Quantitative Methods in FIxed Income - Assignment 1

Group 19:

- Max Dijkman (509960)
- Pieter Pel (525573)
- Jan Tempel (497053)

Source of data: https://www.bankofengland.co.uk/statistics/yield-curves

Direct download link: https://www.bankofengland.co.uk/-/media/boe/files/statistics/yield-curves/glcnominalddata.zip

Import stuff

In [ ]:
```
import os # changing directories
import pandas as pd # storing data
import copy # make copys
import matplotlib.pyplot as plt # plots
import seaborn as sns # plots
import datetime as dt # datetime objects
from math import exp, sqrt # exponential and square root function
import numpy as np # arrays
from statsmodels.tsa.api import VAR # VAR model
```

Methods and classes

In [ ]:
```
def isfloat(str: str) -> bool:
        '''Method that returns wether a string only consists of digits and periods'

        str_list = [*str]

        for char in str_list:
            if not (char.isnumeric() or char == '.'):
                return False

        return True

class MaturitiesDf(pd.DataFrame):
    '''Class that extends the pandas DataFrame, which provides some handy features

    def __init__(self, *args, **kwargs):
        # use the __init__ method from DataFrame to ensure
        # that we're inheriting the correct behavior
        super(MaturitiesDf, self).__init__(*args, **kwargs)

        self.date_col = None
```

```
        self.year_dict = None

        if not self.date_col is None:
            self.year_dict = self.get_year_dict()

        # Reset the indices
        self.reset_index(drop=True, inplace=True)

    @property
    # this method is makes it so our methods return an instance
    # of MaturitiesDf, instead of a regular DataFrame
    def _constructor(self):
        return MaturitiesDf

    # Allows self.date_col and self.year_dict to carry over
    def __getitem__(self, key):
        result = super(MaturitiesDf, self).__getitem__(key)

        # Check if the result is a DataFrame (e.g., it's a subsection)
        if isinstance(result, pd.DataFrame):
            # Copy over attributes
            result.date_col = self.date_col
            result.year_dict = self.year_dict

        return result

    def maturities_until(self, mat: float | str):
        '''Method that returns the maturities of a dataframe up until a certain mat

        maturities = [col for col in self.columns if isfloat(col) and float(col) <=

        return maturities

    def maturities_from(self, mat: float| str):
        '''Method that returns the maturities of a dataframe from a certain maturit

        maturities = [col for col in self.columns if isfloat(col) and float(col) >=

        return maturities

    def get_year_dict(self):
        '''Method that returns a dictionary that contains the starting indices of e

        if self.date_col is None:
            raise Exception("You must first specify the date column using [NAME OF

        year_dict = dict()

        # Return empty dictionary if the dataframe is emtpy
        if len(self) == 0:
            return year_dict

        # Loop over all rows of the date column
        for index in self.index:

                # If the year isn't in the dictionary add it
```

```
                year = str(self.iloc[index][self.date_col].year)
                if not year in year_dict.keys():

                    year_dict[year] = index
                    year_dict[int(year)] = index

        return year_dict

    def get_year_start_index(self, year: int | str) -> int:
        '''Method that returns the starting index of a year'''

        if self.year_dict is None:
            self.year_dict = self.get_year_dict()

        return self.year_dict[year]

    def get_year_end_index(self, year: int | str):
        '''Method that returns the ending index of a year'''

        if self.year_dict is None:
            self.year_dict = self.get_year_dict()

        return self.year_dict[int(year) + 1]

    def clean_df(self):
        '''Method that cleans the provided dataframe'''
        print('Cleaning the dataframe')

        cleaned_df = copy.copy(self)

        # Rename columns
        column_dict = {cleaned_df.columns[index]: str(x) for index, x in enumerate(
        column_dict['Unnamed: 0'] = 'date'
        cleaned_df = cleaned_df.rename(columns = column_dict)

        # Drop empty rows
        cleaned_df = cleaned_df.dropna(thresh=2)
        cleaned_df = cleaned_df[cleaned_df['date'] != 'years:']

        # Reset indices
        cleaned_df.reset_index()

        return cleaned_df
```

Load data

In [ ]:
```
# Excel files to extract
excel_names = ['GLC Nominal daily data_1979 to 1984.xlsx',
               'GLC Nominal daily data_1985 to 1989.xlsx',
               'GLC Nominal daily data_1990 to 1994.xlsx',
               'GLC Nominal daily data_1995 to 1999.xlsx',
               'GLC Nominal daily data_2000 to 2004.xlsx',
               'GLC Nominal daily data_2005 to 2015.xlsx',
               'GLC Nominal daily data_2016 to present.xlsx',]
```

```
# Make dictionary with the correct sheet names
sheet_names = {name: '4. nominal spot curve' for name in excel_names}
sheet_names['GLC Nominal daily data_2005 to 2015.xlsx'] = '4. spot curve'
sheet_names['GLC Nominal daily data_2016 to present.xlsx'] = '4. spot curve'

os.chdir('..') # Starts under /src, go to parent

df = MaturitiesDf(pd.DataFrame())

for name in excel_names:

    # Load single sheet as dataframe and automatically clean the sheet (handy!)
    df_single = MaturitiesDf(pd.read_excel(f'data/{name}', sheet_name=sheet_names[n
    df_single = df_single.clean_df()

    # Concat the dataframes
    df = pd.concat([df, df_single])

os.chdir(os.getcwd() + '/src') # Go bak to /src
```

```
Cleaning the dataframe
Cleaning the dataframe
Cleaning the dataframe
Cleaning the dataframe
Cleaning the dataframe
Cleaning the dataframe
Cleaning the dataframe
```

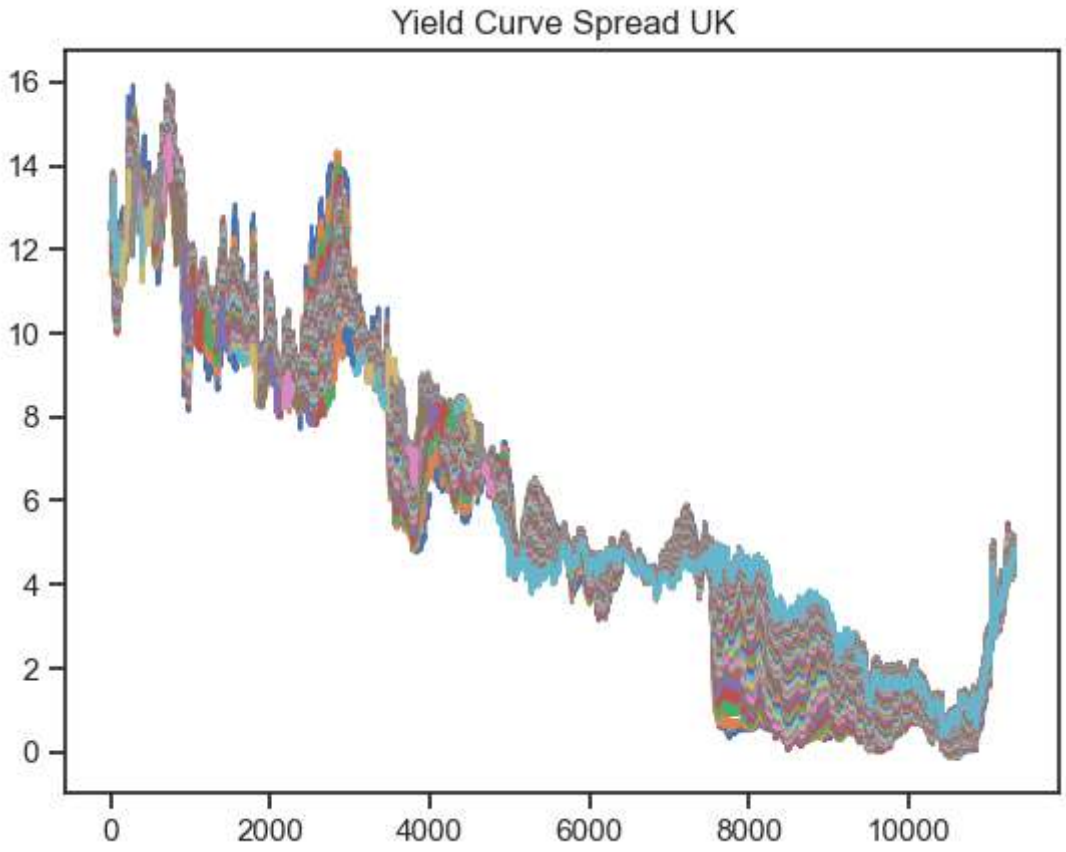
In [ ]:
```
# Note: maturities 0.5 and 20+ have a lot of 'missing' data, 1 has 15; could consid

df.date_col = 'date'
df.year_dict = df.get_year_dict()
```

Part I - Data

All available maturities

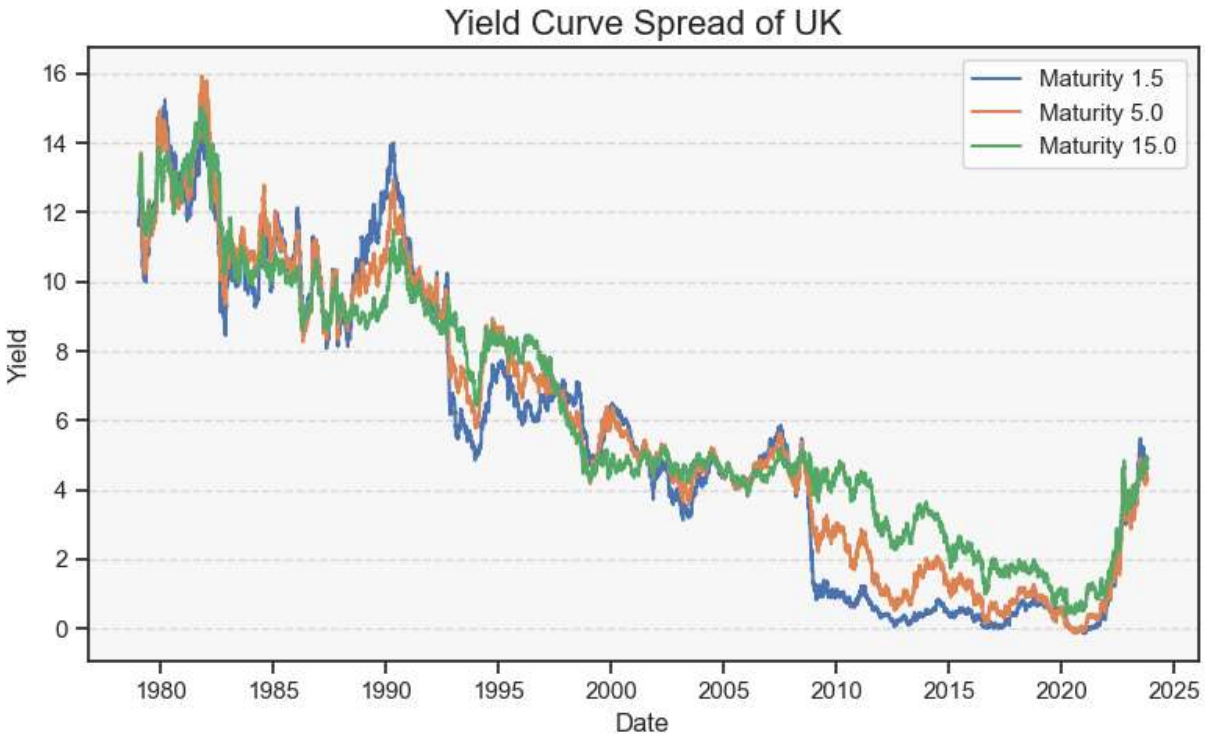
In [ ]:
```
# Set a seaborn style
sns.set(style = "ticks")

df.plot(x='date', y=df.columns[1:],
        legend = False,
        title = 'Yield Curve Spread UK')

# NOTE: SPREAD APPEARS LARGER AT THE END BECAUSE THERE ARE HIGHER MATURITIES AVAILA
```

Out[ ]:
```
<Axes: title={'center': 'Yield Curve Spread UK'}>
```



Subsection of maturities

In [ ]:
```
maturities_to_plot = ['1.5', '5.0', '15.0']

# Plot
plt.figure(figsize=(8, 5))

for maturity in maturities_to_plot:
    sns.lineplot(data=df, x='date', y=maturity, label=f'Maturity {maturity}', linew

# Add grid lines
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Set background color
plt.gca().set_facecolor('#F7F7F7')

# Add legend
plt.legend(loc="upper right")

# Title and labels
plt.title('Yield Curve Spread of UK', fontsize=16)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Yield', fontsize=12)

# Display the plot
plt.tight_layout()
plt.show()
```

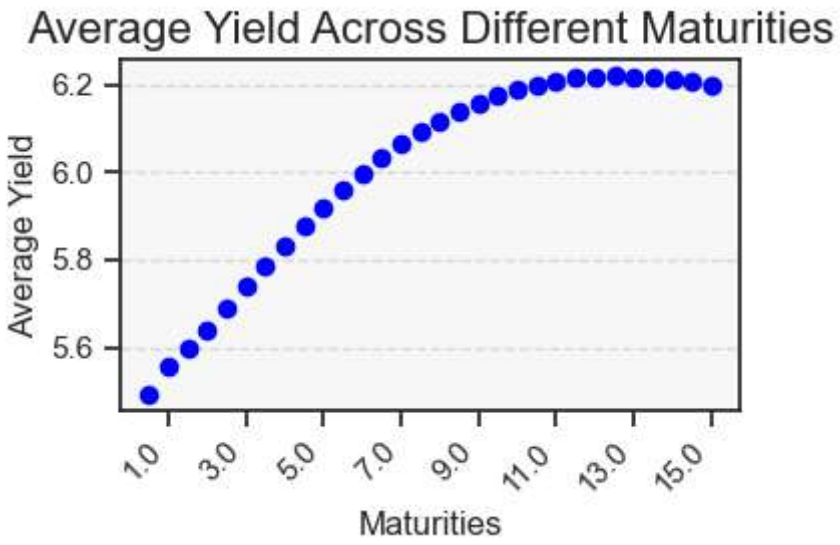


Part II - Stylized Facts

SF I: The average yield curve over time is increasing and concave

In [ ]:
```
maturities_to_plot = df.maturities_until(15)
averages = df[maturities_to_plot].mean()

def plot_maturities_against(maturities_to_plot, against, title, y_label, xticks=mat
    # Plot
    plt.figure(figsize=(4, 3))

    # Scatter plot with markers and labels
    plt.scatter(maturities_to_plot, against, marker='o', color='blue', label=y_labe

    # Add title and labels
    plt.title(title, fontsize=16)
    plt.xlabel('Maturities', fontsize=12)
    plt.ylabel(y_label, fontsize=12)

    # Add grid lines
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    # Set background color
    plt.gca().set_facecolor('#F7F7F7')

    # Customize x-axis ticks
    plt.xticks(rotation=45, ha='right')  # Rotate ticks for better readability
    plt.xticks(xticks)  # Adjust the number of ticks

    # Display the plot
    plt.tight_layout()
```

