```python
import os
import pickle
from datetime import datetime
from settings import NEWS_TEXT_DIR, GRAPHS_DIR
import numba as nb
import scipy
import matplotlib.pyplot as plt
from io import StringIO
import sys

import numpy as np
import pandas as pd
from itertools import compress
from settings import DATA_DIR, GRAPHS_DIR, DICT_PARSE_COLS


def get_dt_index(df: pd.DataFrame, dt_index_col=None, is_rename_date: bool = True):
    if dt_index_col is None: dt_index_col = 'date'
    df = df.set_index(pd.DatetimeIndex(df[dt_index_col].apply(lambda x: datetime.strptime(x, "%Y-%m-%d"))))
    if dt_index_col in df.columns:
        df = df.drop(dt_index_col, axis=1)
    if is_rename_date: df.index.name = 'date'
    return df


def load_pd_df(file_name, file_path=None, is_replace_nan=True, **kwargs):
    file_type = file_name.split('.')[-1]

    if file_path is None:
        file_path = DATA_DIR

    if file_type == 'csv':
        return pd.read_csv(os.path.join(file_path, file_name), **kwargs)
    elif file_type == 'xlsx':
        return pd.read_excel(os.path.join(file_path, file_name), **kwargs)
    elif file_type == 'feather':
        df = pd.read_feather(os.path.join(file_path, file_name), **kwargs)
        # if is_replace_nan:
        #     warnings.warn("replacing nan for feather format, pass 'is_replace_nan=False' to disable")
        #     df = df.replace({'nan': np.nan})
        return df
    else:
        raise KeyError(f"{file_type} unknown")


def save_pd_df(df, file_name: str, file_path=None):
    file_type = file_name.split('.')[-1]
    if file_path is None:
        file_path = GRAPHS_DIR

    if file_type == "csv":
        df.to_csv(os.path.join(file_path, file_name))
    elif file_type == "feather":
        df.to_feather(os.path.join(file_path, file_name))
    else:
        raise KeyError(f"{file_type} unknown")


class Capturing(list):
    def __init__(self, file_name: str, file_dir: str = None):
        if file_dir is None:
```

```
            file_dir = GRAPHS_DIR
        self.file_dir = file_dir
        self.file_name = file_name
        pass

    def __enter__(self):
        self._stdout = sys.stdout
        sys.stdout = self._stringio = StringIO()
        return self

    def __exit__(self, *args, **kwargs):
        self.extend(self._stringio.getvalue().splitlines())
        del self._stringio
        sys.stdout = self._stdout
        write_to_txt("\n".join(self), self.file_name, file_dir=self.file_dir)


def write_to_txt(output: str, file_name, file_dir=None):
    if file_dir is None: file_dir = GRAPHS_DIR
    f = open(os.path.join(file_dir, file_name), 'w+')
    f.write(output)
    f.close()
    pass


def save_pkl(file: dict, f_name: str, f_path: str = None):
    if f_path is None:
        f_path = NEWS_TEXT_DIR
    t = open(os.path.join(f_path, f"{f_name}"), "wb+")
    pickle.dump(file, t)
    t.close()
    pass


def save_fig(fig, f_name: str, f_path: str = None):
    if f_path is None:
        f_path = GRAPHS_DIR
    fig.savefig(os.path.join(f_path, f_name))
    pass


def load_pickle(f_name, f_path=None):
    if f_path is None:
        f_path = NEWS_TEXT_DIR
    t = open(os.path.join(f_path, f_name), 'rb')
    file = pickle.load(t)
    t.close()
    return file


@nb.njit()
def frobenius_norm(a):
    norms = np.empty(a.shape[0], dtype=a.dtype)
    for i in nb.prange(a.shape[0]):
        norms[i] = np.sqrt(np.sum(a[0] ** 2))
    return norms


@nb.njit()
def arr_norm(arr, axis=0):
    return np.sqrt(np.sum(arr ** 2, axis=axis))
```

```python
@nb.njit()
def arr_to_unity(arr):
    # norm = frobenius_norm(arr)
    norm = arr_norm(arr, axis=1)
    is_null = np.abs(norm) == 0
    norm[is_null] = np.ones(is_null.sum()) * 1e-8
    arr = np.divide(arr, norm[:, None])
    return arr


@nb.njit()
def vec_similarity(arr, search_terms):
    arr, search_terms = arr_to_unity(arr), arr_to_unity(search_terms)
    return np.dot(arr, search_terms.T)


def arr_min_max_scale(arr):
    if arr.min() != arr.max():
        return (arr - arr.min()) / (arr.max() - arr.min())
    else:
        return arr


def pd_join_freq(df1, df2, freq: str = 'D', keep_left_index: bool = True, **kwargs):
    df1, df2 = df1.copy(), df2.copy()

    for d in [df1, df2]:
        if d.index.name == freq:
            d.index.name = f'{d.index.name}_2'

