above. Initially, you can use an MLP with a single hidden layer consisting of 10 hidden neurons. Train it using the same training data as above and plot

the one-step-ahead predictions as well as the residuals, on both the training and validation data.

*Hint:* You will need the methods `fit` and `predict` of `MLPRegressor`. Read the user guide of `scikit-learn` for more details. Recall that a NAR model is conceptually very similar to an AR model, so you can reuse part of the code from above.

## A12:

```
In [19]: p = 54
# Define MLP model
MLP = MLPRegressor(hidden_layer_sizes=(10,), activation='tanh', max_iter=5000)

# Training points
len_train = len(y_train)

# Construct the regression matrix Phi for training data
m = len_train - p
phi_tr = np.zeros((m, p))
for j in range(p):
    phi_tr[:, j] = y_train[p - j - 1:len_train - j - 1]

# Drop the first p values from the target vector y
yy_train = y_train[p:].to_numpy().flatten()

# Fit MLP
MLP.fit(X=phi_tr, y=yy_train)

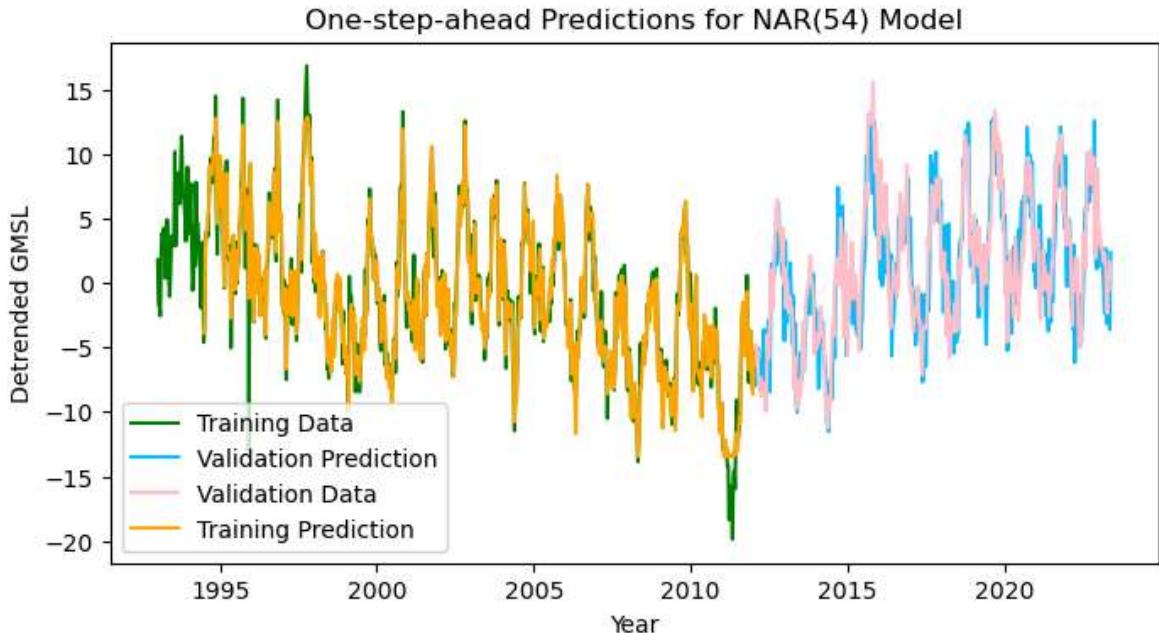
# 1-step ahead prediction for training data
y_pred_train = MLP.predict(phi_tr)

# Validation points: Include the last p values from the training data for constr
new_y_valid = np.concatenate([y_train[len_train - p:], y_valid])
