Regime Modeling with NLP (with modules)

October 13, 2023

Group Members: Dhruv Baid, Prajakta Phadke, Uday Sharma

Internal Modules

cross_validation.py

[ ]:
```
from typing import Type

import numpy as np
import pandas as pd
from sklearn.model_selection import ParameterGrid

import NaiveBayesClassifier as nbc
import directional_change as dc
import hidden_markov_model as hmm
import trading_strategy as ts
import logistic_regression as lr
import svm as svm
import pandas as pd
from operator import itemgetter

def _initialize_loss(minimize: bool):
    if minimize:
        return np.inf
    else:
        return -np.inf


class CustomCrossValidation:
    def __init__(self, pipeline_class: Type, parameter_grid: dict, verbose:␣
 ↪bool = False):
        self.optimal_parameters = None
        self.losses = None
        self.pipeline_class = pipeline_class
        self.parameter_grid = parameter_grid
        self.is_verbose = verbose
        self.optimal_loss = None
```

```
        self.grid_size = None
        self.metric = None

    def fit(self, data: pd.DataFrame, metric: str = None, minimize: bool =␣
↪True):

        self.losses = []
        self.optimal_parameters = None
        self.optimal_loss = None
        self.metric = metric

        optimum = _initialize_loss(minimize)
        parameter_grid = ParameterGrid(self.parameter_grid)
        self.grid_size = len(parameter_grid)
        for idx, params in enumerate(parameter_grid):
            pipeline = self.pipeline_class(df_ts=data, **params)
            self._pprint(idx, "Parameters: {}".format(params))
            pipeline.fit()
            self._pprint(idx, "Training complete.")
            loss = pipeline.trading_metrics
            self._pprint(idx, "Loss: {}".format(loss))
            loss['parameters'] = params
            self.losses.append(loss)
            if metric is not None:
                optimum = self._find_optimum_value(loss, metric, minimize,␣
↪optimum, params)

    def get_optimal_parameters(self):
        return self.optimal_parameters

    def get_losses(self):
        return self.losses

    def get_optimal_loss(self):
        return self.optimal_loss

    def _find_optimum_value(self, loss: dict, metric: str, minimize: bool,␣
↪optimum: float, parameters: dict):
        if minimize:
            if loss[metric] < optimum:
                self.optimal_parameters = parameters
                self.optimal_loss = loss
                return loss[metric]
        else:
            if loss[metric] > optimum:
                self.optimal_parameters = parameters
                self.optimal_loss = loss
```

```
                return loss[metric]
        return optimum

    def _pprint(self, idx, out):
        if self.is_verbose:
            print("Iteration: {} of {}: {}".format(idx + 1, self.grid_size,␣
↪out))

    def get_results_in_latex(self, caption = None, loss_columns: list = None,␣
↪parameter_columns: list = None):
        if parameter_columns is None:
            parameter_columns = list(self.parameter_grid.keys())
        if loss_columns is None:
            loss_columns = list(self.get_optimal_loss().keys())
            loss_columns.remove('parameters')
        parameter_columns = sorted(parameter_columns)
        loss_columns = sorted(loss_columns)
        flattened_results = \
            [list(itemgetter(*parameter_columns)(row['parameters']))
             + list(itemgetter(*loss_columns)(row)) for row in self.
