```python
import scipy
import numpy as np
import pandas as pd
from datetime import datetime
from scipy.stats import norm, halfcauchy, beta, gamma
from statsmodels.tsa.stattools import adfuller

from src.pymc_modelling import get_samp
from itertools import chain
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm

from src.utils import pd_join_freq


def x_get_ytm_newton(f, p, c, T, guess=.05, max_iter=2_000) -> (float, float):
    """
    Newton's method solver for YTM
    :param f: face value
    :param p: price
    :param c: coupon
    :param T: number of future periods
    :param guess: ytm guess
    :param max_iter: maximum iterations
    :return: (ytm, residual)
    """
    get_ytm = lambda y: f * (1 + y) ** (-T) + c * (1 - (1 + y) ** (-T)) / y - p
    ytm = scipy.optimize.newton(get_ytm, guess, maxiter=max_iter)
    return ytm, get_ytm(ytm)


def srs_get_ytm(srs: pd.Series, maturity_date: datetime, coupon: float,
                periodicity: str = 'y', fac_v_scale: float = 1.0, ytm_guess: float = .025) -> np.array:
    """
    Obtain YTM for a pd.Series of prices with datetime index
    :param srs: series of prices with datetime index
    :param maturity_date:
    :param coupon:
    :param periodicity: frequency coupon payments are made in
    :param fac_v_scale: bonds are priced as a percentage of their face value, 100 -> bond trades at par
                       function assumes par trading == 1, if par trading == 100, then scale = 100
    :param ytm_guess:
    :return: pd.Series of YTMs
    """
    dict_period = {'y': 30 * 12, '6m': 30 * 6, 'm': 30}
    assert periodicity in list(dict_period.keys()), f"please specify periodicity as either {list(dict_period.keys())}"
    assert isinstance(srs.index, pd.DatetimeIndex), "please specify datetime index"

    out, residuals = [], []
    for idx, val in pd.DataFrame(srs).iterrows():

        T = int((maturity_date - idx).days / (30 * 12))

        try:
            ymt, resid = x_get_ytm_newton(f=fac_v_scale, p=val.values, c=coupon, T=T, guess=ytm_guess)
        except RuntimeError as e:
            print(fac_v_scale, val.values, coupon, T, ytm_guess, e)
            continue

        out.append(ymt[0])
        residuals.append(resid)

    stat = scipy.stats.describe(residuals)
    print(f"Overall solver residuals: mean {stat.mean}, std: {np.sqrt(stat.variance)}")

    return np.array(out)


def get_aic(arr, fitted_dist):
    """
    Calculates AIC of a fitted distribution
    :param arr:
    :param fitted_dist:
    :return:
    """
    ll = np.log(np.product(fitted_dist.pdf(arr)))
    aic = 2 * len(arr) - 2 * np.prod(ll)
    return aic


def get_fitted_dist(arr: np.array, dists: list) -> object:
    """
    Fits and compares distribution according to AIC
    Currrently supports normal and beta
    :param arr:
    :return: frozen scipy distribution
    """
    dict_dists = {'norm': norm, 'gamma': gamma, 'beta': beta, 'halfcauchy': halfcauchy}
    for dist in dists:
        assert dist in dict_dists.keys(), f"{dist} not specified in function"

    fitted_dists = []
    for dist in dists:
        fitted_dists.append(
            dict_dists[dist](*dict_dists[dist].fit(arr))
        )
    aic = []
    for f_dist in fitted_dists:
        aic.append(get_aic(arr, f_dist))

    # returns dist with min aic
    return fitted_dists[aic.index(min(aic))]


def srs_apply_impute_data(srs: pd.Series) -> pd.Series:
    """
    Imputes missing data as a draw from a fitted distribution on existing data