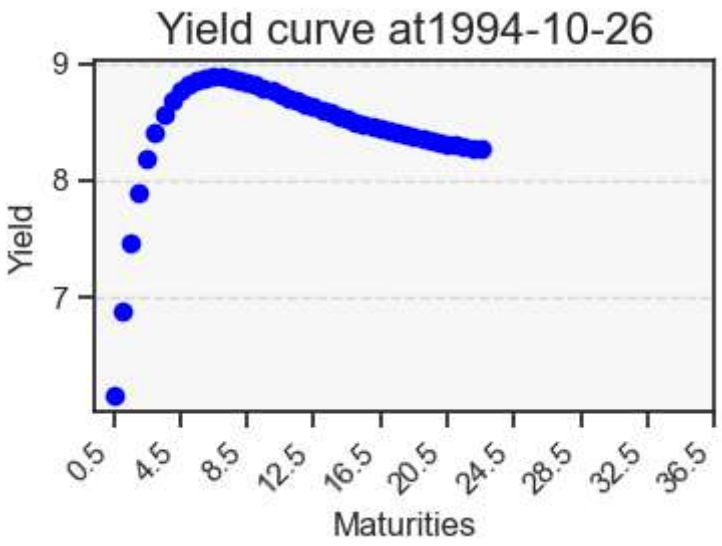
```
    plt.show()

plot_maturities_against(maturities_to_plot, averages, 'Average Yield Across Differe
```



SF II: The yield curve can take on a variety of shapes

In [ ]:
```
# Interesting indices: 340, ...
index = 4000
print(df.iloc[index]['date'])

maturities_to_plot = df.maturities_from(0.5)
plot_maturities_against(maturities_to_plot, df.iloc[index][1:], title='Yield curve
```

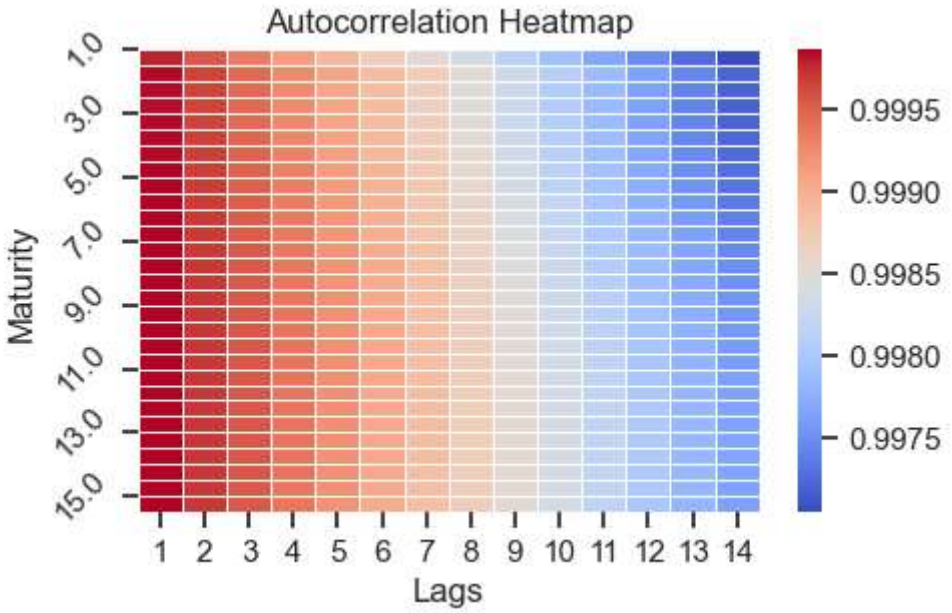
```
1994-10-26 00:00:00
```



SF III: Yield dynamics are (very) persistent

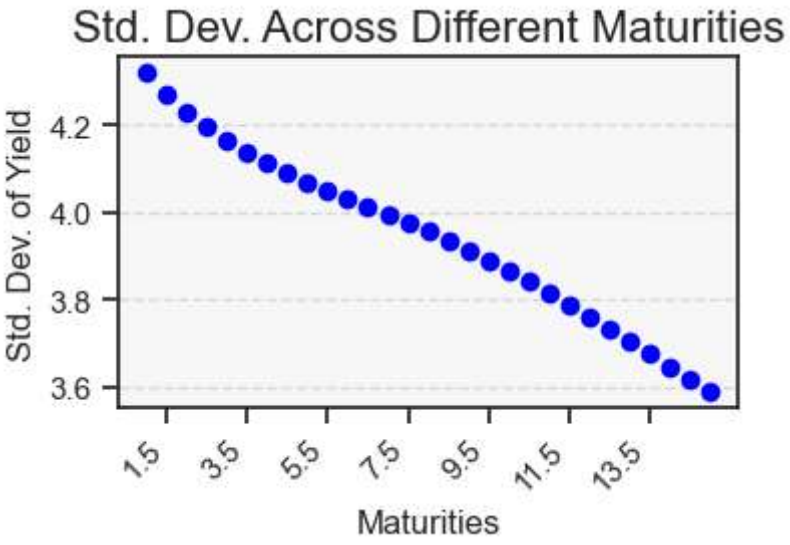
SF IV: Yields for long maturities are more persistent than yield for shorter maturities

In [ ]:
```
maturities_to_plot = df.maturities_until(15)[1:] # Dont include 0.5 due to missing
maturities_to_plot_list = list(maturities_to_plot)

# Compute autocorrelations
lags = range(1, 15)

autocorrs = pd.DataFrame(columns = ['maturity'] + [str(i) for i in lags])

# Loop over all maturities
for col in df.columns[1:]:

    # Loop over all lags
    row = [col]
    for lag in lags:

        # Add autocorrelation to row
        row.append(df[col].autocorr(lag=lag))

    # Add row to dataframe
    autocorrs.loc[len(autocorrs)] = row

# Print autocrrelations
autocorrs[0:len(maturities_to_plot_list)]

# Assuming autocorrs is the DataFrame containing autocorrelation values
plt.figure(figsize=(5, 3))  # Adjust the size according to your preference

# Remove the 'maturity' column for visualization
heatmap_data = autocorrs[0:len(maturities_to_plot_list)].drop(columns='maturity')

# Create the heatmap
sns.heatmap(heatmap_data, cmap="coolwarm", cbar=True, linewidths=.5)#, center=0.999

# Set y ticks
tick_positions = list(range(0, len(maturities_to_plot_list), 4))
plt.yticks(ticks=tick_positions, labels=maturities_to_plot_list[::4], rotation=45)

# Set labels and title
plt.xlabel('Lags')
plt.ylabel('Maturity')
plt.title('Autocorrelation Heatmap')

plt.show()
```



SF V: The short end of the yield curve is more volatile than the long end

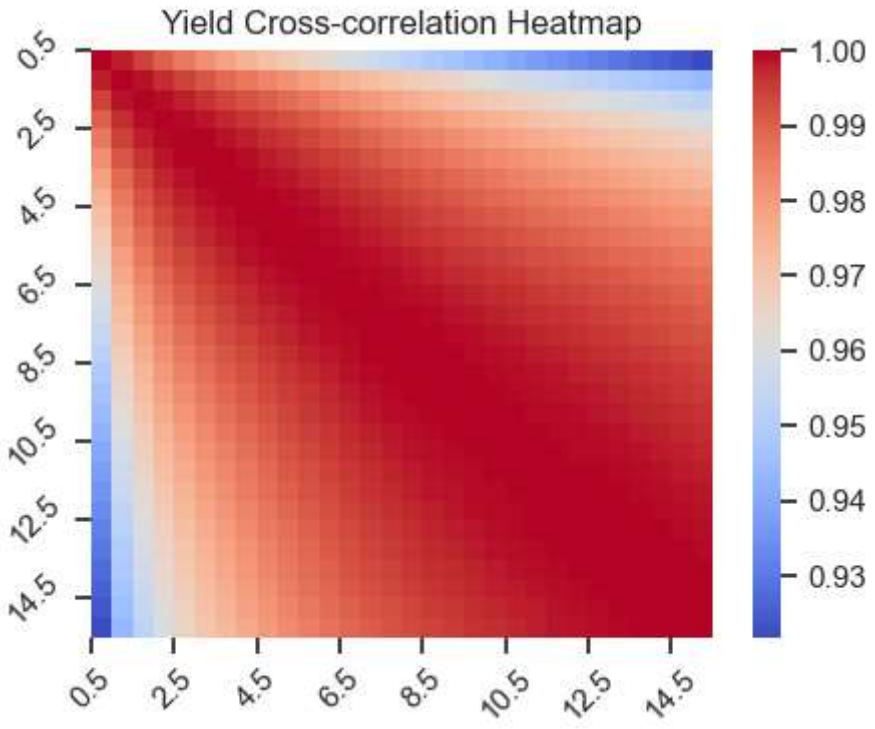
In [ ]:
```
maturities_to_plot = df.maturities_until(15)[1:] # Dont include 0.5 due to missing

indices = range(0, len(df)) # 9353 is 2016
standard_devs = df.iloc[indices][maturities_to_plot].std()

plot_maturities_against(maturities_to_plot, standard_devs.values, 'Std. Dev. Across
```



SF VI: Yield for different maturities have high cross-correlations

In [ ]:
```
# # Create a heatmap of subsample
# indices = range(df.get_year_start_index(2016), len(df))
# correlation_matrix = df.iloc[indices].drop('date', axis=1).corr()

# plt.figure(figsize=(10, 8))  # Adjust the size according to your preference
# sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm', cbar=True)
```

```
# plt.title('Correlation Matrix Heatmap since 2016')
# plt.show()

# Create heatmap of entire sample
maturities_to_plot = df.maturities_until(15)
correlation_matrix = df[maturities_to_plot].iloc[indices].corr()

plt.figure(figsize=(5, 3.8))  # Adjust the size according to your preference
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm', cbar=True)

# Convert maturities_to_plot to a list
maturities_to_plot_list = list(maturities_to_plot)

# Set y-axis tick positions and labels
tick_positions = list(range(0, len(maturities_to_plot_list), 4))
plt.yticks(ticks=tick_positions, labels=maturities_to_plot_list[::4], rotation=45)
plt.xticks(ticks=tick_positions, labels=maturities_to_plot_list[::4], rotation=45)

plt.title('Yield Cross-correlation Heatmap')
plt.show()
```



Part III - Forecasting

- Forecast 20, 60, 125, 250 days ahead using expanding windows
- Number of years: January 1979 to November 2023 => 45.92 years => rounded down, 60% is 27 years
- Thus in-sample: 1979 to end 2005, out-of-sample: 2006 to November 2023

Random Walk

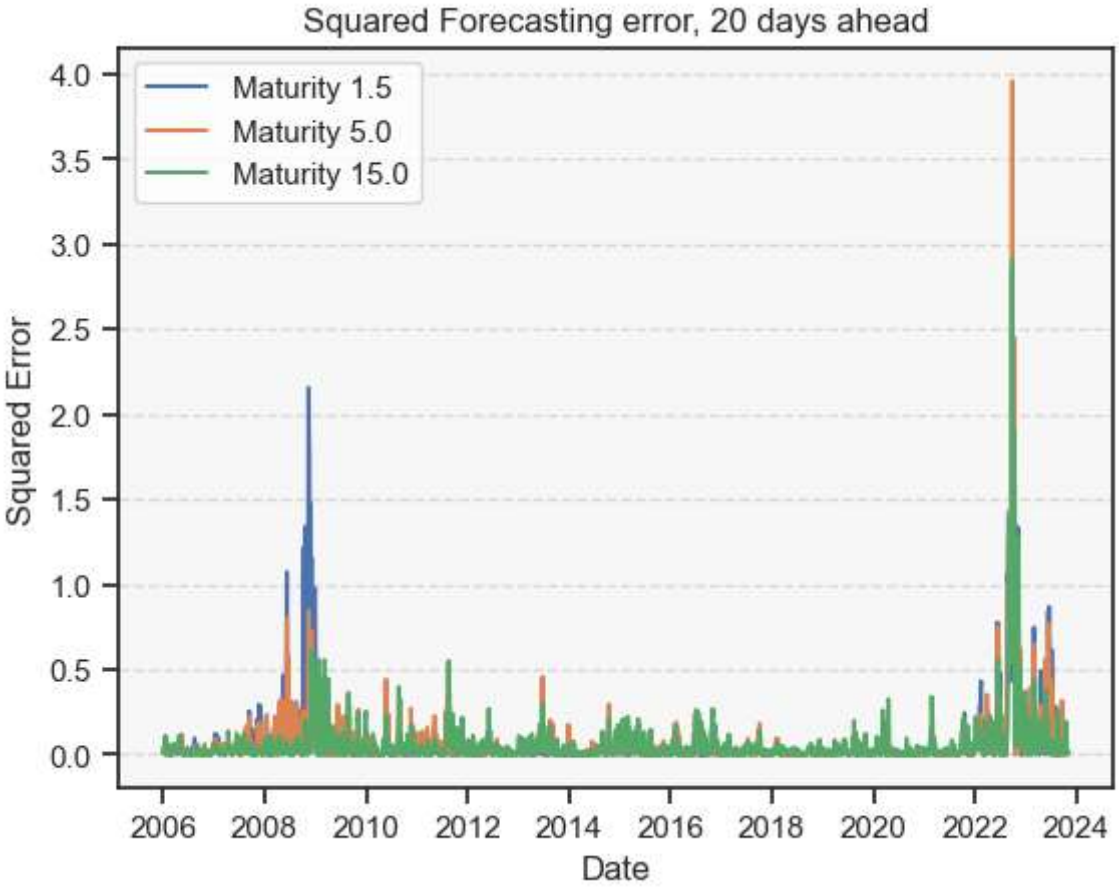
In [ ]:
```
horizon = 20 # days

from_maturity = df.maturities_from(1.5)
until_maturity = df.maturities_until(15)
dns_maturities = [x for x in from_maturity if x in until_maturity]
```