len_valid = len(new_y_valid)

# Construct the regression matrix Phi for validation data
phi_val = np.zeros((len_valid - p, p))
for j in range(p):
    phi_val[:, j] = new_y_valid[p - j - 1:len_valid - j - 1]
# 1-step ahead prediction for validation data
y_pred_valid = MLP.predict(phi_val)

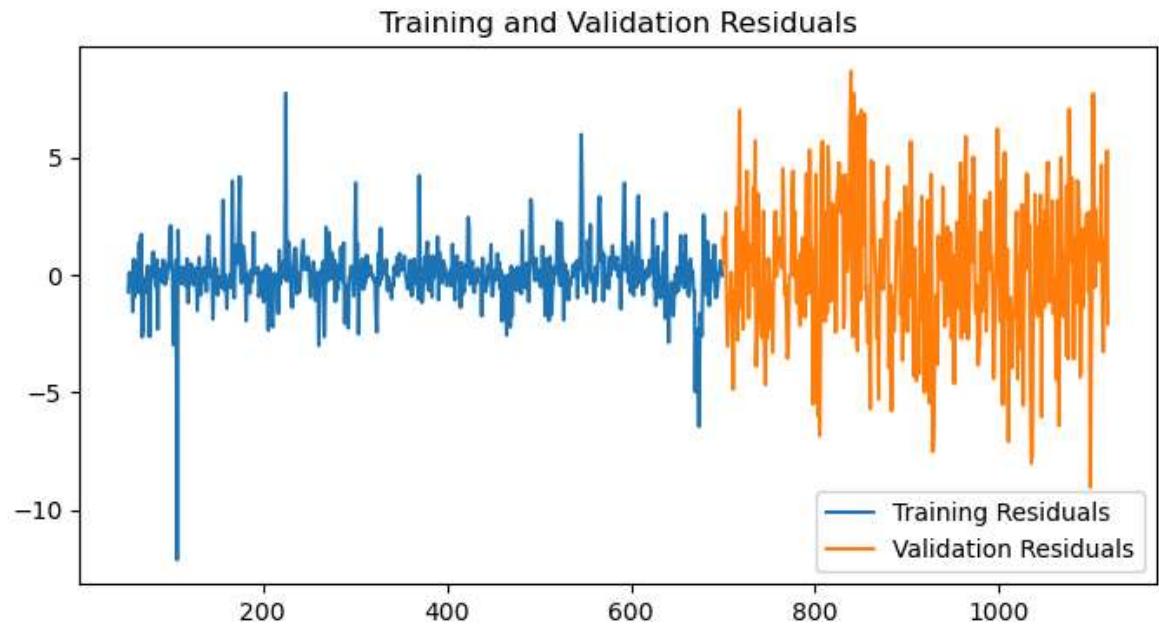
#plot
plt.figure(figsize=(8, 4))
plt.plot(X_train, y_train, label = "Training Data", color='green')
plt.plot(X_valid, y_pred_valid, label = "Validation Prediction", color='deepskyblue')
plt.plot(X_valid, y_valid, label = "Validation Data", color='pink')
plt.plot(X_train[p:], y_pred_train, label = "Training Prediction", color='orange')

# Add Labels, title, and Legend
plt.xlabel('Year')
plt.ylabel('Detrended GMSL')
plt.title(f'One-step-ahead Predictions for NAR({p}) Model')
plt.legend()
plt.show()
```



```
In [20]: #residuals
train_residual = np.squeeze(y_train[p:]) - y_pred_train
validation_residual = np.squeeze(y_valid) - y_pred_valid

plt.figure(figsize=(8, 4))
plt.plot(train_residual, label = "Training Residuals")
plt.plot(validation_residual, label = "Validation Residuals")
plt.title("Training and Validation Residuals")
plt.legend()
plt.show()
```