    :param srs:
    :return:
    """
    arr = srs.dropna().values

    # assert len(arr) / srs.isna().sum() > .02, "Low data imputation basis"
    assert len(arr) > 20, "Few data points for distribution estimate"
    srs.loc[srs.isna()] = get_fitted_dist(arr, dists=['gamma', 'norm']).rvs(srs.isna().sum())
    return srs


def get_homogeneous_shape(srs: pd.Series, target_count, dists: list = ['gamma', 'norm']):
    arr = srs.dropna().values
    if len(arr) < target_count:
        arr = srs.dropna().values
        dist = get_fitted_dist(arr, dists=dists)
        return np.concatenate([arr, dist.rvs(target_count - len(arr))])
    else:
        return arr[get_samp(len(arr), target_count)]


def pd_join_dfs(lst_dfs: list, index_name: str = 'date'):
    df = pd.DataFrame(
        index=pd.date_range(
            start=min([*chain(*[[i.index.min(), i.index.max()] for i in lst_dfs])]),
            end=max([*chain(*[[i.index.min(), i.index.max()] for i in lst_dfs])]),
            freq="D",
        )
    )

    for d in lst_dfs:
        df = df.join(d, how='outer')
        # del d

    df.index.name = index_name
    return df


def pd_groupby(df, cols, agg_freq: str, agg_func: str):
    assert type(cols) == list, "please specify columns as list"
    df = df[cols].copy()

    df[agg_freq] = df.index.to_period(agg_freq)

    if agg_func == 'median':
        df = df.groupby(agg_freq).median()
    elif agg_func == 'mean':
        df = df.groupby(agg_freq).mean()
    elif agg_func == 'last':
        df = df.groupby(agg_freq).last()
    elif "q_" in agg_func:
        q = float(agg_func.split("_")[1]) / 100
        df = df.groupby(agg_freq).apply(lambda x: np.quantile(x.dropna(), q) if len(x.dropna()) > 1 else np.nan)
    elif agg_func == 'q1':
        df = df.groupby(agg_freq).apply(lambda x: np.quantile(x.dropna(), .25) if len(x.dropna()) > 1 else np.na
    elif agg_func == 'q3':
        df = df.groupby(agg_freq).apply(lambda x: np.quantile(x.dropna(), .75) if len(x.dropna()) > 1 else np.na
    else:
        raise KeyError(f'{agg_func} unknonw, please specify in func')

    df.index = df.index.to_timestamp()
    return df


def adf_test_summary(ser):
    # ADF H0: there is a unit root

    specs = {'constant': 'c', 'constant trend': 'ct', 'constant ltend, qtrend': 'ctt', 'none': 'n'}
    results = {}

    for pretty, spec in specs.items():
        adf, pval, ulag, nobs, cval, icb = adfuller(ser, regression=spec)
        keys = ['adf-stat', 'p-value', 'lags', 'obs', *cval.keys(), 'inf crit']
        res = [adf, pval, ulag, nobs, *cval.values(), icb]
        results[pretty] = dict(zip(keys, res))

    if ser.name is not None:
        title = ser.name.upper()
    else:
        title = ''

    print('-' * 77)
    print(f'ADF Test {title}: H0 there is a unit root')
    print('-' * 77)
    print(pd.DataFrame(results).transpose().round(3).iloc[:, :-1])
    print('\n')

    pass


def hausman(fe, re):
    b = fe.params
    B = re.params
    v_b = fe.cov
    v_B = re.cov
    df = b[np.abs(b) < 1e8].size
    chi2 = np.dot((b - B).T, np.linalg.inv(v_b - v_B).dot(b - B))

    pval = scipy.stats.chi2.sf(chi2, df)
    return chi2, df, pval


def plt_stacked_bar(df, figsize: tuple = (20, 6), bar_width: float = 1.0):
    bottom = np.zeros(df.shape[0])
    dict_df = {k: np.array(list(v.values())) for k, v in df.to_dict().items()}
    color = cm.rainbow(np.linspace(0, 1, len(dict_df)))

    fig, ax = plt.subplots(figsize=figsize)
    for i, _ in enumerate(dict_df.items()):