In [ ]:
```
def get_rw_forecasts(df, maturities, horizon):
    '''Method that obtains the random walk forecast for given maturitities and hori

    # Copy the yield dataframe and drop the rows that won't have a prediction
    forecasts_df = copy.copy(df)
    forecasts_df.date_col = df.date_col
    forecasts_df = forecasts_df.iloc[horizon:]

    # Loop over all maturities
    for maturity in df.maturities_from(0.5):

        # Extract the forecasts and drop the ones that won't be used for prediction
        forecasts = df[maturity].iloc[:-horizon]

        # Put the forecasts in the dataframe (effectively shifting the yield values
        forecasts_df.loc[:, maturity] = forecasts.values

    # Drop the columns that are not needed
    forecasts_df = forecasts_df[forecasts_df.columns.intersection(maturities + [df.

    return forecasts_df
```

In [ ]:
```
# Make dataframe that has all the squared errors
def get_sq_error_df(forecasts_df_og, actual_df_og):
    '''Method that returns a dataframe that contains the squared error given a data

    # Copy the original dataframes
    forecasts_df = copy.copy(forecasts_df_og)
    actual_df = copy.copy(actual_df_og)

    # Calculate and return the error dataframe
    errors = (forecasts_df.drop(labels=df.date_col, axis = 1) - actual_df.drop(labe
    errors['date'] = forecasts_df['date']
    errors.date_col = 'date'
    return errors

forecasts_df = get_rw_forecasts(df, dns_maturities, horizon)
errors = get_sq_error_df(forecasts_df, df[horizon:])
```

In [ ]:
```
starting_year = 2006

# IDEA: ADD RECESSIONS AS GREY BANDS

def error_plot(errors, maturities, horizon, legendloc = 'upper left', base_title='S
    '''Method that plots the erors over time of a dataframe that contains the error
```

```
    # Plot each maturity separately and assign labels for the legend
    for maturity in maturities:
        plt.plot(errors['date'], errors[maturity], label=f'Maturity {maturity}')

    # Set background color
    plt.gca().set_facecolor('#F7F7F7')

    # Add grid lines
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    # Add legend
    plt.legend(loc=legendloc)

    # Add titles
    plt.title(f'{base_title}, {horizon} days ahead')
    plt.xlabel('Date')
    plt.ylabel(y_label)

    plt.show()

error_plot(errors.iloc[errors.get_year_start_index(starting_year):], ['1.5', '5.0',
```



Nelson-Siegel

Methods

In [ ]:
```python
# Later add these functions to their own .py file or the the

def ns_slope(lam, tau):
    '''Method that return the slope factor loading of the Nelson-Siegel model'''

    return (1-exp(-lam*tau)) / (lam*tau)

def ns_curve(lam, tau):
    '''Method that return the curve factor loading of the Nelson-Siegel model'''

    return (1-exp(-lam*tau)) / (lam*tau) - exp(-lam*tau)

def ns_yield(beta1, beta2, beta3, lam, tau):
    '''Method that return the yield of the Nelson-Siegel model'''

    level = 1
    slope = ns_slope(lam, tau)
    curve = ns_curve(lam, tau)

    return beta1 * level + beta2 * slope + beta3 * curve

def get_B(lam, maturities):
    '''Method that return the factor loading matrix of the Nelson-Siegel model '''

    matrix = None

    for tau in maturities:
        tau = float(tau)

        row = np.array([1, ns_slope(lam, tau), ns_curve(lam, tau)])

        if matrix is None:
            matrix = row
        else:
            matrix = np.vstack([matrix, row])

    return matrix

# Could definetely be added to the MaturitiesDf class too
def obtain_betas(df: MaturitiesDf, lam: float):
    '''Method to obtain the betas for a MaturitiesDf given a value of lambda'''

    # Initialize the dataframe that is returned
    betas_df = pd.DataFrame(columns = ['beta1', 'beta2', 'beta3'])

    # Loop over all times
    for t in range(len(df)):

        # Calculate the betas
        y = df.drop(df.date_col, axis = 1).iloc[t]
        B = get_B(lam, df.maturities_from(0.5))
        betas_row = np.linalg.inv(B.T @ B) @ B.T @ y

        # Add the betas to the datarame
        row = {'beta1': betas_row[0], 'beta2': betas_row[1], 'beta3': betas_row[2]}
```

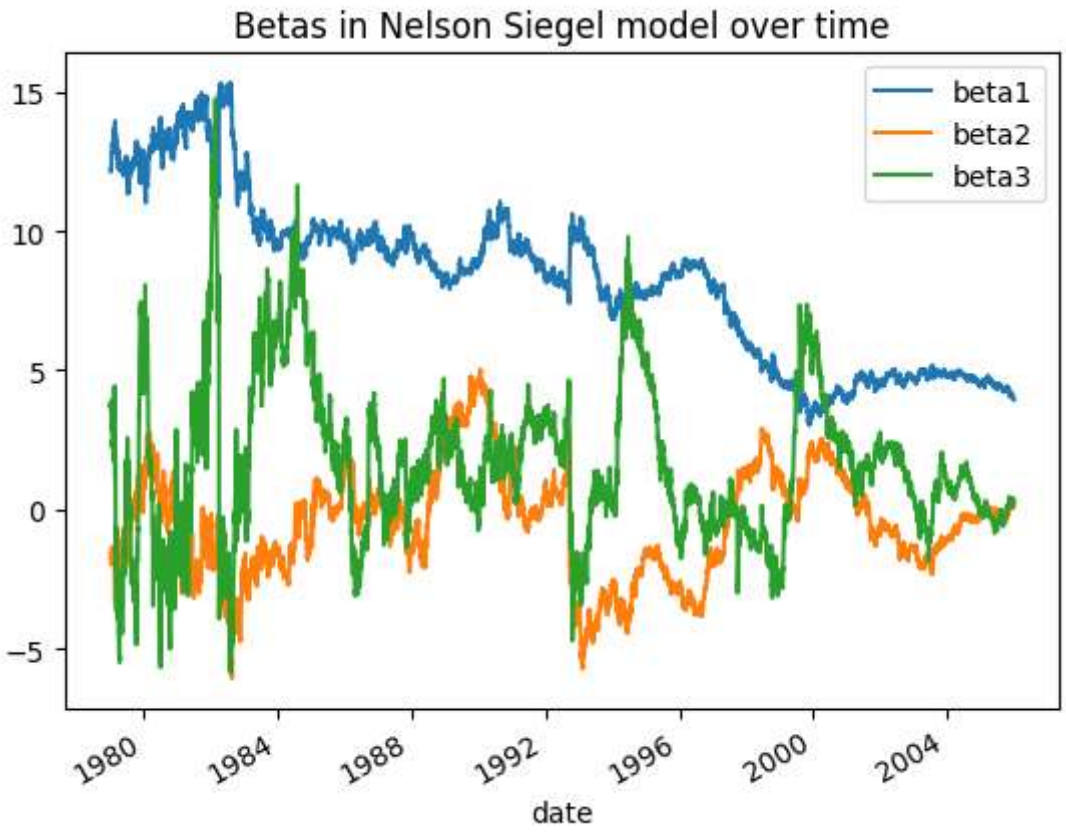
```
        betas_df.loc[t] = row

    betas_df['date'] = df['date'].values
    return betas_df

def obtain_ns_forecasts(df: MaturitiesDf, betas, lam):

    # Copy dataframe
    df_ = copy.copy(df)

    # Loop over all the times
    for t in range(len(df_)):

        beta1 = betas.iloc[t]['beta1']
        beta2 = betas.iloc[t]['beta2']
        beta3 = betas.iloc[t]['beta3']

        # Loop over all maturities
        for tau in df_.maturities_from(0.5):

            # Calculate the forecast
            df_.loc[t, tau] = ns_yield(beta1, beta2, beta3, lam, float(tau))

    df_['date'] = df['date']

    return df_
```

Find betas

In [ ]:
```
# Create the dataframe that we will predict

df_dns = df[[df.date_col] + dns_maturities]
df_dns.reset_index(drop = True, inplace = True)

# Split in in-sample and out-of-sample
df_dns_is = df_dns[:df_dns.get_year_end_index(2005)]
df_dns_oos = df_dns[df_dns.get_year_start_index(2006):]
df_dns_oos.reset_index(drop = True, inplace = True)

lambda_ = 1 / 50
```

In [ ]:
```
# Obtain the betas
betas = obtain_betas(df_dns_is, lambda_)
betas_oos = obtain_betas(df_dns_oos, lambda_)
```

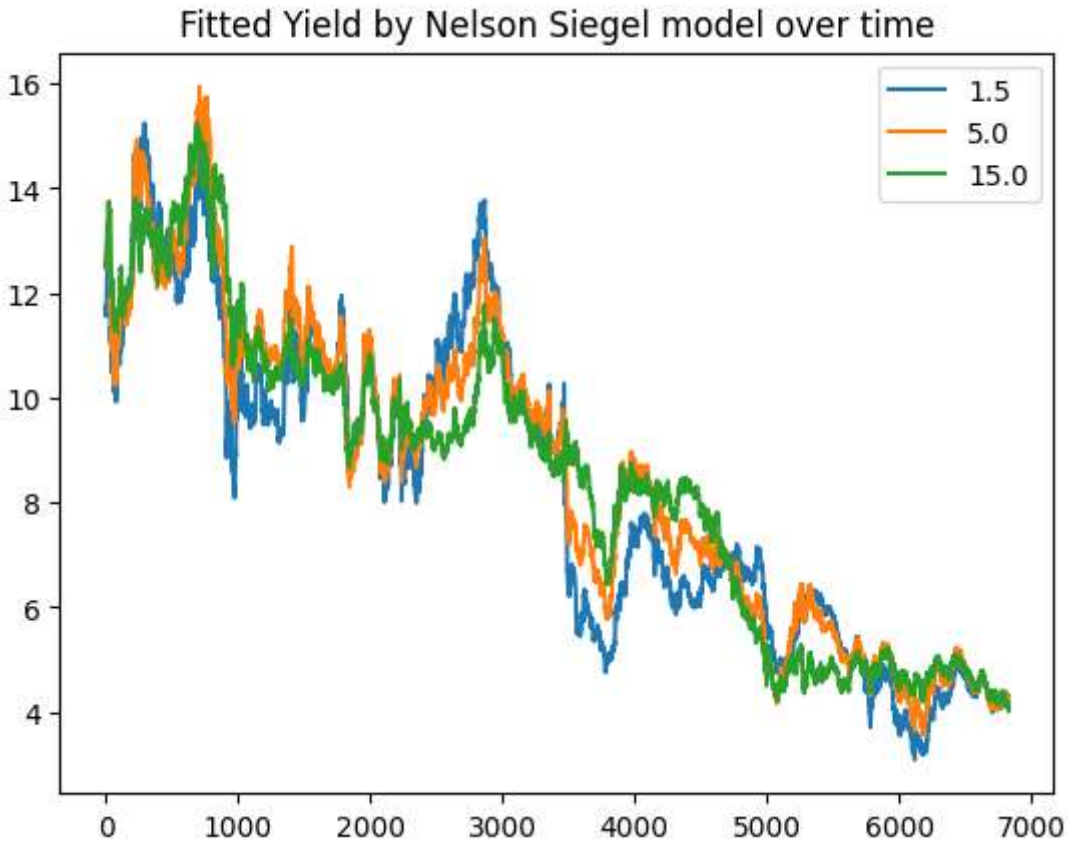
In [ ]:
```
# Plot of the betas over time
betas.plot(x='date', y=['beta1', 'beta2', 'beta3'],
        legend = True,
        title = 'Betas in Nelson Siegel model over time');
```



Fit in-sample using the betas

In [ ]:
```
dns_f_df = obtain_ns_forecasts(df_dns_is, betas, lambda_)
```

In [ ]:
```
maturities_to_plot = ['1.5', '5.0', '15.0']
dns_f_df.plot(x='date', y=maturities_to_plot,
        legend = True,
        title = 'Fitted Yield by Nelson Siegel model over time');
```



Analyze the beta time series

In [ ]:
```
model = VAR(betas.drop('date', axis = 1).dropna())
```

In [ ]:
```
fit = model.fit(1)
fit.summary()
```

In [ ]:
```
def get_ns_forecasted_betas(betas_is, betas_oos, horizon, p):

    available_betas = copy.copy(betas_is)
    forecasted_betas = pd.DataFrame(columns=betas_is.columns)

    # Loop over the out-of-sample period until before the horizon
    for index in betas_oos.index[:-horizon]:

        # Predicting for
        date = betas_oos.iloc[index+horizon]['date']

        # Fit VAR model
        data = available_betas.drop('date', axis = 1).dropna().to_numpy()
        model = VAR(data)
        fit = model.fit(p)

        # Add forecasted row to dataframe with date
        f = fit.forecast(data, horizon)[-1]
        frow = {'date': date, 'beta1': f[0], 'beta2': f[1], 'beta3': f[2]}
        forecasted_betas.loc[index] = frow
```

```
        # Add row to available betas
        available_betas = pd.concat([available_betas, betas_oos.iloc[[index]]])

    return forecasted_betas
```