**Q13:** Try to experiment with different choices for the hyperparameters of the network (e.g. number of hidden layers and units per layer, activation function, etc.) and the optimizer (e.g. `solver` and `max_iter` ).

Are you satisfied with the results? Why/why not? Discuss what the limitations of this approach might be.

### A13:

For different hidden\_layer\_sizes:

```
In [21]: p = 54
# Define MLP model
MLP = MLPRegressor(hidden_layer_sizes=(33,33), activation='tanh', max_iter=5000)

# Training points
len_train = len(y_train)

# Construct the regression matrix Phi for training data
m = len_train - p
phi_tr = np.zeros((m, p))
for j in range(p):
    phi_tr[:, j] = y_train[p - j - 1:len_train - j - 1]

# Drop the first p values from the target vector y
yy_train = y_train[p:].to_numpy().flatten()

# Fit MLP
MLP.fit(X=phi_tr, y=yy_train)

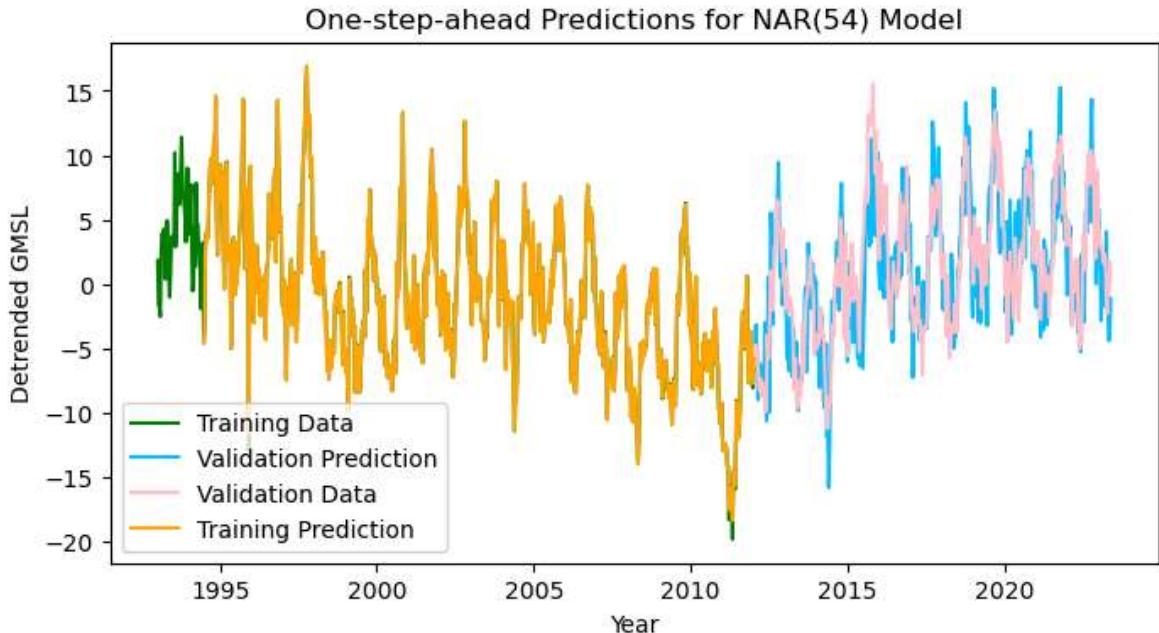
# 1-step ahead prediction for training data
y_pred_train = MLP.predict(phi_tr)

# Validation points: Include the last p values from the training data for constr
new_y_valid = np.concatenate([y_train[len_train - p:], y_valid])
len_valid = len(new_y_valid)