        l, w = _

        p = ax.bar(df.index, w, label=l, width=bar_width, bottom=bottom, alpha=.5, color=color[i])
        bottom += w

    return fig, ax


def get_individual_perc_error(df_in, agg_col: str, pi_data: pd.DataFrame,
                              agg_col_suffix: str = None,
                              ind_cols: list = ['date_recorded', 'id'], count_thresh: int = 7):
    sub = df_in.reset_index().groupby(ind_cols)[agg_col].last().dropna().unstack()
    sub = sub.loc[:, sub.count() > count_thresh]
    sub = pd_join_freq(sub, pi_data, freq='M', keep_left_index=False, how='left')
    diff = sub.iloc[:, :-1].values - sub.iloc[:, -1].values[:, None]

    diff_act = np.array([diff[:, i][~np.isnan(diff[:, i])].mean() for i in range(diff.shape[1])]) * 100
    diff_mse = np.array([(diff[:, i][~np.isnan(diff[:, i])] ** 2).mean() for i in range(diff.shape[1])]) * 100

    if agg_col_suffix is not None:
        suffix = f"_{agg_col_suffix}"
    else:
        suffix = ""
    sub_act = pd.DataFrame(data=diff_act,
                           index=sub.iloc[:, :-1].columns,
                           columns=[f'{agg_col + suffix}_error_act']).reset_index(names=['id'])
    sub_mse = pd.DataFrame(data=diff_mse,
                           index=sub.iloc[:, :-1].columns,
                           columns=[f'{agg_col + suffix}_error_mse']).reset_index(names=['id'])

    sub = pd.merge(df_in, sub_act, left_on='id', right_on='id', how='left')
    sub = pd.merge(sub, sub_mse, left_on='id', right_on='id', how='left')

    return sub


def _xcorr_plot(corr, confu, confl, n_lags, dpi: int = 200, figsize: tuple = (14, 6)):
    index = np.linspace(-n_lags, n_lags, n_lags * 2 + 1)

    fig, ax = plt.subplots(figsize=figsize, dpi=dpi, )
    for i, idx in enumerate(index):
        ax.vlines(idx, 0, corr[i], color='blue', )

    ax.plot(index, corr, lw=0, marker='.', color='blue', label='corr')
    ax.plot(index, np.zeros(len(index)), color='black', alpha=.8)
    ax.fill_between(index, confl, confu, color='grey', alpha=.2, label='95% conf.')
    ax.vlines(0, *ax.get_ylim(), color='black', label='$t=0$', alpha=.5)

    ax.set_xlabel('lags in $t$')
    ax.set_ylabel('correlation')

    ax.set_title(f'Cross correlation with {int((len(corr) - 1) / 2)} lags')
    fig.legend()
    fig.tight_layout()
    return fig, ax


def xcorr(arr: 'float | Array like', arr1: 'float | Array like' = None, n_lags: int = None, plot_res: bool = True,
          **kwargs):
    """
    Cross and auto-correlation plot for arr
    :param arr:
    :param arr1:
    :param n_lags:
    :param plot_res:
    :param kwargs:
    :return:
    """
    arr = (arr - arr.mean()) / arr.std()
    if arr1 is None:
        arr1 = arr.copy()
```

```
else:
    arr1 = (arr1 - arr1.mean()) / arr1.std()
    assert len(arr) == len(arr1), "arrays do not align"

if n_lags is None:
    n_lags = len(arr) - 1
else:
    n_lags = min(n_lags, len(arr) - 1)

corr = np.correlate(arr, arr1, 'full')
corr = pd.Series(corr, index=np.linspace(-(int((len(corr) - 1) / 2)), (int((len(corr) - 1) / 2)), len(corr)))
corr /= corr.loc[0]
corr *= scipy.stats.pearsonr(arr, arr1).statistic

arr_lags = np.linspace(-n_lags, n_lags, n_lags * 2 + 1)
corr = corr.loc[arr_lags].values

conf = np.array([np.sqrt(2 / (len(arr) - np.abs(k))) for k in arr_lags])
confu, confl = 0 + 1.96 * conf, 0 - 1.96 * conf

if plot_res:
    fig = _xcorr_plot(corr, confu, confl, n_lags, **kwargs)
else:
    fig = None

return corr, conf, arr_lags, fig
```