↪get_losses()]
        df = pd.DataFrame(data=flattened_results, columns=parameter_columns +␣
↪loss_columns)
        df.reset_index(inplace=True)
        df['index'] += 1
        df.rename(columns={'index': 'Iterations', 'DC_indicator': 'dc␣
↪indicator'}, inplace=True)
        styler = df.style
        styler.set_precision(4)
        styler.hide_index()
        styler.highlight_max(color='red', axis=0, subset=self.metric)
        ret_val = styler.to_latex(hrules=True,
                                  environment='longtable',
                                  column_format='c' * df.columns.__len__(),
                                  caption=caption)
        ret_val = ret_val.replace('\\background-colorred', '\cellcolor{red}')
        return ret_val




class Pipeline:

    def __init__(self, df_ts: pd.DataFrame, type_: str = 'equity', type_mapper:␣
↪dict = {'equity':['^GSPC'],'bond':['^IRX'],
```

```
                'fx':['GBP=X']}, start_date: str = "2005-01-01",
                train_end: str = "2017-12-31", valid_start: str = "2018-01-01",
                valid_end:str = "2019-12-31", test_start:str = "2020-01-01",
                theta: float = 0.025, num_regimes: int = 2, trading_day: dict =␣
↪{'equity':6.5, 'fx':12,'bond':9},
                DC_indicator: str = "R", threshold: float = 0.5, strat: str =␣
↪"JC1", init_cap: int = 1, to_test: bool = False,
                epsilon: float = 0.5, model: str = 'naive_bayes',␣
↪provide_labels: bool = False, labels = None):

        """Initializes the pipeline parameters.

        Args:
            df_ts (pd.DataFrame): price dataframe. Pulled from Yahoo Finance.
            type_ (str, optional): Asset class. Defaults to 'equity'. 'equity'␣
↪or 'fx' or 'bond'
            type_mapper (_type_, optional): Maps the type to tickers. Defaults␣
↪to {'equity':['^DJI','^GSPC'],'bond':['^TNX', '^IRX'], 'fx':
↪['RUB=X','GBP=X']}.
            start_date (str, optional): Start date for training set. Defaults␣
↪to "2005-01-01".
            train_end (str, optional): End date for training set. Defaults to␣
↪"2017-12-31".
            valid_start (str, optional): Start date for validation set.␣
↪Defaults to "2018-01-01".
            valid_end (str, optional): End date for validation set. Defaults to␣
↪"2019-12-31".
            test_start (str, optional): Start date for test set. Defaults to␣
↪"2020-01-01".
            theta (float, optional): Threshold for DC. Defaults to 0.025.
            num_regimes (int, optional): Number of regimes. Defaults to 2.
            trading_day (_type_, optional): Offset between open and close data.␣
↪Defaults to {'equity':6.5, 'fx':12,'bond':9}.
            DC_indicator (str, options): Which indicator to use
            threshold (float, optional): What threshold for TMV do we trade
            strat (str, optional): Name of strategy
            init_cap (int, optional): Starting capital for the strategy
            to_test (bool, optional): Whether we are fitting on the trainging␣
↪set or testing on test set
            epsilon (float, optional): Min prob for predicting class 1
            provide_labels (bool, optional): whether we supply the regime labels
            labels: the above labels
        """
        self.df_ts = df_ts
        self.type_ = type_
        self.type_mapper = type_mapper
```

```
        self.tickers = type_mapper[type_]
        self.ticker = self.tickers[0]
        self.start_date = start_date
        self.train_end = train_end
        self.valid_start = valid_start
        self.valid_end = valid_end
        self.test_start = test_start
        self.theta = theta
        self.num_regimes = num_regimes
        self.trading_day = trading_day
        self.DC_indicator = DC_indicator
        self.dict_indicators = {}
        self.model = model
        self.provide_labels = provide_labels
        self.labels = labels

        self.regimes_valid = {}  # Regimes predicted on validation set
        self.trading_metrics = {}  # Metrics for trading strategy
        self.threshold = threshold  # Threshold for TMV for strategy
        self.strat = strat  # Name for strategy ( 'control' for control␣
↪strategy, anything else for test strategy)
        self.init_cap = init_cap

        self.to_test = to_test
        self.regimes_test = {} # Regimes predicted on test set
        self.trading_metrics_test = {} # Metrics for trading strategy on test␣
↪set
        self.epsilon = epsilon # Prob for regime 1

    def fit(self, plot: bool = False, verbose: bool = False):
        """Fits the pipeline

        Args:
            plot (bool, optional): Plot the regimes. Defaults to False.
            verbose (bool, optional): Whether debug output has to be printed.␣
↪Defaults to False.