# Construct the regression matrix Phi for validation data
phi_val = np.zeros((len_valid - p, p))
for j in range(p):
    phi_val[:, j] = new_y_valid[p - j - 1:len_valid - j - 1]
# 1-step ahead prediction for validation data
y_pred_valid = MLP.predict(phi_val)

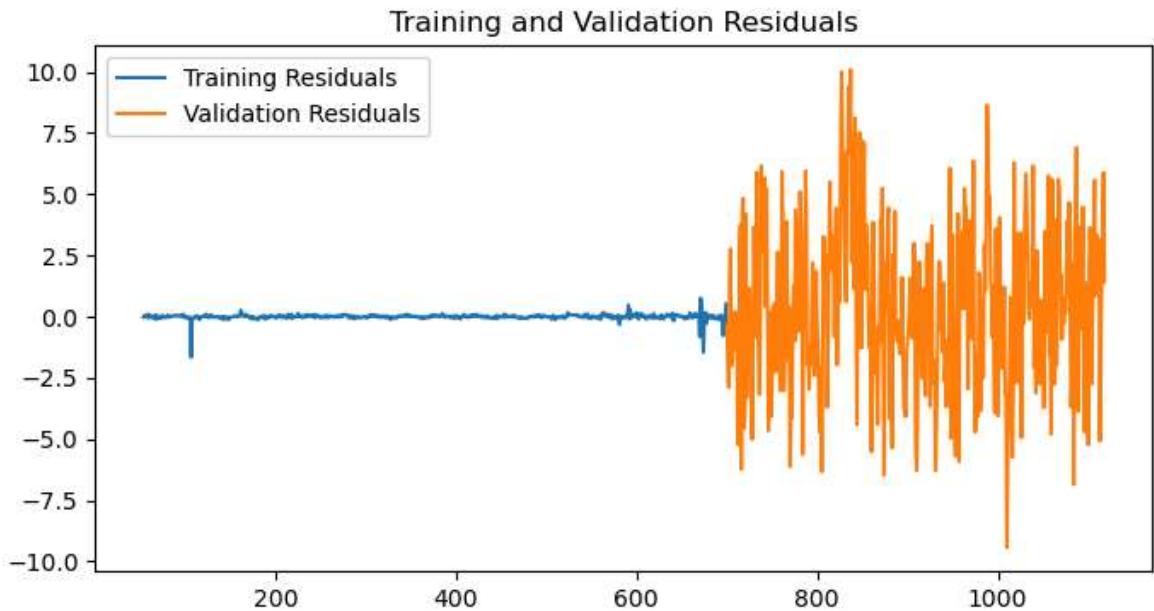
#plot
plt.figure(figsize=(8, 4))
plt.plot(X_train, y_train, label = "Training Data", color='green')
plt.plot(X_valid, y_pred_valid, label = "Validation Prediction", color='deepskyblue')
plt.plot(X_valid, y_valid, label = "Validation Data", color='pink')
plt.plot(X_train[p:], y_pred_train, label = "Training Prediction", color='orange')

# Add Labels, title, and legend
plt.xlabel('Year')
plt.ylabel('Detrended GMSL')
plt.title(f'One-step-ahead Predictions for NAR({p}) Model')
plt.legend()
plt.show()
```



```
In [22]: #residuals
train_residual = np.squeeze(y_train[p:]) - y_pred_train
validation_residual = np.squeeze(y_valid) - y_pred_valid

plt.figure(figsize=(8, 4))
plt.plot(train_residual, label = "Training Residuals")
plt.plot(validation_residual, label = "Validation Residuals")
plt.title("Training and Validation Residuals")
plt.legend()
plt.show()
```