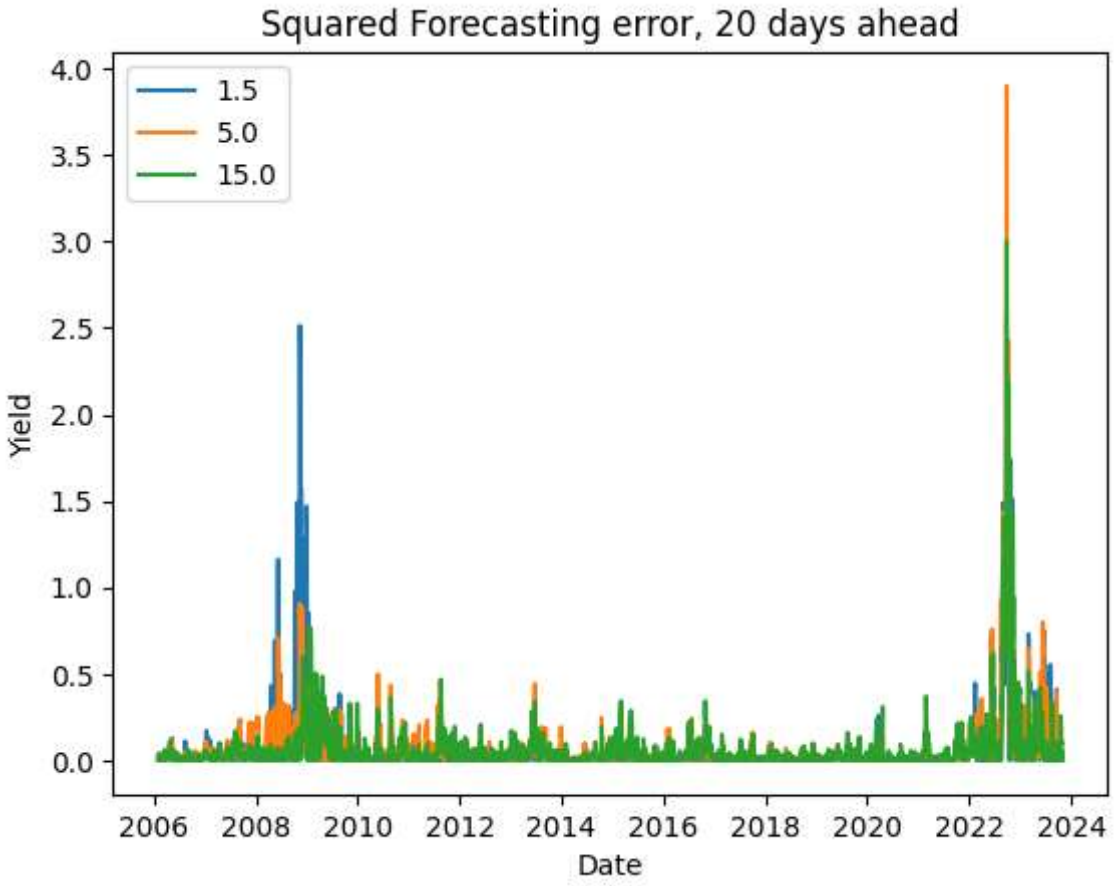
In [ ]:
```
# Get dataframe of predicted betas
horizon = 20
p = 1
forecasted_betas = get_ns_forecasted_betas(betas, betas_oos, horizon, p)
```

In [ ]:
```
forecasted_yields = obtain_ns_forecasts(df_dns_oos[horizon:], forecasted_betas, lam
```

Put together in single method

In [ ]:
```
def get_dns_forecasts(df, maturities, split_index, horizon, lambda_ = 1/50, p = 1):
    '''Method that obtains Dynamic-Nelson-Siegel forecasts for given maturites, spl

    # Create the dataframe that we will predict
    df_dns = df[[df.date_col] + maturities]
    df_dns.reset_index(drop = True, inplace = True)

    # Split in in-sample and out-of-sample
    df_dns_is = df_dns[:split_index]
    df_dns_oos = df_dns[split_index:]
    df_dns_oos.reset_index(drop = True, inplace = True)

    # Obtain the betas
    betas = obtain_betas(df_dns_is, lambda_)
    betas_oos = obtain_betas(df_dns_oos, lambda_)

    # Get dataframe of predicted betas
    forecasted_betas = get_ns_forecasted_betas(betas, betas_oos, horizon, p)

    # Get forecasted yields
    forecasted_yields = obtain_ns_forecasts(df_dns_oos[horizon:], forecasted_betas,

    return forecasted_yields
```

In [ ]:
```
errors_dns_oos = get_sq_error_df(forecasted_yields, df_dns_oos[horizon:])

error_plot(errors_dns_oos, ['1.5', '5.0', '15.0'], 20)
```



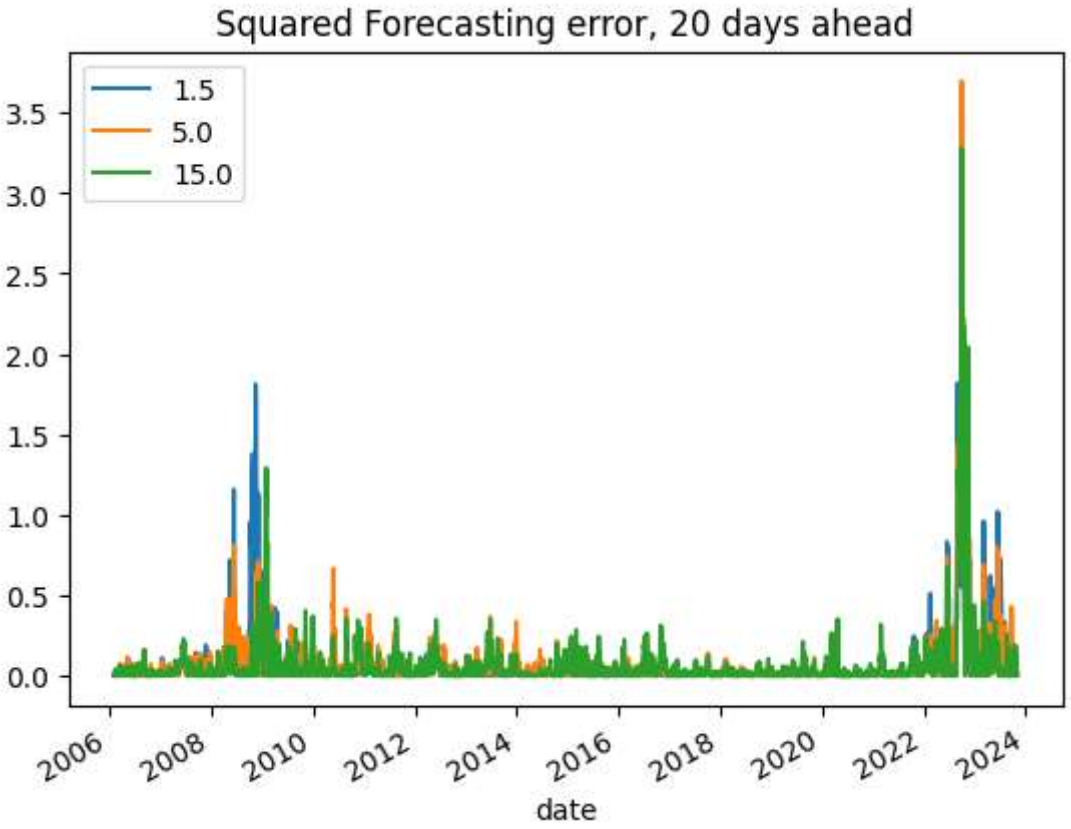
VAR

In [ ]:
```python
def get_var_forecasts(df: MaturitiesDf, maturities, split_index, p, horizon):

    # Define in and out-of-sample dataframes
    df_is = df[maturities].iloc[:split_index].dropna().to_numpy()
    df_oos = df[maturities].iloc[split_index:].dropna().to_numpy()

    available_yields = copy.copy(df_is)
    forecasted_yields = pd.DataFrame(columns= ['date'] + maturities)

    # Loop over the out-of-sample period until before the horizon
    for index in betas_oos.index[:-horizon]:

        # Predicting for
        date = df.iloc[split_index + index + horizon]['date']

        # Fit VAR model
        model = VAR(available_yields)
        fit = model.fit(p)

        # Add forecasted row to dataframe with date
        f = fit.forecast(available_yields, horizon)[-1] # Extract last element

        # Add forecasted yields and date to the row
        frow = {tau: f[i] for i, tau in enumerate(maturities)}
        frow['date'] = date
```

```
        forecasted_yields.loc[index] = frow

        # Add row to available yields
        available_yields = np.vstack([available_yields, df_oos[index]])

    return forecasted_yields
```

In [ ]:
```
p = 1
horizon = 20
split_index = df.get_year_end_index(2005)

var_forecasted_yields = get_var_forecasts(df, dns_maturities, split_index, p=p, hor
```

In [ ]:
```
errors_var_oos = get_sq_error_df(var_forecasted_yields, df_dns_oos[horizon:])
error_plot(errors_var_oos, ['1.5', '5.0', '15.0'], 20)
```



All three at once

Handy function and class

In [ ]:
```
def get_multiple_forecasts(df, maturities_to_use, split_index, horizons, models, p_
    '''Method that can obtain forecasts for multiple models and horizons at once an

    if returning_dict is None:
        returning_dict = dict()

    # Loop over all horizons
```

```
    for h in horizons:

        try:
            horizon_dict = returning_dict[h]
        except:
            horizon_dict = dict()

        # Loop over all models
        for model in models:

            if model == 'RW':
                yields_f = get_rw_forecasts(df, maturities_to_use, h).iloc[split_in

            elif model == 'DNS':
                yields_f = get_dns_forecasts(df, maturities_to_use, split_index, h,

            elif model == 'VAR':
                model = f'VAR_{p_var}'
                yields_f = get_var_forecasts(df, maturities_to_use, split_index, p_

            # Add forecasted yields to horizon dict
            horizon_dict[model] = yields_f

        # Add horizon dict to returning dict
        returning_dict[h] = horizon_dict

    return returning_dict

class ForecastsDict:
    '''Class that contains a dictionary of forecasted yields'''

    def __init__(self, dict, actual_df):
        self.dict = copy.copy(dict)
        self.actual_df = copy.copy(actual_df)
        self.horizons = list(dict.keys())
        self.models= list(dict[list(self.horizons)[0]].keys())
        self.mse_df = self.get_mse_df()

    def get_mse_df(self):
        '''Method that obtain a dataframe of the MSEs'''
        mse_df = dict()

        # Loop over the horizons
        for h in self.horizons:
            h_dict = dict()

            # Loop over the models
            for model in self.models:

                # Get the error dataframe
                yields_f = copy.copy(self.dict[h][model])
                actual_df = copy.copy(self.actual_df)
                yields_f.date_col = actual_df.date_col
                error_df = get_sq_error_df(yields_f, actual_df[horizon:]) #

                # Score the squared error means in the dictionary
```

```
                sq_error_means = error_df.drop('date', axis=1).mean(axis=0)
                h_dict[model] = sq_error_means

            mse_df[h] = h_dict

        return mse_df

    def plot_multiple(self, maturities_to_plot, horizons = None, models = None, leg
        '''Method that can plot multiple maturities'''

        if horizons is None:
            horizons = self.horizons

        if models is None:
            models = self.models

        # Loop over all horizons
        for h in horizons:

            # Loop over all forecasted yields
            for model, yields_f in self.dict[h].items():

                if model not in models:
                    continue

                # Print where we are at
                print(f'Horizon: {h}, Model: {model}')

                # Plot the errors
                actual_df = copy.copy(self.actual_df)
                errors = get_sq_error_df(yields_f, actual_df[h:])
                error_plot(errors, maturities_to_plot, h, legendloc=legendloc, base

    def plot_relative_errors(self, model1, model2, maturities_to_plot):
        '''Method that can plot the relative erorrs of two models'''

        actual_df = copy.copy(self.actual_df)
        errors0 = get_sq_error_df(self.dict[horizon][model1], actual_df[horizon:])
        errors1 = get_sq_error_df(self.dict[horizon][model2], actual_df[horizon:])

        relative_errors = errors0.drop('date', axis=1) - (errors1.drop('date', axis
        relative_errors['date'] = errors0['date']

        error_plot(relative_errors, maturities_to_plot, horizon, 'upper right', bas

    def plot_mses(self, horizon, models = None):
        '''Method that can plot the MSEs of a horizon'''

        # If no models given, just use them all
        if models is None:
            models = self.models

        # Create an empty DataFrame to store the results
        result_df = pd.DataFrame(index=self.mse_df[horizon]['RW'].index)

        # Populate the DataFrame with MSE values for each model
```

```
        for model in models:
            result_df[model] = self.mse_df[horizon][model]

        # Plot the MSE values for each model
        for model in models:
            plt.scatter(result_df.index, result_df[model], label=model, marker='D')

        # Add legend
        plt.legend()

        # Add titles and labels
        plt.title(f'Mean Squared Error for Horizon {horizon}')
        plt.xlabel('Maturity')
        plt.ylabel('Mean Squared Error')

        # Add grid lines
        plt.grid(axis='y', linestyle='--', alpha=0.7)

        # Set background color
        plt.gca().set_facecolor('#F7F7F7')

        # Customize x-axis ticks
        plt.xticks(rotation=45, ha='right')  # Rotate ticks for better readability
        plt.xticks(result_df.index[::4])  # Adjust the number of ticks

        # Display the plot
        plt.tight_layout()

        # Show the plot
        plt.show()

    def get_rmse_table(self, models = None, horizons=None):
        '''Method that obtain a table of RMSEs for multiple models and horizons'''

        if horizons is None:
            horizons = self.horizons

        if models is None:
            models = self.models

        # Initialize returning dataframe
        rmse_df = pd.DataFrame(columns=models)
        rmse_df['horizon'] = horizons
        rmse_df.set_index('horizon', inplace=True)

        # Loop over the horizons
        for h in horizons:

            # Create row that is added
            row = {model: sqrt(self.mse_df[h][model].mean()) for model in models}

            # Add row
            rmse_df.loc[h] = row

        # Return dataframe
        return rmse_df
```

Obtain and save the forecasts

In [ ]:
```
horizons = [20, 60, 125, 250]
models = ['VAR']#['RW', 'DNS', 'VAR']
split_index = df.get_year_end_index(2005)

forecasts_dict = get_multiple_forecasts(df, dns_maturities, split_index, horizons,
```

In [ ]:
```
# Saved forecasts dict
import pickle
# with open('forecasted_dict_0.3.pickle', 'wb') as f:
#     pickle.dump(forecasts_dict, f)

with open('forecasted_dict_0.2.pickle', 'rb') as f:
    LOADED_OBJECT = pickle.load(f)
```