        """

        # df_ts = dc.get_data(self.tickers, self.start_date, self.
↪trading_day[self.type_]/2)

        self.ts = {}
        self.ts['train'] = self.df_ts.loc[:self.train_end]
        self.ts['valid'] = self.df_ts.loc[self.valid_start:self.valid_end]
        self.ts['test'] = self.df_ts.loc[self.test_start:]

        self.dc = {}
```

```
        for cat in ['train', 'valid', 'test']:
            self.dc[cat] = dc.get_DC_data_v2(self.ts[cat], self.theta)

        self.tmv = {}
        self.T = {}
        self.R = {}
        for cat in ['train', 'valid', 'test']:
            self.tmv[cat], self.T[cat], self.R[cat] = {}, {}, {}
            self.tmv[cat] = dc.get_TMV(self.dc[cat], self.theta)
            self.T[cat] = dc.get_T(self.dc[cat])
            self.R[cat] = dc.get_R(self.tmv[cat], self.T[cat], self.theta)

        self.dict_indicators['R'] = self.R
        self.dict_indicators['T'] = self.T
        self.dict_indicators['TMV'] = self.tmv

        self.regimes = {}

        reg, _ = hmm.fit_hmm(self.num_regimes, self.ts['train'], self.
↪dict_indicators[self.DC_indicator]['train'],
                             self.ticker, plot=plot, verbose=verbose)
        self.regimes = reg

        '''Creating labels for validation set using a specified Classifier'''
        if self.model=='naive_bayes':
            self.regimes_valid = nbc.do_all_NBC(self.dict_indicators[self.
↪DC_indicator]['train'].values.reshape(-1, 1),
                                                self.regimes,
                                                self.dict_indicators[self.
↪DC_indicator]['valid'].values.reshape(-1, 1), self.epsilon)
        elif self.model=='logistic_regression':
            self.regimes_valid = lr.do_all_LR(self.dict_indicators[self.
↪DC_indicator]['train'].values.reshape(-1, 1),
                                                self.regimes,
                                                self.dict_indicators[self.
↪DC_indicator]['valid'].values.reshape(-1, 1), self.epsilon)
        elif self.model=='svm':
            self.regimes_valid = svm.do_all_SVM(self.dict_indicators[self.
↪DC_indicator]['train'].values.reshape(-1, 1),
                                                self.regimes,
                                                self.dict_indicators[self.
↪DC_indicator]['valid'].values.reshape(-1, 1),
                                                self.epsilon)

        self.regimes_valid = pd.Series(self.regimes_valid, index=self.
↪dict_indicators[self.DC_indicator]['valid'].index)
```

```
        self.trading_metrics = ts.get_loss_function_for_pipeline(self.
↪ts['valid'], self.dc['valid'], self.regimes_valid,
                                                                 self.theta,␣
↪init_cap=self.init_cap, strat=self.strat,
                                                                 threshold=self.
↪threshold)
        self.trading_metrics = self.trading_metrics[self.strat]


        if( self.to_test ):
            if not self.provide_labels:
                if self.model=='naive_bayes':
                    self.regimes_test = nbc.do_all_NBC(self.
↪dict_indicators[self.DC_indicator]['train'].values.reshape(-1, 1), self.
↪regimes, self.dict_indicators[self.DC_indicator]['test'].values.reshape(-1,␣
↪1), self.epsilon)
                elif self.model=='logistic_regression':
                    self.regimes_test = lr.do_all_LR(self.dict_indicators[self.
↪DC_indicator]['train'].values.reshape(-1, 1), self.regimes, self.
↪dict_indicators[self.DC_indicator]['test'].values.reshape(-1, 1), self.
↪epsilon)
                elif self.model=='svm':
                    self.regimes_test = svm.do_all_SVM(self.
↪dict_indicators[self.DC_indicator]['train'].values.reshape(-1, 1), self.
↪regimes, self.dict_indicators[self.DC_indicator]['test'].values.reshape(-1,␣
↪1), self.epsilon)
                self.regimes_test = pd.Series( self.regimes_test, index = self.
↪dict_indicators[self.DC_indicator]['test'].index )
                self.trading_metrics_test = ts.get_loss_function_for_pipeline(␣
↪self.ts['test'], self.dc['test'], self.regimes_test, self.theta, init_cap =␣
↪self.init_cap, strat = self.strat, threshold = self.threshold)
                self.trading_metrics_test = self.trading_metrics_test[self.