For different activation function:

```
In [23]: p = 54
# Define MLP model
MLP = MLPRegressor(hidden_layer_sizes=(33,33), activation='logistic', max_iter=5

# Training points
len_train = len(y_train)
```

```

# Construct the regression matrix Phi for training data
m = len_train - p
phi_tr = np.zeros((m, p))
for j in range(p):
    phi_tr[:, j] = y_train[p - j - 1:len_train - j - 1]

# Drop the first p values from the target vector y
yy_train = y_train[p: ].to_numpy().flatten()

# Fit MLP
MLP.fit(X=phi_tr, y=yy_train)

# 1-step ahead prediction for training data
y_pred_train = MLP.predict(phi_tr)

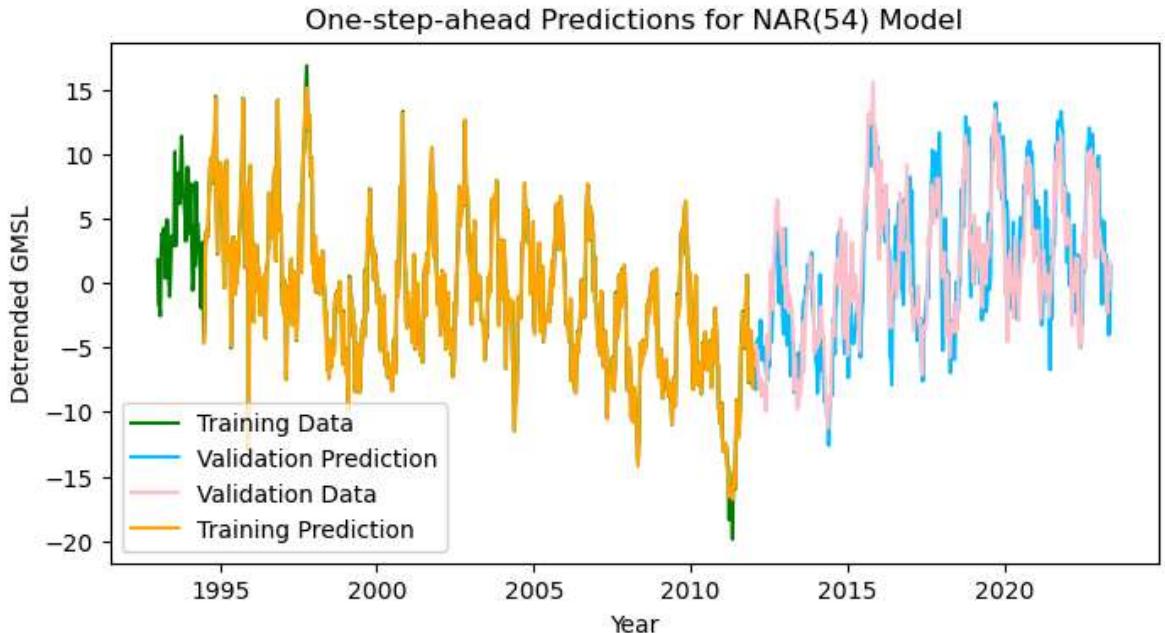
# Validation points: Include the last p values from the training data for constr
new_y_valid = np.concatenate([y_train[len_train - p:], y_valid])
len_valid = len(new_y_valid)

# Construct the regression matrix Phi for validation data
phi_val = np.zeros((len_valid - p, p))
for j in range(p):
    phi_val[:, j] = new_y_valid[p - j - 1:len_valid - j - 1]
# 1-step ahead prediction for validation data
y_pred_valid = MLP.predict(phi_val)

#plot
plt.figure(figsize=(8, 4))
plt.plot(X_train, y_train, label = "Training Data", color='green')
plt.plot(X_valid, y_pred_valid, label = "Validation Prediction", color='deepskyblue')
plt.plot(X_valid, y_valid, label = "Validation Data", color='pink')
plt.plot(X_train[p:], y_pred_train, label = "Training Prediction", color='orange')

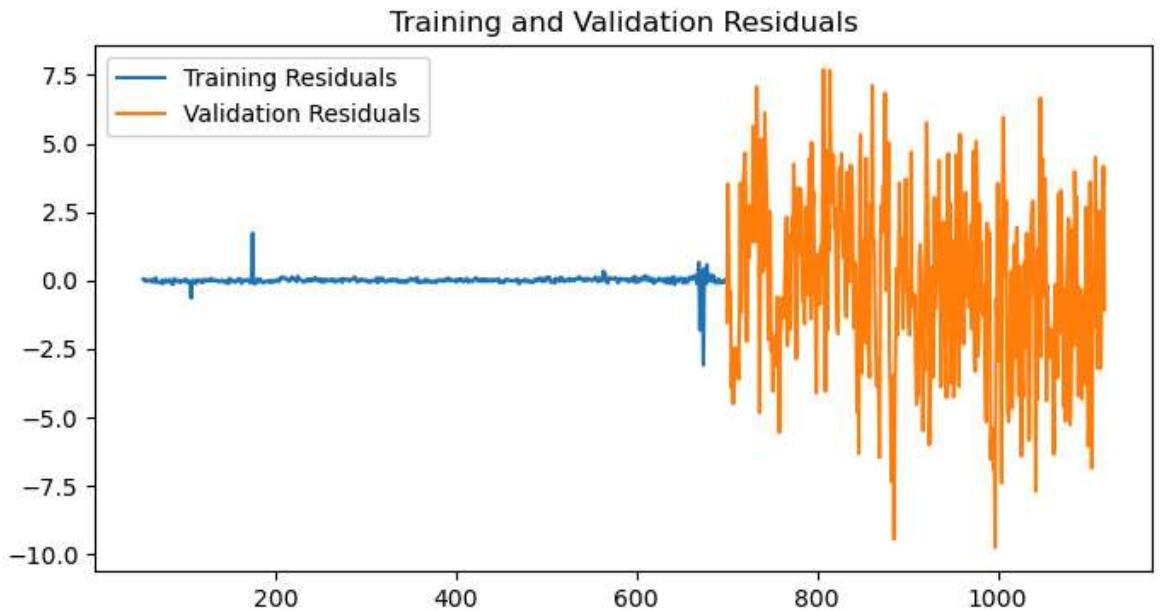
# Add Labels, title, and Legend
plt.xlabel('Year')
plt.ylabel('Detrended GMSL')
plt.title(f'One-step-ahead Predictions for NAR({p}) Model')
plt.legend()
plt.show()