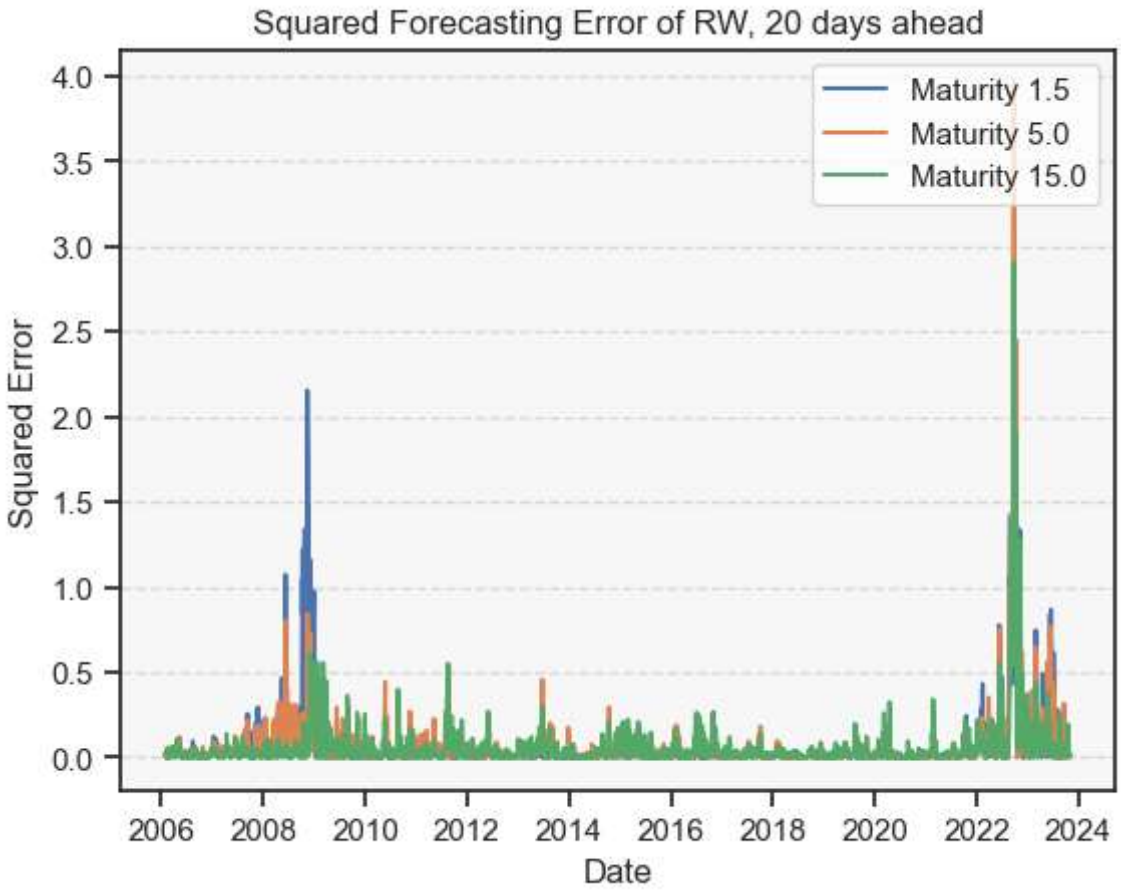
In [ ]:
```
# Set up ForecastDf class instance
#forecasts_dict = copy.copy(LOADED_OBJECT)
forecasts_fd = ForecastsDict(forecasts_dict, df_dns_oos)
```

In [ ]:
```
forecasts_fd_backup = copy.copy(forecasts_fd)
```

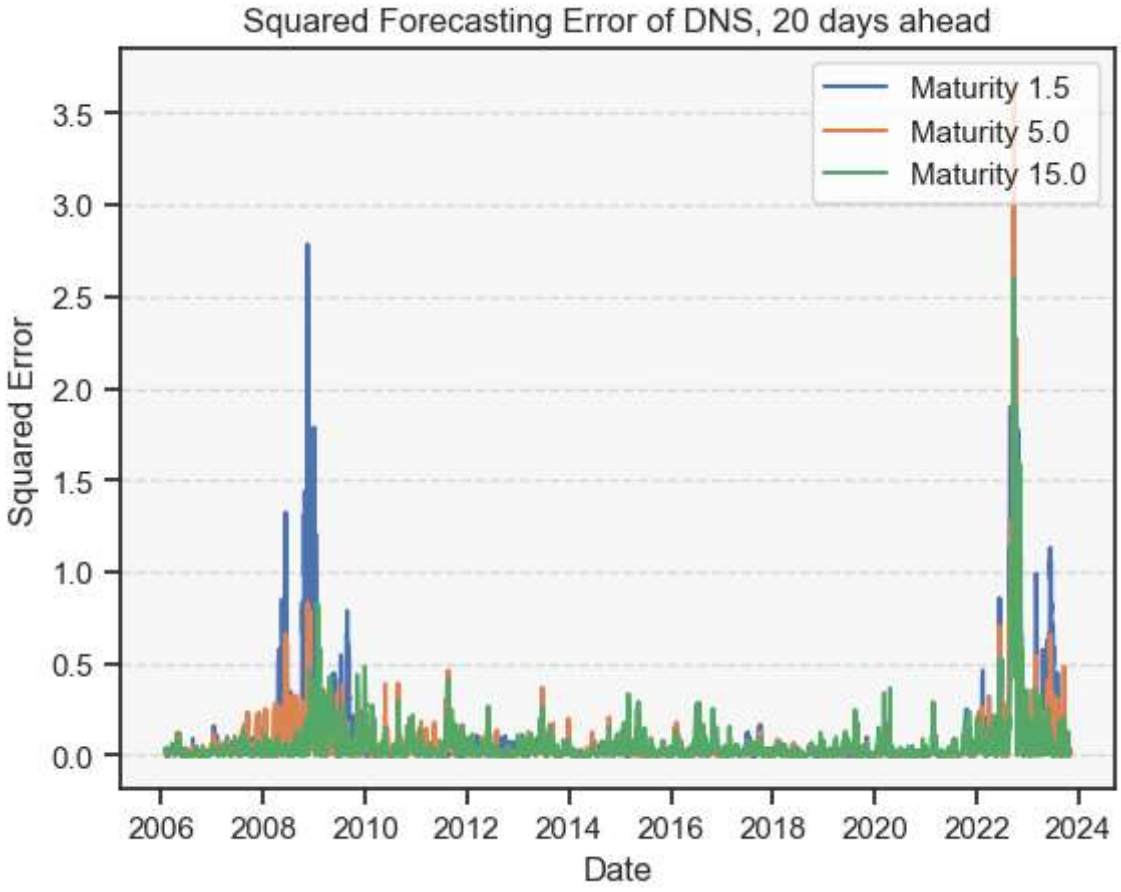
Multiple plots at once

In [ ]:
```
maturities_to_plot = ['1.5', '5.0', '15.0']
forecasts_fd.plot_multiple(maturities_to_plot)
```