↪strat]
            else:
                self.regimes_test = self.labels
                self.trading_metrics_test = ts.get_loss_function_for_pipeline(␣
↪self.ts['test'], self.dc['test'], self.regimes_test, self.theta, init_cap =␣
↪self.init_cap, strat = self.strat, threshold = self.threshold)
                self.trading_metrics_test = self.trading_metrics_test[self.
↪strat]


#%%
```

data.py

[ ]:
```
from datetime import datetime, timedelta
import numpy as np
import pandas as pd
import yfinance as yf


def get_ts_data(tickers, start_date, delta, end_date='2022-12-31'):
    data = yf.download(tickers, start=start_date, end=end_date)  # time series␣
 ↪data
    df_ts_close = data['Adj Close'].dropna()
    df_ts_open = data['Open'].dropna()
    df_ts_open.index = df_ts_open.index + pd.Timedelta(f'{delta}h')  # adjust␣
 ↪time
    df_ts = pd.concat([df_ts_close, df_ts_open]).sort_index()

    return df_ts


def get_text_data(fpath='../assets/fomc_documents.csv', s_date=datetime(1985,␣
 ↪1, 1)):
    # Get starting year for text data
    if isinstance(s_date, datetime):
        s_year = s_date.date().year
    elif isinstance(s_date, str) or isinstance(s_date, int):
        s_year = int(str(s_date)[:4])
    else:
        print(f"Check that your input date is in the correct format!")
        s_year = None
    s_year = s_year if s_year is not None else 1985

    # Read in FOMC Data
    try:
        fomc_data = pd.read_csv(fpath)
    except Exception as e:
        print(f"Please check that {fpath} is a valid path pointing to FOMC CSV␣
 ↪data!")
        return

    fomc_data.meeting_date = pd.to_datetime(fomc_data.meeting_date)
    fomc_data = fomc_data[fomc_data.document_kind.isin([
        'historical_minutes',
        'minutes',
        'minutes_of_actions'
    ])]
    # fomc_data['meeting_year'] = fomc_data.meeting_date.dt.year
```

```
    fomc_data = fomc_data[fomc_data.meeting_date.dt.year >= s_year]
    return fomc_data


if __name__ == "__main__":
    print(f"Please import this file as a module!")
```

directional_change.py

[ ]:
```
import pandas as pd
import yfinance as yf
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as md


def get_data(tickers, start_date, delta, end_date='2022-12-31'):
    data = yf.download(tickers, start=start_date, end=end_date)  # time series␣
 ↪data
    df_ts_close = data['Adj Close'].dropna()
    df_ts_open = data['Open'].dropna()
    df_ts_open.index = df_ts_open.index + pd.Timedelta(f'{delta}h')  # adjust␣
 ↪time
    df_ts = pd.concat([df_ts_close, df_ts_open]).sort_index()

    return df_ts


def get_pct_change(start, end):
    return (end - start) / start


def get_DC_data_v2(prices: pd.Series, theta: float) -> list[tuple]:
    """

    :param prices: prices
    :param theta: threshold
    :return: Returns a list of tuples. Each tuple is of the form
             (Directional Change Confirmation timestamp, Directional Change␣
 ↪Confirmation price,
             Downturn/Upturn time, Downturn/Upturn price)
             {DCC_time, DCC_Price, EXT_time, EXT_Price}
    """

    last_high = last_low = prices[0]
    last_low_time = last_high_time = prices.index[0]
```

```
    is_upward_run = is_downward_run = is_downward_overshoot =␣
↪is_upward_overshoot = False
    ret_val = []

    for timestamp, current_price in prices[1:].items():
        if get_pct_change(last_high, current_price) <= -theta:
            is_downward_run = True
            is_upward_run = False
            is_upward_overshoot = False
            if not is_downward_overshoot:
                # reached a DC confirmation point
                ret_val.append((timestamp, current_price, last_high_time,␣
↪last_high))
                is_downward_overshoot = True
            last_high = current_price
            last_high_time = timestamp
        elif get_pct_change(last_low, current_price) >= theta:
            is_upward_run = True
            is_downward_run = False
            is_downward_overshoot = False
            if not is_upward_overshoot:
                # reached a DC confirmation point
                ret_val.append((timestamp, current_price, last_low_time,␣
↪last_low))
                is_upward_overshoot = True
            last_low = current_price
            last_low_time = timestamp
        if last_low >= current_price:
            last_low = current_price
            last_low_time = timestamp
        if last_high <= current_price:
            last_high = current_price
            last_high_time = timestamp
    return ret_val


def get_DC_data(data: pd.Series, theta: float) -> list[tuple]:
    """Returns the Directional Change (DC) data for a given price series.