```



```
In [24]: #residuals
train_residual = np.squeeze(y_train[p:]) - y_pred_train
validation_residual = np.squeeze(y_valid) - y_pred_valid

plt.figure(figsize=(8, 4))
plt.plot(train_residual, label = "Training Residuals")
plt.plot(validation_residual, label = "Validation Residuals")
plt.title("Training and Validation Residuals")
plt.legend()
plt.show()
```



Setting solver parameter as "Adam":

```
In [25]: p = 54
# Define MLP model
MLP = MLPRegressor(hidden_layer_sizes=(10,), activation='tanh', solver = "adam",

# Training points
len_train = len(y_train)
```

```

# Construct the regression matrix Phi for training data
m = len_train - p
phi_tr = np.zeros((m, p))
for j in range(p):
    phi_tr[:, j] = y_train[p - j - 1:len_train - j - 1]

# Drop the first p values from the target vector y
yy_train = y_train[p: ].to_numpy().flatten()

# Fit MLP
MLP.fit(X=phi_tr, y=yy_train)

# 1-step ahead prediction for training data
y_pred_train = MLP.predict(phi_tr)

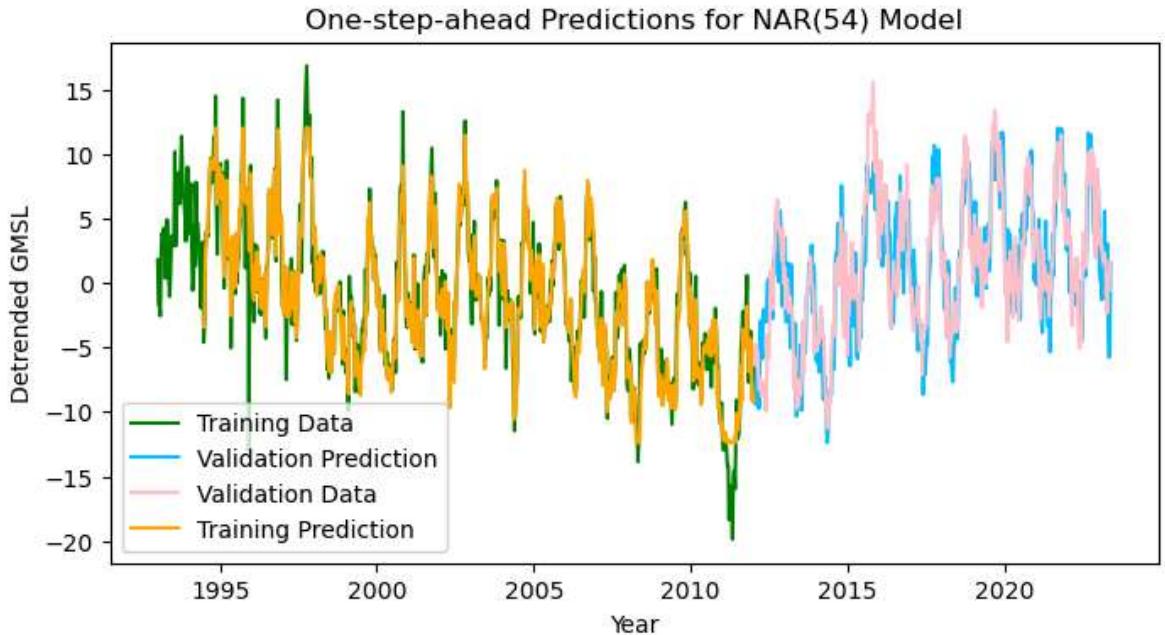
# Validation points: Include the last p values from the training data for constr
new_y_valid = np.concatenate([y_train[len_train - p:], y_valid])
len_valid = len(new_y_valid)

# Construct the regression matrix Phi for validation data
phi_val = np.zeros((len_valid - p, p))
for j in range(p):
    phi_val[:, j] = new_y_valid[p - j - 1:len_valid - j - 1]
# 1-step ahead prediction for validation data
y_pred_valid = MLP.predict(phi_val)

#plot
plt.figure(figsize=(8, 4))
plt.plot(X_train, y_train, label = "Training Data", color='green')
plt.plot(X_valid, y_pred_valid, label = "Validation Prediction", color='deepskyblue')
plt.plot(X_valid, y_valid, label = "Validation Data", color='pink')
plt.plot(X_train[p:], y_pred_train, label = "Training Prediction", color='orange')

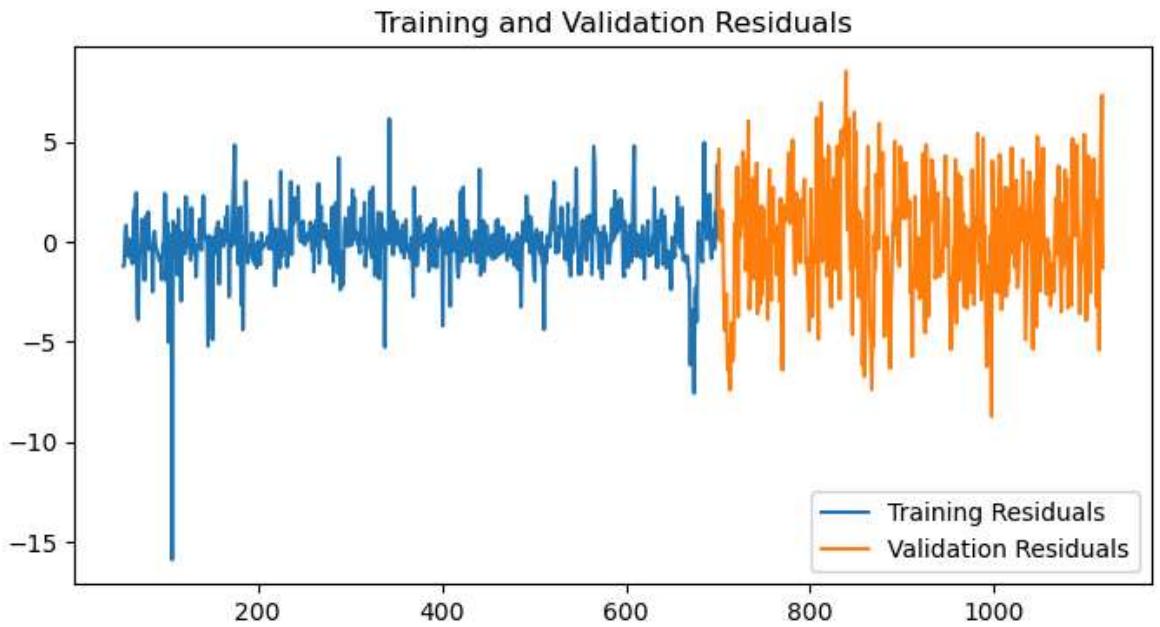
# Add Labels, title, and Legend
plt.xlabel('Year')
plt.ylabel('Detrended GMSL')
plt.title(f'One-step-ahead Predictions for NAR({p}) Model')
plt.legend()
plt.show()