    if keep_left_index:
        df1[df1.index.name] = df1.index
        # df2['index_right'] = df2.index

    df1[freq] = df1.index.to_period(freq)
    df2[freq] = df2.index.to_period(freq)

    df = pd.merge(df1, df2, on=freq, **kwargs).set_index(freq)  # , axis=1)
    df.index = df.index.to_timestamp()
    if keep_left_index:
        df = df.set_index(df1.index.name, drop=False)
    return df


def cross_corr(arr1, arr2, lags: int = 10, is_plot: bool = True, figisize: tuple = None, **kwargs):
    assert arr1.shape == arr2.shape, "please ensure both arrays are of same dimensions"

    if figisize is None:
        figisize = plt.rcParams["figure.figsize"]

    lags = min(len(arr1) - 1, lags)
    y1, y2 = 2 / np.sqrt(len(arr1)), -2 / np.sqrt(len(arr1))
    corr = scipy.signal.correlate(arr1, arr2, mode='full', **kwargs) / np.sqrt(arr1.std() ** 2 * arr2.std() ** 2)
    corr = corr[len(arr1) - 1 - lags: len(arr1) - 1 + lags]

    if is_plot:
        fig, ax = plt.subplots(1, 1, figisize=figisize)
        # idx = [*range(len(corr))]
        idx = np.linspace(-lags, lags, lags * 2)
        ax.fill_between(idx, y1, y2, alpha=.2)
        ax.axhline(y=0, color='black')
```

```
        ax.bar(idx, corr, color='blue', width=.5)
        ax.set_xlabel(f'no. of lags')
        ax.set_ylabel(f'correlation')
        ax.set_title(f"Cross correlation with {lags} lags")
        plt.tight_layout()

        return corr, y1, y2, fig

    else:
        return corr, y1, y2


def pd_df_astype(df_in, dict_dtypes: dict = None):
    df = df_in.copy()
    if dict_dtypes is None:
        dict_dtypes = DICT_PARSE_COLS

    _ = [i in list(dict_dtypes.keys()) for i in df.columns]
    assert sum(_) == len(df.columns), f"{[*compress(df.columns, ~np.array(_))]} not in parse dict"

    _ = [i in df.columns for i in list(dict_dtypes.keys())]
    dict_dtypes = {k: dict_dtypes[k] for k in [*compress(dict_dtypes.keys(), np.array(_))]}

    dict_dtypes_cat = {k: v for k, v in dict_dtypes.items() if 'category' in str(v)}
    dict_dtypes = {k: v for k, v in dict_dtypes.items() if 'category' not in str(v)}

    for col, dtype in dict_dtypes_cat.items():
        if dtype == "categoryO":
            c = pd.CategoricalDtype(sorted([float(i) for i in set(df[col].dropna())]), ordered=True)
            df[col] = df[col].astype(c)
        elif dtype == "category":
            df[col] = df[col].astype(dtype)
        else:
            raise KeyError(f"{dtype} unknown, pleas specify")

    df = df.astype(dict_dtypes)

    return df


def get_stars(p_val: float):
    if p_val <= .01:
        return '***'
    elif p_val <= .05:
        return '**'
    elif p_val <= .1:
        return '*'
    else:
        return ''


def get_statsmodels_tab(lst_models: list, n_round: int = 4, join_on: str = "\n"):
    dfs = [mod.summary().tables[1].data for mod in lst_models]

    cols = [map(list, zip(*[list([mod.model.endog_names] * len(dfs[i][0])), dfs[i][0]])) for i, mod in
            enumerate(lst_models)]
    cols = [[tuple(z) for z in [*i]] for i in cols]

    dfs = [pd.DataFrame(summary[1:], columns=pd.MultiIndex.from_tuples(cols[i])) for i, summary in enumerate(dfs)
    dfs = [df.set_index(df.iloc[:, 0]).iloc[:, 1:] for df in dfs]

    # coefficient table
```

```
    col_vals = ['coef', 'std err', 'pval', 'P>|t|']

    out1 = dfs[0]
    for i in dfs[1:]:
        out1 = out1.join(i)
    out1.index.name = ''
    filt = [out1.columns.get_level_values(1) == f for f in col_vals]
    filt = np.array(filt).T.sum(axis=1) > 0
    out1 = out1.loc[:, filt]

    # extra information
    out2 = [{'N': mod.model.nobs, 'R2': mod.rsquared, 'R2 ajd.': mod.rsquared_adj} for mod in lst_models]
    out2 = pd.DataFrame(out2, index=[mod.model.endog_names for mod in lst_models]).T.round(n_round)

    out3 = {}
    for endog in list(set(out1.columns.get_level_values(0))):
        _ = {}
        for i, row in out1.loc[:, endog].iterrows():
            row = list(row)
            stars = get_stars(float(row[2]))

            _[i] = f"{float(row[0])}{stars}{join_on}[{float(row[1])}]"

        out3[endog] = _
    out3 = pd.DataFrame(out3)

    out4 = out3.T.join(out2.T).T

    return out1, out2, out3, out4


def min_max_scale(x):
    return (x - x.min()) / (x.max() - x.min())
```