    Args:
        data (pd.Series): price
        theta (float): threshold

    Returns:
        # tuple[pd.Series]: Directional Change Confirmation and Extreme Points␣
↪(DCC,EXT)
        return: Returns a list of tuples. Each tuple is of the form
```

```
            (Directional Change Confirmation timestamp, Directional Change␣
↪Confirmation price,
            Downturn/Upturn time, Downturn/Upturn price)
            {DCC_time, DCC_Price, EXT_time, EXT_Price}
    """

    rets = data.pct_change().dropna().to_numpy()  # pct change returns
    DCC = []  # idx for directional change confirmations
    EXT = []  # idx for extreme points

    prev_sign = np.sign(rets[0]).astype(int)  # store sign(return) from the␣
↪previous time step
    accumulated = rets[0]  # accumulated % return
    idx_change = 0  # index the direction changes, candidate for EXT
    sign_already_flagged = 0

    for idx, ret in zip(range(1, len(rets)), rets[1:]):

        ret_sign = np.sign(ret).astype(int)

        if (ret_sign != prev_sign):
            # sign is different from previous time step, trend ends
            idx_change = idx - 1  # previous price point is a candidate for EXT
            accumulated = ret  # reset accumulated sum


        # same sign
        elif ret_sign != sign_already_flagged:
            # once we flag a threshold, we don't flag it again for the same␣
↪trend
            # a peak has to be followed by a trough and vice versa
            accumulated += ret
            if np.abs(accumulated) > theta:
                # we cross the threshold
                DCC.append(idx)
                EXT.append(idx_change)
                sign_already_flagged = ret_sign

        prev_sign = ret_sign  # set the last seen sign to the current sign

    DCC = data.iloc[1:].iloc[DCC]
    EXT = data.iloc[1:].iloc[EXT]

    ans = []
    for i in range(len(DCC)):
        ans.append((DCC.index[i], DCC[i], EXT.index[i], EXT[i]))
    return ans
```

```python
def get_DCC_EXT(DC: list) -> tuple((list, list, list, list)):
    """
    Get the DCC, EXT points with prices at those points and the time

    Args:
        DC(list): output_of_get_DC_data{,_v2}

    Returns:
        (list(DCC), list(DCC_time), list(EXT), list(EXT_time))
    """
    EXT = []
    EXT_index = []
    DCC = []
    DCC_index = []

    for i in range(len(DC)):
        DCC.append(DC[i][1])
        DCC_index.append(DC[i][0])
        EXT.append(DC[i][3])
        EXT_index.append(DC[i][2])

    return (DCC, DCC_index, EXT, EXT_index)


def get_TMV(DC: list, theta: float) -> pd.Series:
    """Gets the total price movement (TMV), which is the absolute percentage of␣
 ↪the price change in a trend, normalized by the threshold.

    Args:
        DC (list): output of get_DC_data{,_v2}
        theta (float): threshold

    Returns:
        pd.Series: total price movement at respective timestamps
    """
    _, _, ext, idx = get_DCC_EXT(DC)

    ext = pd.Series(data=ext, index=idx)
    return ext.pct_change().dropna() / theta


def get_T(DC: list) -> pd.Series:
    """Gets the time for completion of a TMV trend, in days.