```



```
In [26]: #residuals
train_residual = np.squeeze(y_train[p:]) - y_pred_train
validation_residual = np.squeeze(y_valid) - y_pred_valid

plt.figure(figsize=(8, 4))
plt.plot(train_residual, label = "Training Residuals")
plt.plot(validation_residual, label = "Validation Residuals")
plt.title("Training and Validation Residuals")
plt.legend()
plt.show()
```



For different max\_iter:

```
In [27]: p = 54
# Define MLP model
MLP = MLPRegressor(hidden_layer_sizes=(10,), activation='tanh', max_iter=20000)

# Training points
len_train = len(y_train)
```

```

# Construct the regression matrix Phi for training data
m = len_train - p
phi_tr = np.zeros((m, p))
for j in range(p):
    phi_tr[:, j] = y_train[p - j - 1:len_train - j - 1]

# Drop the first p values from the target vector y
yy_train = y_train[p:].to_numpy().flatten()

# Fit MLP
MLP.fit(X=phi_tr, y=yy_train)

# 1-step ahead prediction for training data
y_pred_train = MLP.predict(phi_tr)

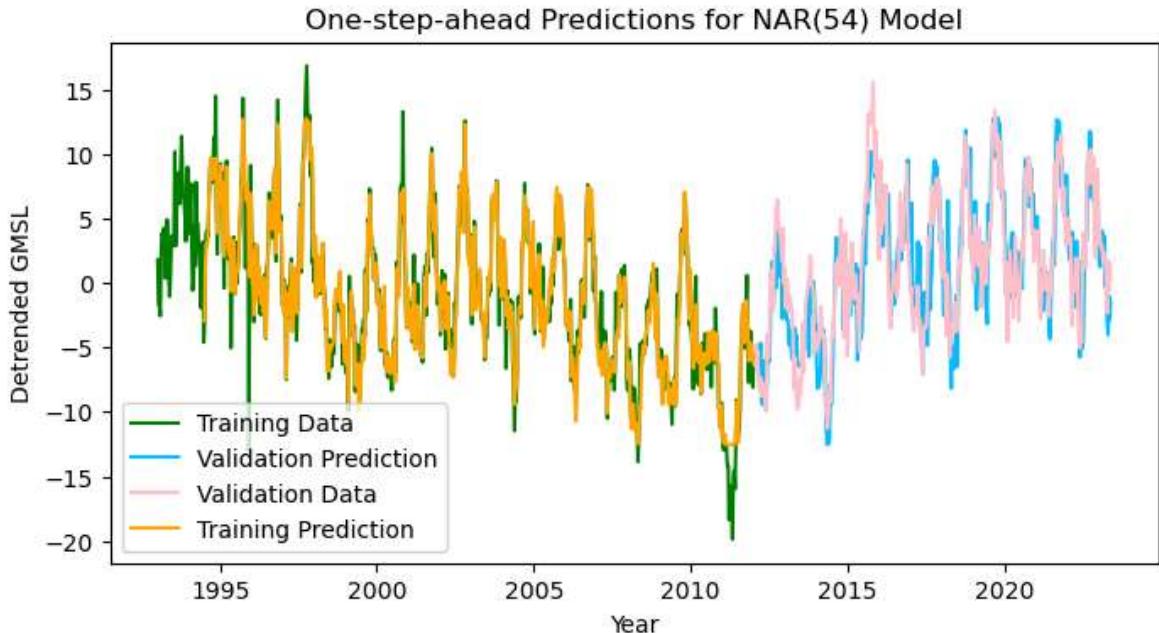
# Validation points: Include the last p values from the training data for constr
new_y_valid = np.concatenate([y_train[len_train - p:], y_valid])
len_valid = len(new_y_valid)

# Construct the regression matrix Phi for validation data
phi_val = np.zeros((len_valid - p, p))
for j in range(p):
    phi_val[:, j] = new_y_valid[p - j - 1:len_valid - j - 1]
# 1-step ahead prediction for validation data
y_pred_valid = MLP.predict(phi_val)

#plot
plt.figure(figsize=(8, 4))
plt.plot(X_train, y_train, label = "Training Data", color='green')
plt.plot(X_valid, y_pred_valid, label = "Validation Prediction", color='deepskyblue')
plt.plot(X_valid, y_valid, label = "Validation Data", color='pink')
plt.plot(X_train[p:], y_pred_train, label = "Training Prediction", color='orange')