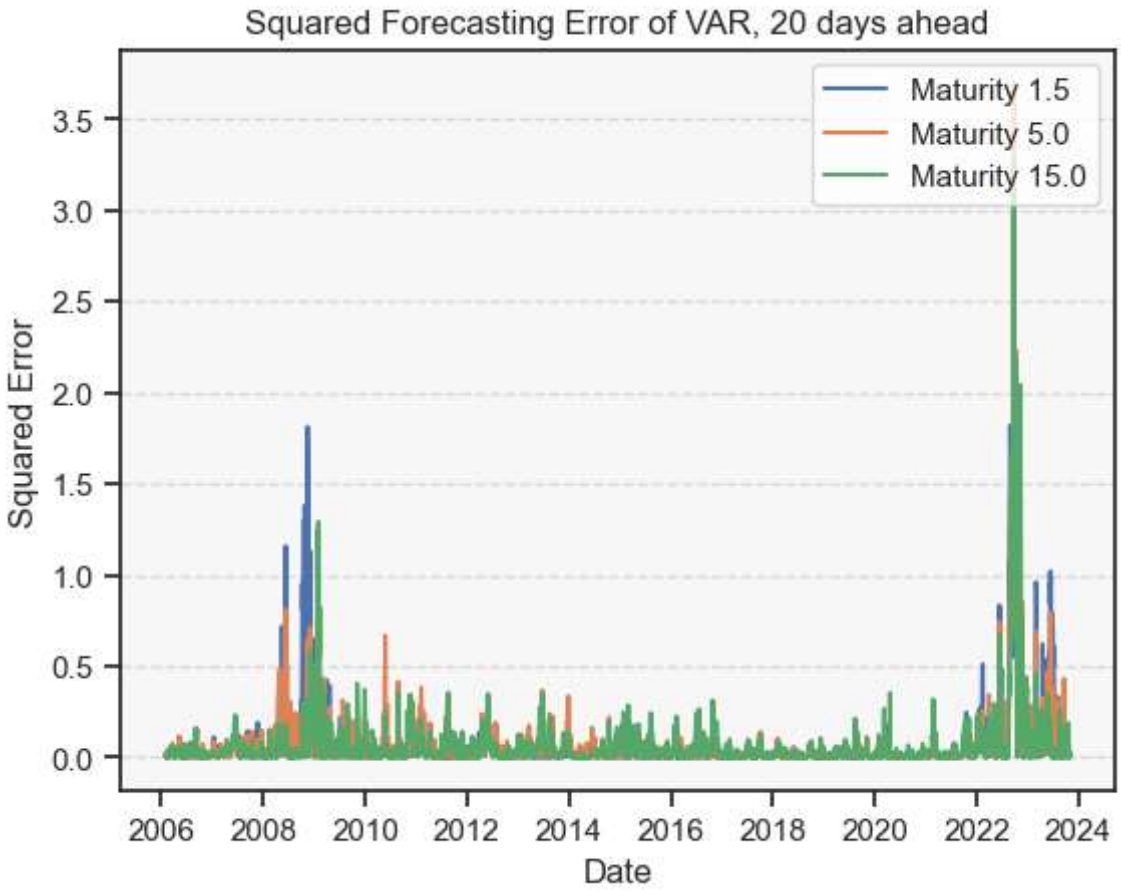
```
Horizon: 20, Model: RW
```



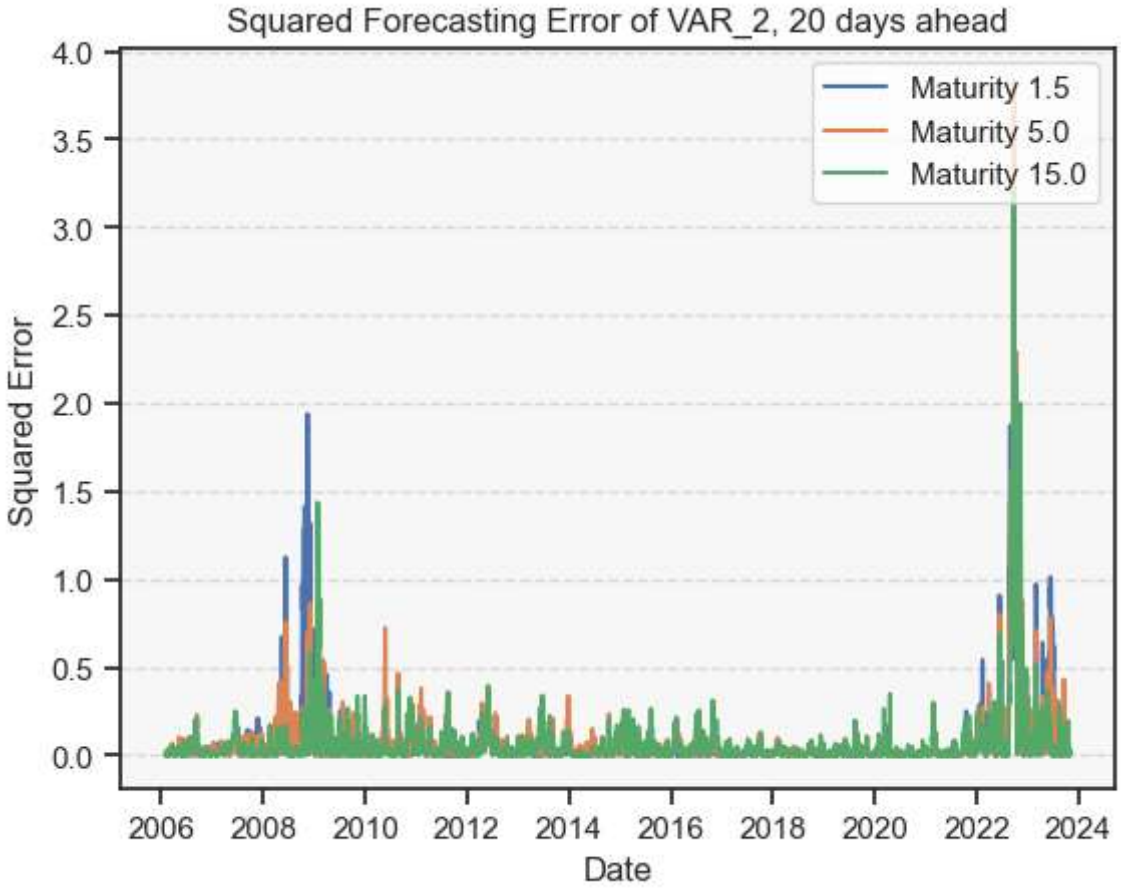

```
Horizon: 20, Model: DNS
```



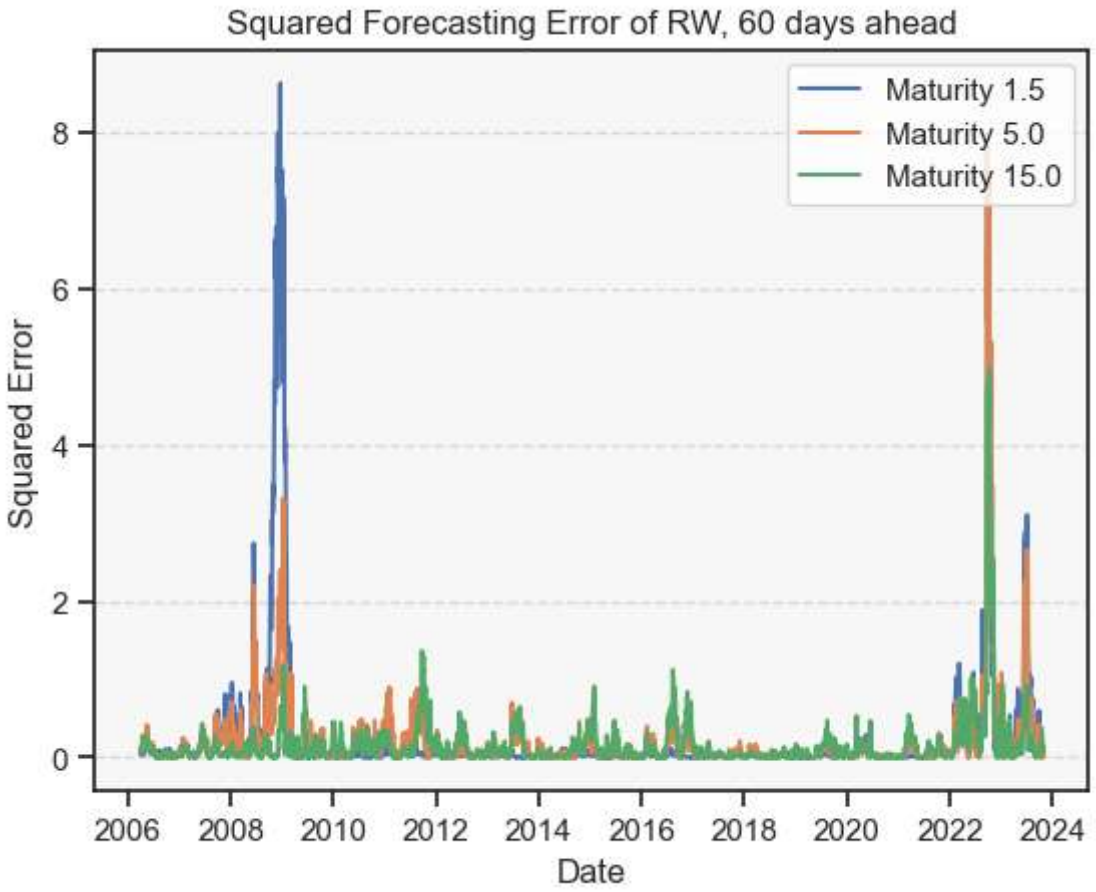

```
Horizon: 20, Model: VAR
```



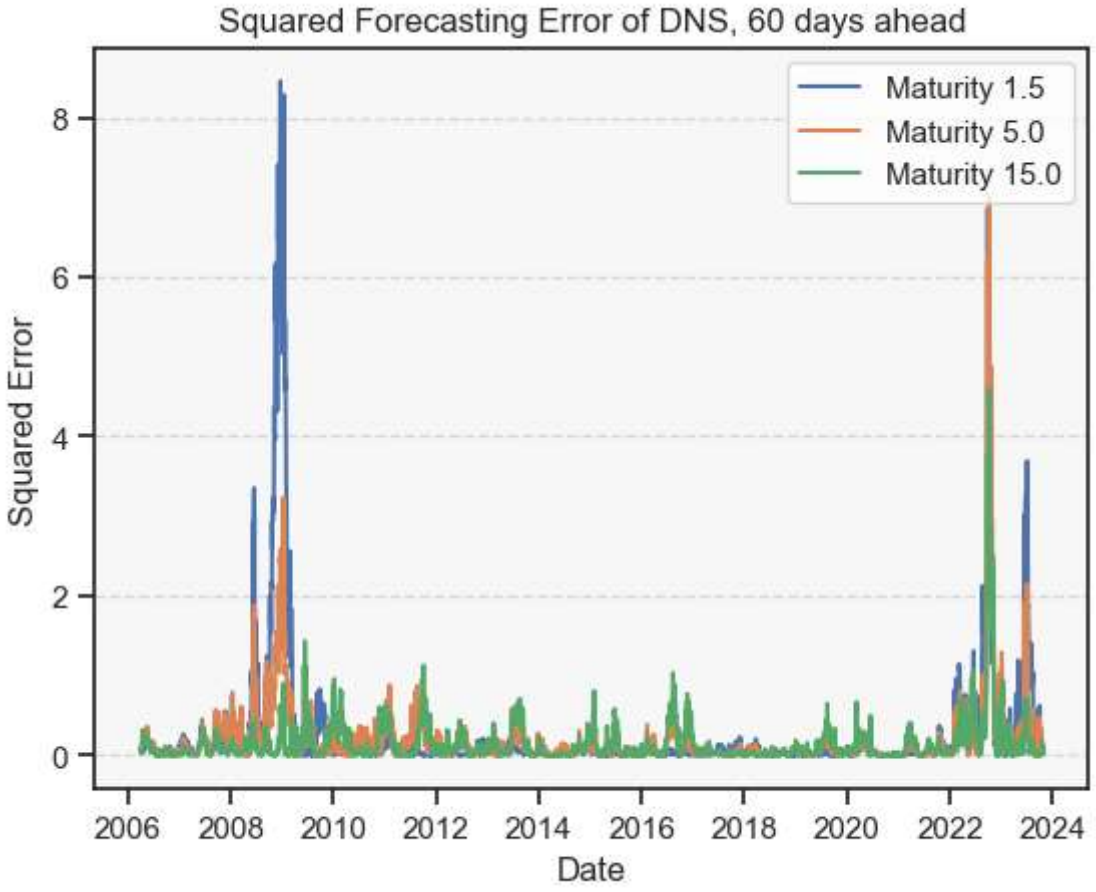

```
Horizon: 20, Model: VAR_2
```



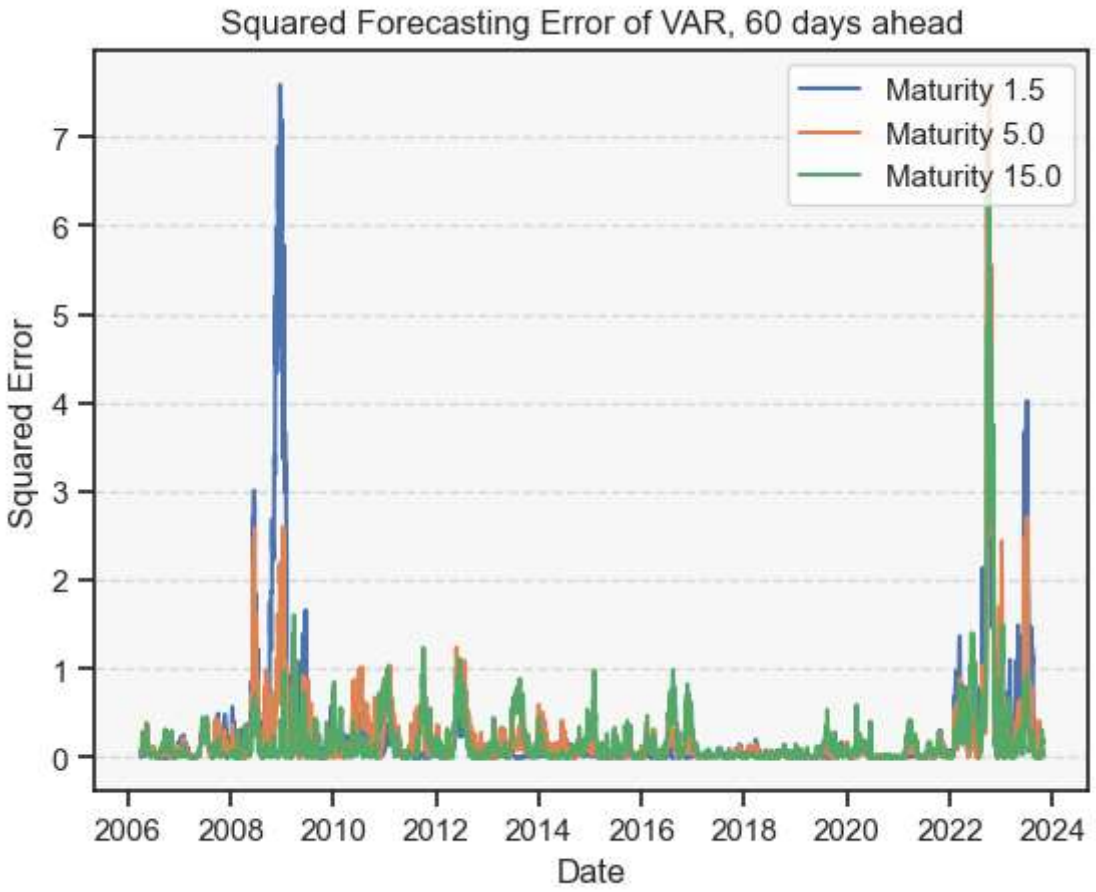

```
Horizon: 60, Model: RW
```



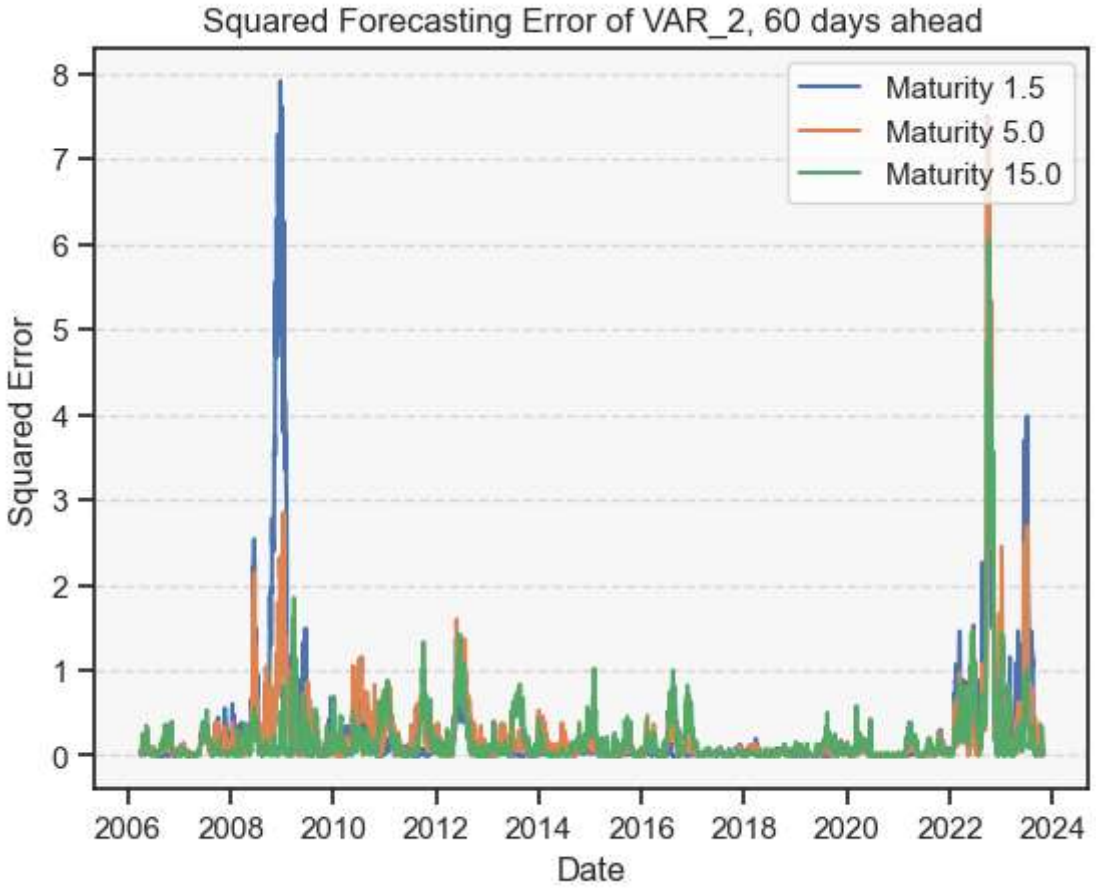

```
Horizon: 60, Model: DNS
```



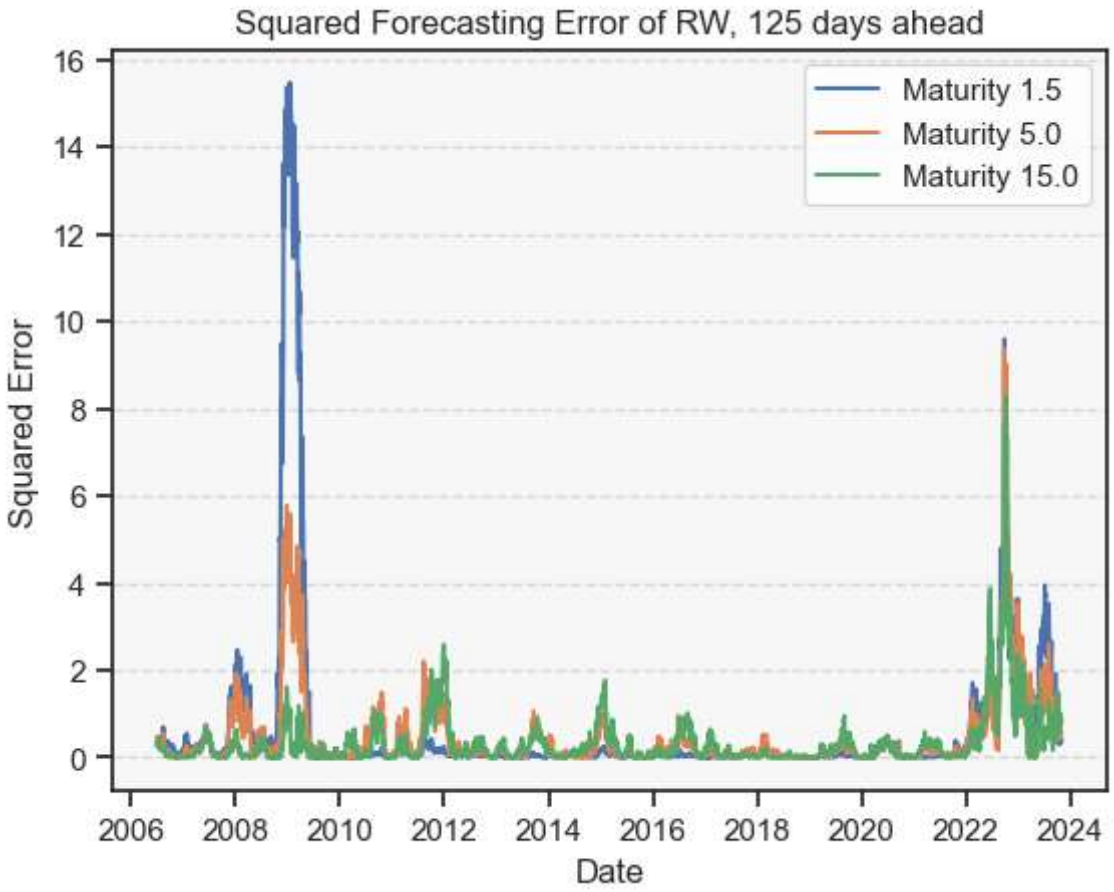

```
Horizon: 60, Model: VAR
```



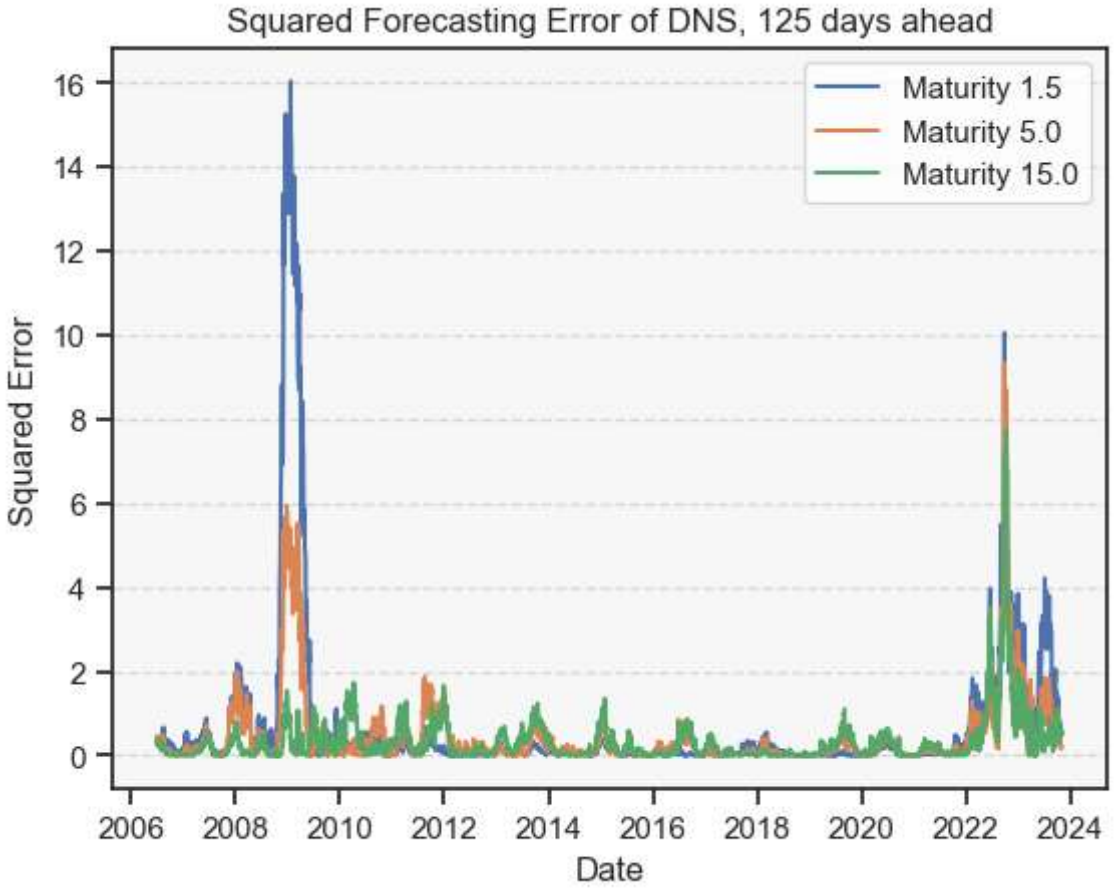

```
Horizon: 60, Model: VAR_2
