```python
import pandas as pd
import numpy as np
import spacy
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.utils import check_random_state
from sklearn.decomposition._online_lda_fast import _dirichlet_expectation_2d

from patsy import dmatrix
import pymc as pm
import arviz as az

import multiprocessing
import time

from tqdm import tqdm
import os
from uuid import uuid4
import pickle

from settings import DATA_DIR, NEWS_TEXT_DIR
from src.utils import load_pickle, save_pd_df


def get_spacy_NLP(lang: str = 'de'):
    """
    Get spacy pipeline by language
    :param lang:
    :return:
    """
    if lang == 'de':
        nlp = spacy.load(
            os.path.abspath(
                "C:\\Users\\LukasGrahl\\Documents\\GIT\\memoire2\\spacy\\de_core_news_lg\\de_core_news_lg-3.7.0")
        )
    elif lang == 'en':
        nlp = spacy.load(
            os.path.abspath(
                "C:\\Users\\LukasGrahl\\Documents\\GIT\\memoire2\\spacy\\en_core_web_lg\\en_core_web_lg-3.7.1")
        )
    else:
        raise KeyError(f"please specify file location of {lang} package")
    return nlp


def get_lda_n_top_words(voc, lda, n_top: int = 10):
    """
    Print top n words by topic for sklearn LDA
    :param voc:
    :param lda:
    :param n_top:
    :return:
    """
    dict_topics = {
        idx_topic: sorted(zip(voc, lda.components_[idx_topic]), key=lambda x: x[1], reverse=True)
        for idx_topic in range(lda.n_components)
    }

    topics = {}
    for idx, lst in enumerate([[i[0] for i in lst] for k, lst in dict_topics.items()]):
        topics[idx] = lst
        print(f"\nTopic {idx}")
        print(" ".join(lst[:n_top]))
```

```python
    return topics



def load_raw_data(f_name: str):
    """
    Load raw articles data as obtained from FACTIVA
    :param f_name:
    :return:
    """
    data = load_pickle(f_name, f_path=DATA_DIR)

    df = pd.DataFrame(data)
    df['id'] = [str(uuid4()) for i in range(len(df))]
    df = df.set_index('id')
    save_pd_df(df, 'news_data.feather')

    df = df.rename(columns={'datetime': 'date'})
    df = df.sort_values('date')
    df['date'] = df.date.apply(lambda x: x.date())

    for idx, row in tqdm([*df.iterrows()]):
        f = open(os.path.join(NEWS_TEXT_DIR, 'orig', f'{idx}.pkl'), 'wb+')
        pickle.dump(row.to_dict(), f)
        f.close()
    pass


class PTWGuidedLatentDirichletAllocation(LatentDirichletAllocation):
    """
    Guided LDA wrapper class for Sklearn LDA
    """

    def __init__(self, n_components=10, doc_topic_prior=None, topic_word_prior=None, learning_method='online',
                 learning_decay=0.7, learning_offset=10.0, max_iter=10, batch_size=128, evaluate_every=-1,
                 total_samples=1000000.0, perp_tol=0.1, mean_change_tol=0.001, max_doc_update_iter=100, n_jobs=None,
                 verbose=0, random_state=None, ptws=None, ptws_bias=None):
        super(PTWGuidedLatentDirichletAllocation, self).__init__(n_components=n_components,
                                                                  doc_topic_prior=doc_topic_prior,
                                                                  topic_word_prior=topic_word_prior,
                                                                  learning_method=learning_method,
                                                                  learning_decay=learning_decay,
                                                                  learning_offset=learning_offset, max_iter=max_iter,
                                                                  batch_size=batch_size, evaluate_every=evaluate_every,
                                                                  total_samples=total_samples, perp_tol=perp_tol,
                                                                  mean_change_tol=mean_change_tol,
                                                                  max_doc_update_iter=max_doc_update_iter, n_jobs=n_jobs,
                                                                  verbose=verbose,
                                                                  random_state=random_state, )  # n_topics=n_topics)
        assert len(ptws) == self.n_components, "number of prior categories must concur with n_components"
        self.ptws = ptws

        if ptws_bias is None:
            self.ptws_bias = self.n_components
        else:
            self.ptws_bias = ptws_bias

    def _init_latent_vars(self, n_features, dtype):
        """Initialize latent variables."""