    Args:
```

```
        DC (list): output of get_DC_data{,_v2}

    Returns:
        pd.Series: time for completion of trends at respective timestamps
    """
    # extract number of days and hours between extreme points

    _, _, ext, idx = get_DCC_EXT(DC)

    t_ext = pd.Series(idx).diff().dropna().apply(lambda x: x.days + (x.seconds /
 ↪/ 3600) / 24)
    t_ext.index = idx[1:]
    return t_ext


def get_R(tmv: pd.Series, T: pd.Series, theta: float) -> pd.Series:
    """Gets the absolute return (R), which is the time-adjusted return of DC.

    Args:
        tmv (pd.Series): total price movement
        T (pd.Series): time for completion of a trend
        theta (float): threshold

    Returns:
        pd.Series: time-adjusted return of DC
    """
    return tmv * theta / T


def annotate_plot(ax, sample_ext, sample_dcc):
    prop1 = dict(arrowstyle="-|>,head_width=0.75,head_length=0.8",
                 shrinkA=0, shrinkB=0, color="red", lw=2)
    prop2 = dict(arrowstyle="-|>,head_width=0.75,head_length=0.8",
                 shrinkA=0, shrinkB=0, color="green", lw=2)
    prop3 = dict(arrowstyle="-", shrinkA=0, shrinkB=0, color="black",␣
 ↪ls="dashed", lw=2)
    prop4 = dict(arrowstyle="<-", shrinkA=0, shrinkB=0, color="black",␣
 ↪ls="dashed", lw=2)
    prop5 = dict(arrowstyle="->", shrinkA=0, shrinkB=0, color="black",␣
 ↪ls="dashed", lw=2)

    plt.annotate("", xytext=(sample_ext.index[0], sample_ext[0]),
                 xy=(sample_dcc.index[0], sample_dcc[0]), arrowprops=prop1)

    plt.annotate("", xytext=(sample_ext.index[0], sample_ext[0]),
                 xy=(sample_ext.index[0], sample_dcc[0]), arrowprops=prop5)
```

```
    plt.annotate("Theta", xytext=(md.date2num(sample_ext.index[0]) + 0.1,␣
↪sample_dcc[0] + 5),
                 xy=(sample_ext.index[0], sample_ext[0]), rotation=270)

    plt.annotate("", xytext=(sample_dcc.index[0], sample_dcc[0]),
                 xy=(sample_ext.index[1], sample_ext[1]), arrowprops=prop2)
    plt.annotate("Overshoot", xytext=(sample_dcc.index[0], sample_dcc[0] - 5),
                 xy=(sample_ext.index[1], sample_ext[1]))

    plt.annotate("", xytext=(sample_ext.index[1], sample_ext[1]),
                 xy=(sample_dcc.index[1], sample_dcc[1]), arrowprops=prop1)
    plt.annotate("", xytext=(sample_dcc.index[1], sample_dcc[1]),
                 xy=(sample_ext.index[2], sample_ext[2]), arrowprops=prop2)

    plt.annotate("Overshoot", xytext=(md.date2num(sample_dcc.index[1]) + 2,␣
↪sample_dcc[1] + 5),
                 xy=(sample_ext.index[2], sample_ext[2]))

    plt.annotate("", xytext=(sample_ext.index[0], sample_dcc[0]),
                 xy=(sample_dcc.index[0], sample_dcc[0]), arrowprops=prop5)

    plt.annotate("", xytext=(sample_ext.index[1], sample_ext[1]),
                 xy=(sample_dcc.index[1], sample_ext[1]), arrowprops=prop5)

    plt.annotate("", xytext=(sample_dcc.index[1], sample_ext[1]),
                 xy=(sample_dcc.index[1], sample_dcc[1]), arrowprops=prop5)

    plt.annotate("Theta", xytext=(md.date2num(sample_dcc.index[1]) + 0.2,␣
↪sample_ext[1] + 5),
                 xy=(sample_dcc.index[1], sample_dcc[1]), rotation=90)

    plt.annotate("", xytext=(sample_dcc.index[1], sample_ext[1]),
                 xy=(sample_ext.index[2], sample_ext[1]), arrowprops=prop5)

    plt.annotate("", xytext=(sample_ext.index[2], sample_ext[1]),
                 xy=(sample_ext.index[2], sample_ext[2]), arrowprops=prop5)

    ax.axvspan(sample_ext.index[0], sample_ext.index[1], color='red', alpha=0.