```



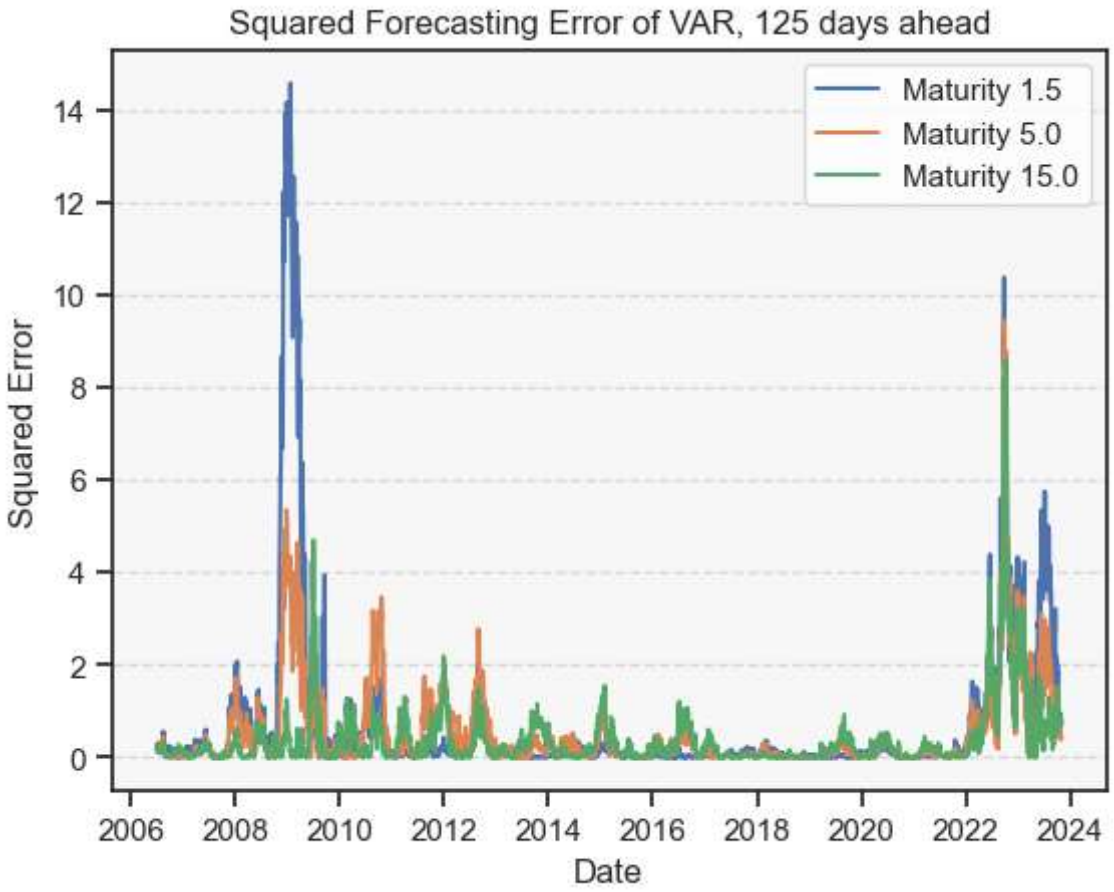

```
Horizon: 125, Model: RW
```



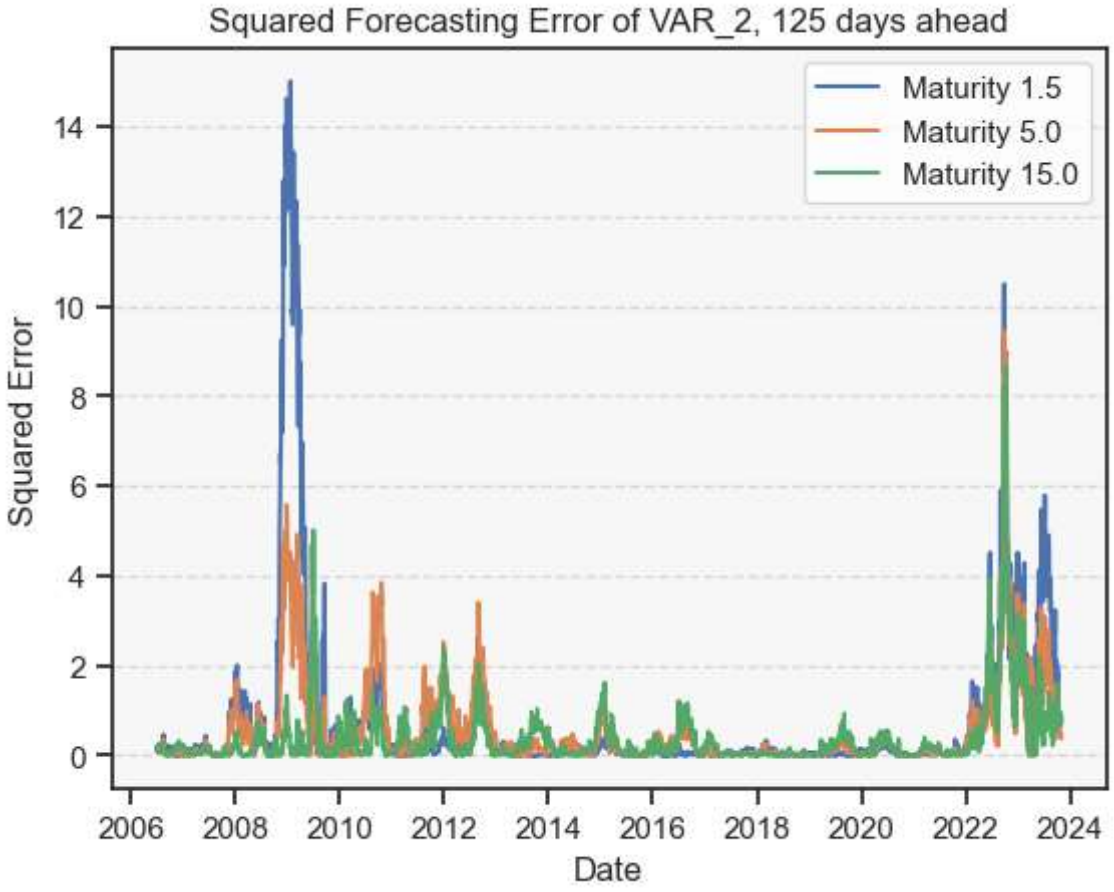

```
Horizon: 125, Model: DNS
```



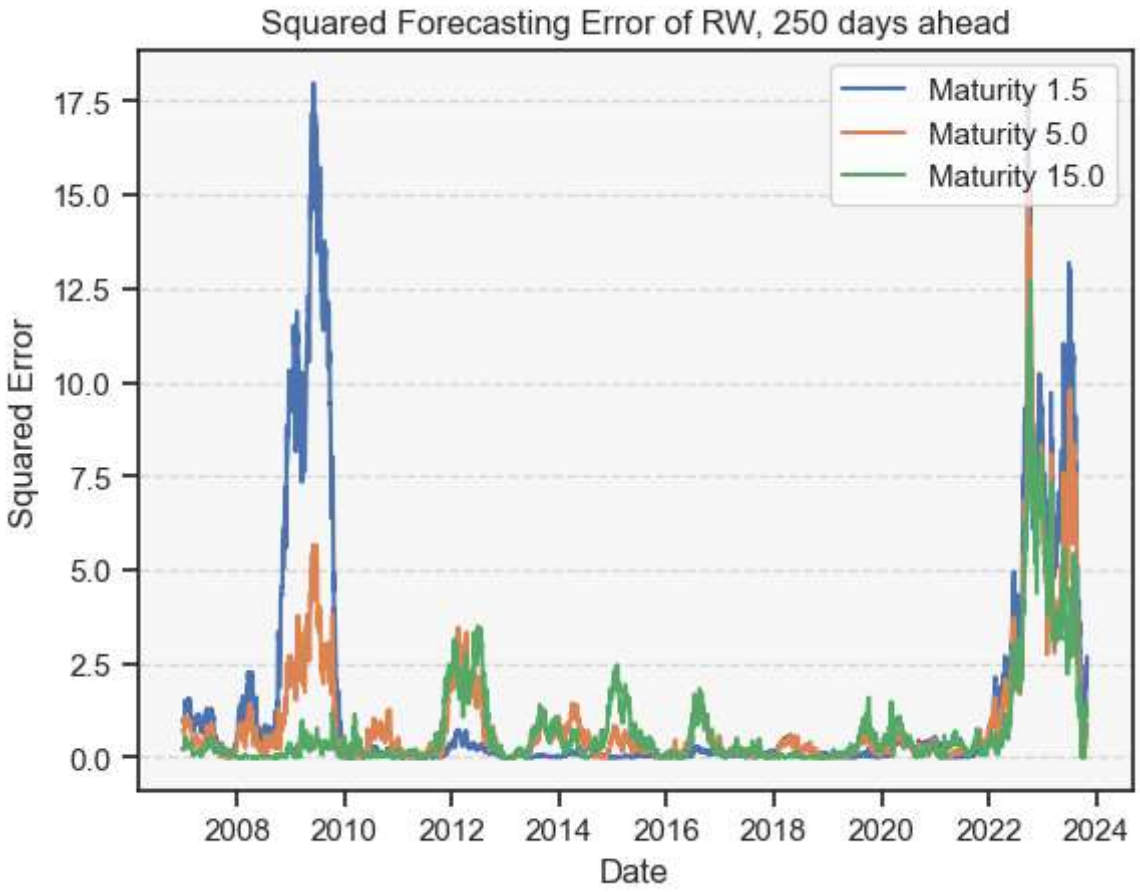

```
Horizon: 125, Model: VAR
```



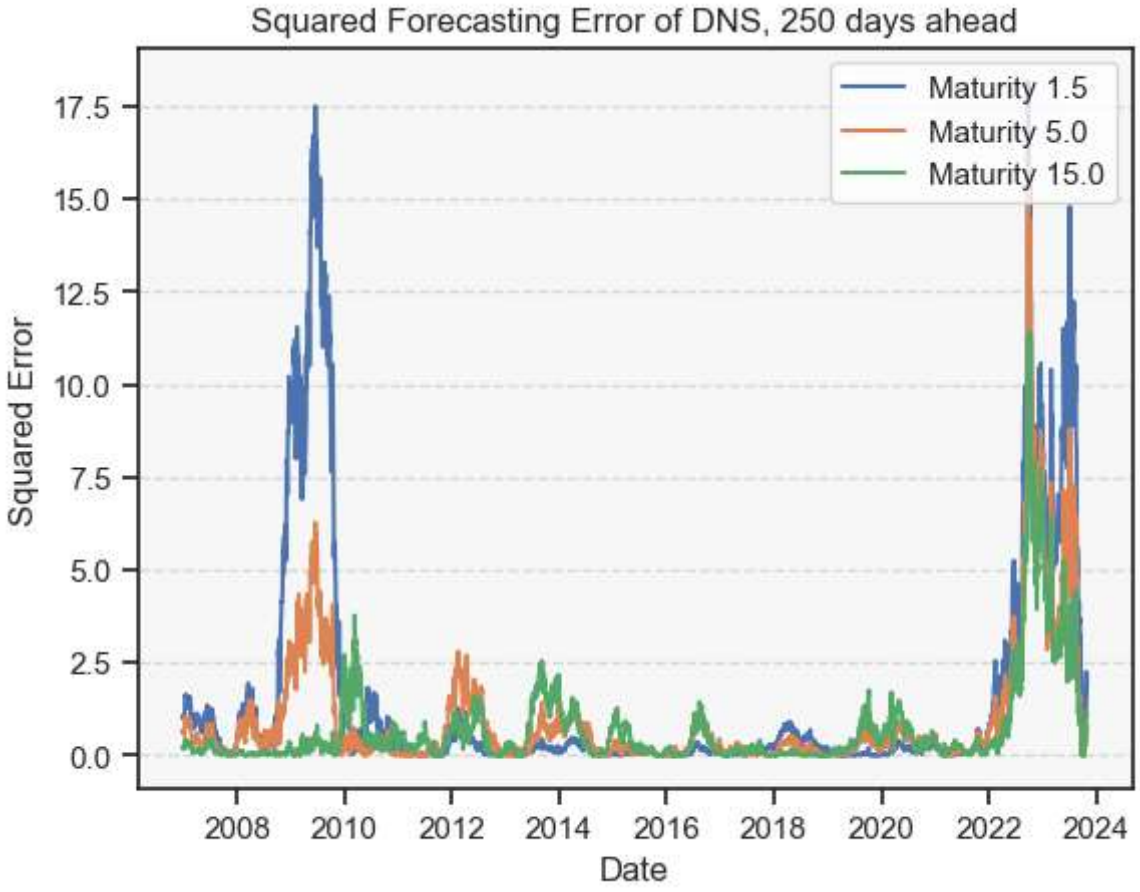

```
Horizon: 125, Model: VAR_2
```



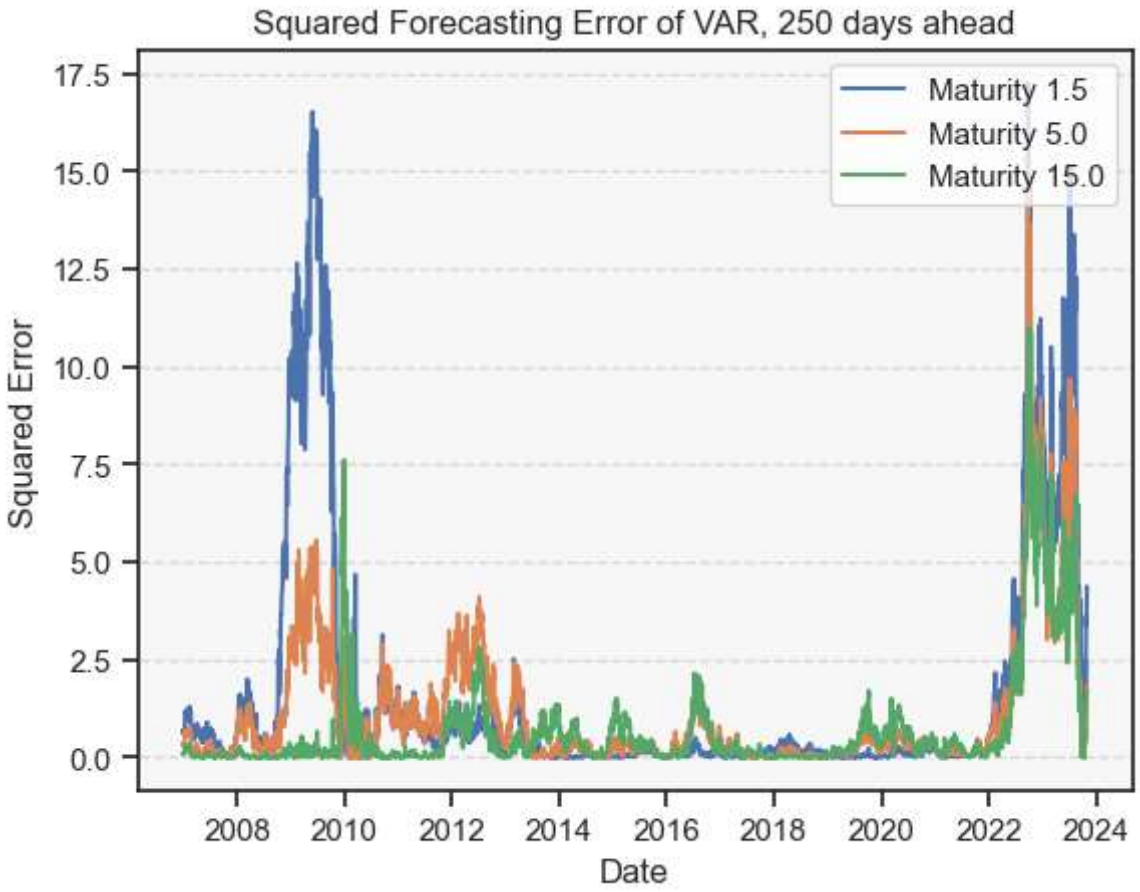

```
Horizon: 250, Model: RW
```



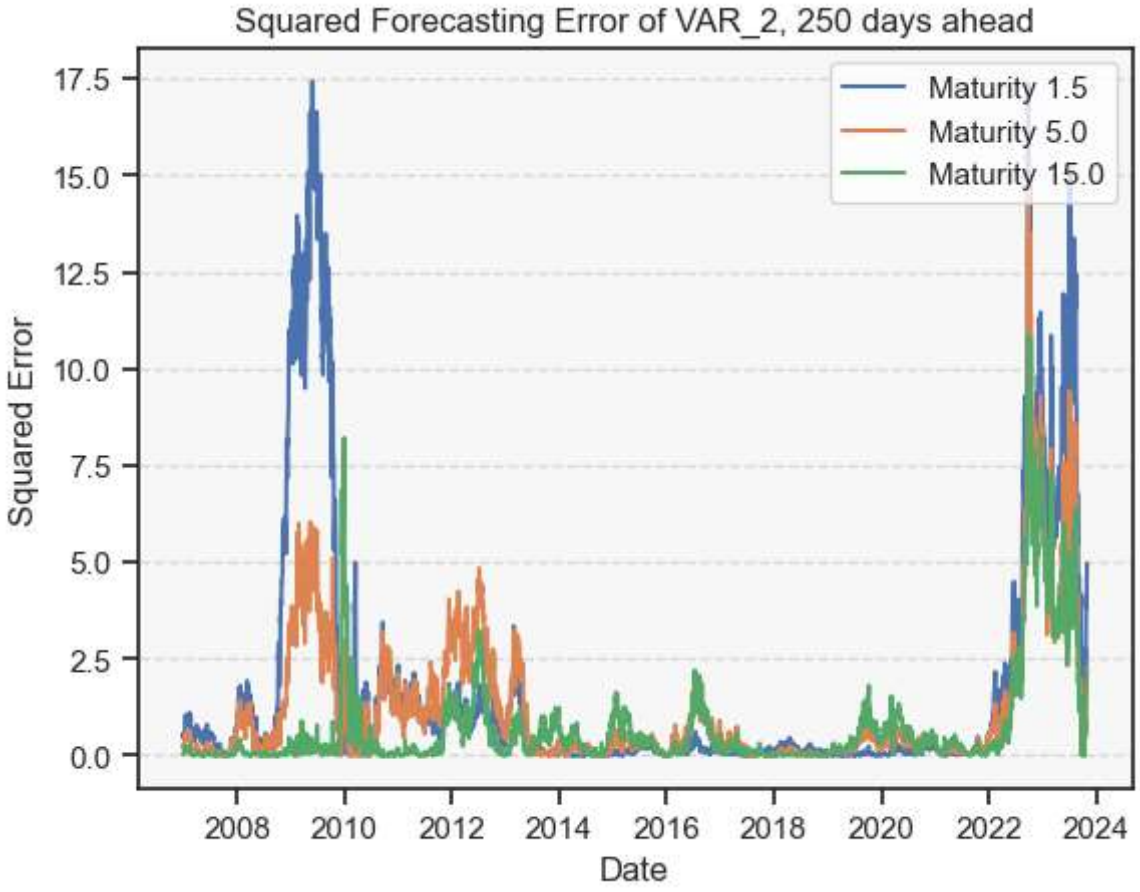

```
Horizon: 250, Model: DNS
```




```
Horizon: 250, Model: VAR
```

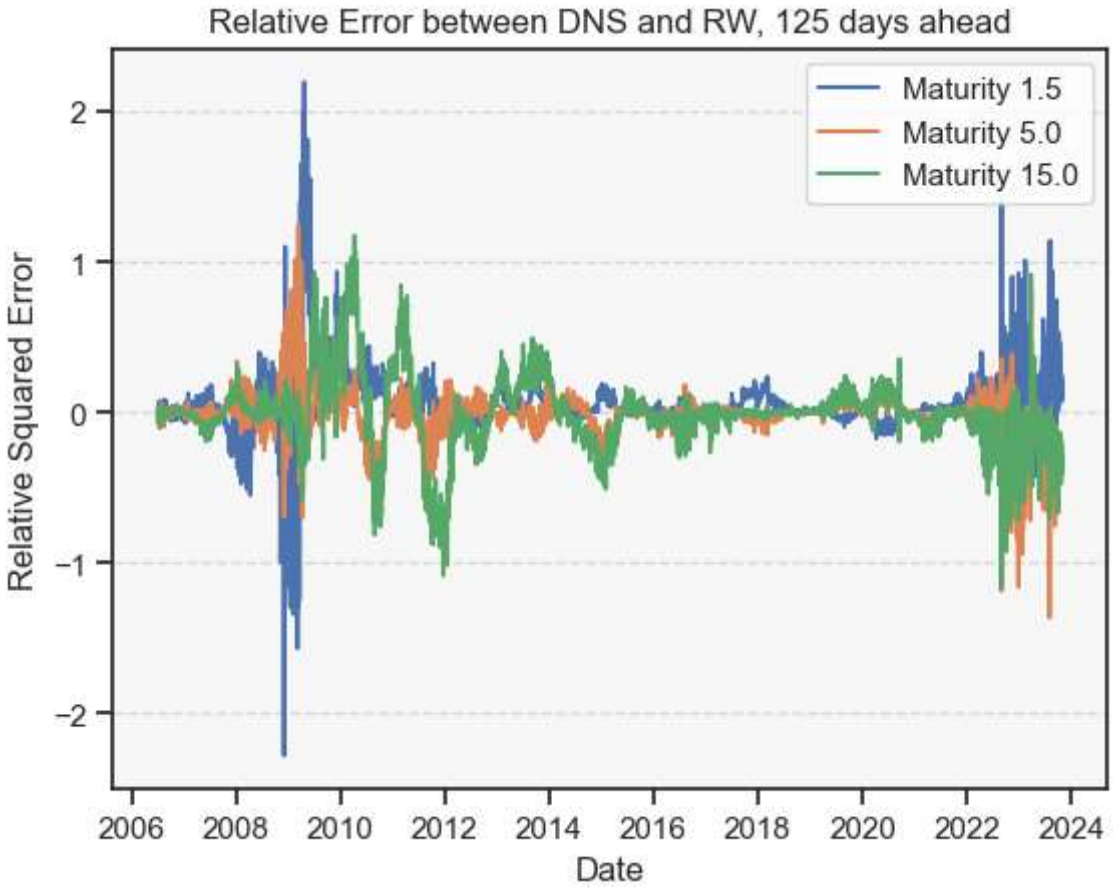



```
Horizon: 250, Model: VAR_2
```