        self.random_state_ = check_random_state(self.random_state)
        self.n_batch_iter_ = 1
```

```python
        self.n_iter_ = 0

        if self.doc_topic_prior is None:
            self.doc_topic_prior_ = 1. / self.n_components
        else:
            self.doc_topic_prior_ = self.doc_topic_prior

        if self.topic_word_prior is None:
            self.topic_word_prior_ = 1. / self.n_components
        else:
            self.topic_word_prior_ = self.topic_word_prior

        init_gamma = 100.
        init_var = 1. / init_gamma
        # In the literature, this is called `lambda`
        self.components_ = self.random_state_.gamma(
            init_gamma, init_var, (self.n_components, n_features))

        # Transform topic values in matrix for prior topic words
        if self.ptws is not None:
            for topic_ind, lst_prior_words in enumerate(self.ptws):
                for word_ind in lst_prior_words:
                    # self.components_[topb:, word_index] *= word_topic_values
                    # word_index = ptw[0]
                    # word_topic_values = ptw[1]
                    self.components_[:, word_ind] *= 0
                    self.components_[topic_ind, word_ind] = self.topic_word_prior_ * self.ptws_bias

        # In the literature, this is `exp(E[log(beta)])`
        self.exp_dirichlet_component_ = np.exp(
            _dirichlet_expectation_2d(self.components_))


def get_topic_smooth(ser: pd.Series, n_knots: int = 5, is_samp_post_prior: bool = True, **kwargs):
    """
    Bayesian MCMC spline regression estimation for n_knots on ser
    :param ser:
    :param n_knots: number of spline knots
    :param is_samp_post_prior: sample predictive prior
    :param kwargs:
    :return:
    """
    knot_list = np.linspace(0, len(ser), n_knots + 2)[1:-1]

    B = dmatrix(
        "bs(cnt, knots=knots, degree=3, include_intercept=True)-1",
        {"cnt": range(len(ser)), "knots": knot_list[1:-1]}
    )

    with pm.Model() as mod:
        tau = pm.HalfCauchy("tau", 1)
        beta = pm.Normal("beta", mu=0, sigma=tau, shape=B.shape[1])
        mu = pm.Deterministic("mu", pm.math.dot(B.T.T, beta))
        sigma = pm.HalfNormal("sigma", 1)
        pm.Normal("likelihood", mu, sigma, observed=ser.values)

        trace = pm.sample(1000, nuts_sampler="numpyro", chains=2, **kwargs)
        if is_samp_post_prior:
            prior = pm.sample_prior_predictive()
            post = pm.sample_posterior_predictive(trace)

            return mod, prior, trace, post, B
```

```python
    else:
        return mod, trace, B


def evalute_optimal_smoothing(ser, search_range: range):
    """
    Runs serval spline regressions for a search_range of integer values
    :param ser:
    :param search_range: integer range
    :return:
    """
    mods, traces = {}, {}

    for k in search_range:
        mod, trace, _ = get_topic_smooth(
            ser,
            n_knots=k,
            is_samp_post_prior=False,
            return_inferencedata=True,
            idata_kwargs={'log_likelihood': True}
        )

        mods[k] = mod
        traces[k] = trace

    df = az.compare(traces, ic="waic")
    return df, mods, traces


def _run(arguemnts):
    """
    helper function to run_parrallel
    :param arguemnts:
    :return:
    """
    df, id_col, cols = arguemnts

    dict_compare_az, dict_best_nknot, dict_compare_traces, dict_data_grouped = {}, {}, {}, {}
    for col in cols:
        g = df.groupby(id_col)[col].sum().replace({0: np.nan})

        az_df, mods, traces = evalute_optimal_smoothing(g, search_range=range(5, 80, 5))
        dict_best_nknot[col] = az_df[az_df['rank'] == 0].index[0]
        dict_compare_az[col] = az_df
        dict_compare_traces[col] = traces
        dict_data_grouped[col] = g

    return dict_compare_az, dict_best_nknot, dict_compare_traces, dict_data_grouped


def run_parallel(df, id_col: str = 'date'):
    """
    Runs evalute_optimal_smoothing MCMC spline regression in parallel
    :param df:
    :param id_col:
    :return:
    """

    assert id_col in df.columns, f"{id_col} not contained in columns"

    inputs = list(df.drop(id_col, axis=1).columns)
    N = int(np.ceil(len(inputs) / os.cpu_count()))
```

```
    inputs = [(df, id_col, tuple(inputs[i:i + N])) for i in range(0, len(inputs), N)]

    with multiprocessing.Pool(processes=os.cpu_count()) as pool:
        start = time.time()
        res = pool.map(_run, inputs)
    print(f"This process ran {time.time() - start:<=.4}")
    return res
```