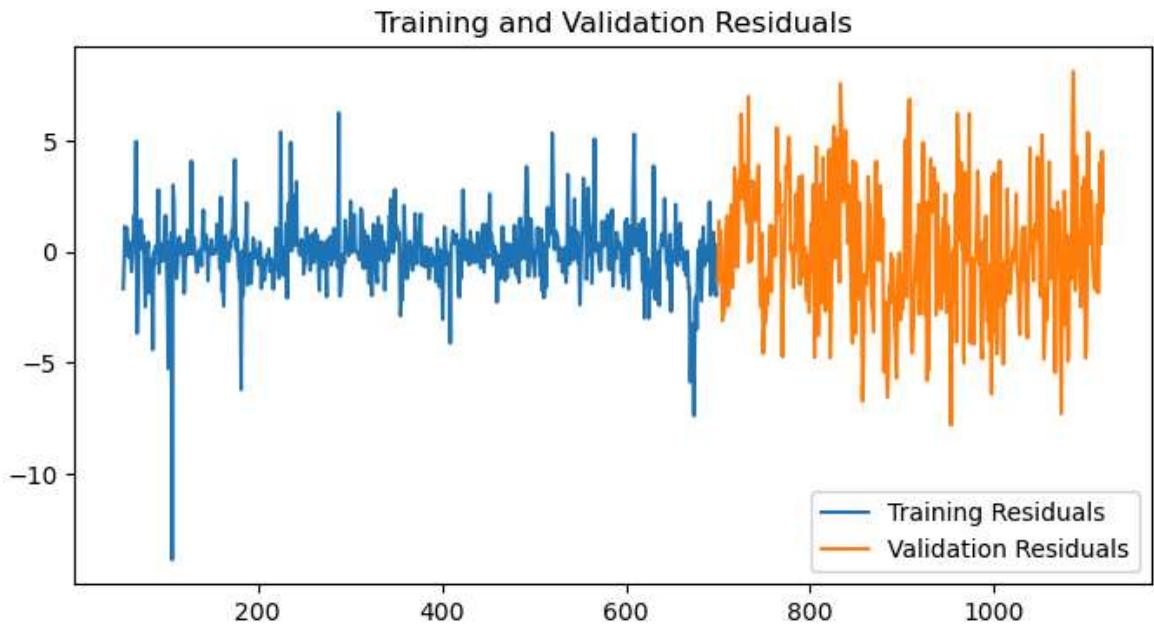
# Add labels, title, and legend
plt.xlabel('Year')
plt.ylabel('Detrended GMSL')
plt.title(f'One-step-ahead Predictions for NAR({p}) Model')
plt.legend()
plt.show()

```



```
In [28]: #residuals
train_residual = np.squeeze(y_train[p:]) - y_pred_train
validation_residual = np.squeeze(y_valid) - y_pred_valid

plt.figure(figsize=(8, 4))
plt.plot(train_residual, label = "Training Residuals")
plt.plot(validation_residual, label = "Validation Residuals")
plt.title("Training and Validation Residuals")
plt.legend()
plt.show()
```



When the hidden layers is increased (set as (33,33)), our training data fits better but on validation data we can see that there is no significantly change in prediction and residual values on validation data. We should set hidden layers and unit carefully due to avoiding the overfitting problem.

When we change the activation function as logistic function, our training predictions data fits better however our validation predictions data fits worse than before. Logistic

activation function leads to vanishing gradient problem when we using hidden layers. That's why the model cannot be generalized better.

Adam algorithm uses adaptive learning rates and often it converges faster than other solver algorithms. In our task, there can be risk of vanishing gradient problem because of tanh activation but adam algotihm helps to avoid this problem.

In more complex networks (like `hidden_layer_sizes=(33,33)`), we need more iteration to converge for good solution. If there is no enough limit of iterations, model can stop before fully converge. In this task, we need more iterations because our network is complex than "`hidden_layer_sizes=(10)`".