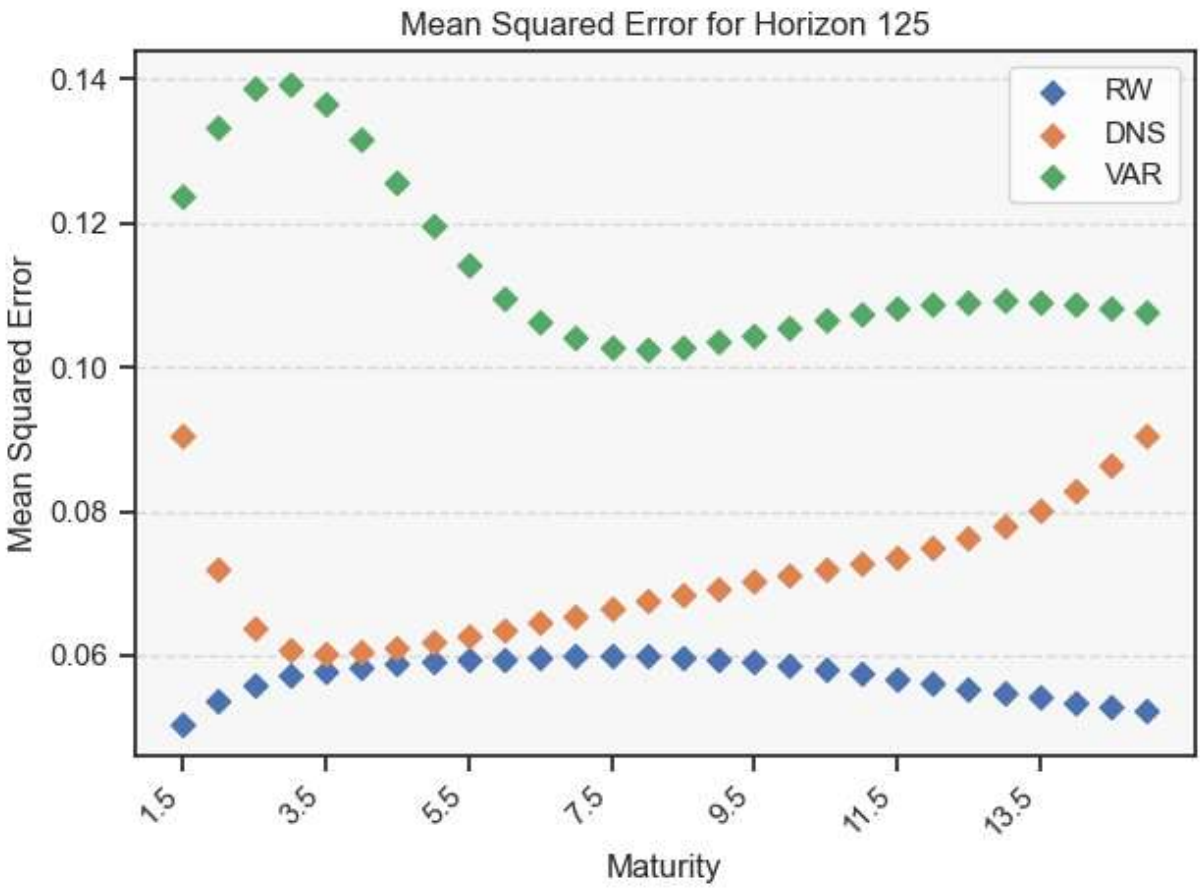
Relative Error plot

In [ ]:
```
horizon = 125
maturities_to_plot = ['1.5', '5.0', '15.0']

forecasts_fd.plot_relative_errors('DNS', 'RW', maturities_to_plot)
```



Plot of MSEs against maturities

In [ ]:
```
horizon = 125
#forecasts_df.actual_df = df_dns_oos
forecasts_fd.plot_mses(horizon, models = ['RW', 'DNS', 'VAR'])
```



RMSE Table

In [ ]:
```
forecasts_fd.get_rmse_table(models = ['RW', 'DNS', 'VAR'])
```

Out[ ]:

horizon	RW	DNS	VAR
20	0.240933	0.247342	0.255634
60	0.241032	0.249937	0.294782
125	0.238933	0.266456	0.33733
250	0.235051	0.322379	0.382427