↪25)
    ax.axvspan(sample_ext.index[1], sample_ext.index[2], color='green', alpha=0.
↪25)

    ax.text((md.date2num(sample_ext.index[0]) + 2), y=1420, s='Downward Trend')
    ax.text((md.date2num(sample_ext.index[1]) + 2), y=1420, s='Upward Trend')


if __name__ == '__main__':
```

```
    print('Please import this file as a module.')  # %%
```

hidden_markov_model.py

[ ]:
```
import pandas as pd
import numpy as np
from hmmlearn import hmm
import matplotlib.pyplot as plt
plt.style.use('seaborn')

def fit_hmm(n_components: int, price: pd.Series, indicator: pd.Series, ticker:␣
↪str, plot: bool =False, verbose: bool = False) -> tuple[pd.Series, hmm.
↪GaussianHMM]:
    """Fits a Hidden Markov model to the data and predicts regimes on it.␣
↪Optionally makes a plot.

    Args:
        n_components (int): number of regimes
        price (pd.Series): price series of the instrument
        indicator (pd.Series): indicator series we wish to fit the model on
        ticker (str): ticker of the instrument
        plot (bool, optional): whether the regimes need to be plotted. Defaults␣
↪to False.
        verbose (bool, optional): whether debugging output needs to be printed.␣
↪Defaults to False.

    Returns:
        tuple[pd.Series,hmm.GaussianHMM]: the predicted regimes and the HMM␣
↪model
    """

    X = indicator.to_numpy().reshape(-1,1)

    models, scores = [], []
    for idx in range(10):
        model = hmm.GaussianHMM(n_components=n_components,␣
↪covariance_type="full", n_iter=1000,
            random_state=idx)

        model.fit(X)
        models.append(model)
        scores.append(model.score(X))

    model = models[np.argmax(scores)]

    regimes = pd.Series(model.predict(X))
```

```
    regimes.index = indicator.index

    regimes = standardize_regime_labels(regimes,verbose=verbose)

    if plot:
        fig, ax = plt.subplots()
        price.plot(ax=ax, color='black')
        clr = {0:'grey',1:'red',2:'green'}

        for time_start, time_end, regime in zip(regimes.index[:-1], regimes.
 ↪index[1:], regimes.values[:-1]):
            ax.axvspan(time_start,time_end, alpha=0.8, color=clr[regime])
        ax.set_title(f"regimes for {ticker}")
        ax.set_ylabel("price")
        plt.show()

    return regimes, model


def standardize_regime_labels(regimes: pd.Series, verbose: bool = True) -> pd.
 ↪Series:
    """
    This is helper function to standardize regime labels. It is based on the␣
 ↪assumption
    that regime 1 (index 0) is the normal regime and in the long term, the␣
 ↪market is mostly in the
    normal regime.
    :param regimes: A series indicating the regimes and indexed by a datetime
    :param verbose:
    :return:
    """
    start = regimes.index[0]
    initial_regime = regimes[0]
    prev_regime = regimes[0]
    prev_time = regimes.index[0]
    total_duration_in_initial_regime = 0

    if len(np.unique(regimes)) == 1:
        total_duration_in_initial_regime = (regimes.index[-1] - regimes.
 ↪index[0]).total_seconds()
    else:
        for time, regime in regimes[1:].items():
            if regime == initial_regime:
                total_duration_in_initial_regime += (time - prev_time).
 ↪total_seconds()
            prev_time = time
```

```
            prev_regime = regime

    total_duration = (regimes.index[-1] - regimes.index[0]).total_seconds()

    if verbose:
        print('Total duration of time: {}'.format(total_duration))
        print('Total duration spent in Regime {}: {}'.format(initial_regime,␣
↪total_duration_in_initial_regime))
        print('Proportion of time spent in Regime {}: {}'.
↪format(initial_regime, total_duration_in_initial_regime / total_duration))

    # if (initial_regime == 0) and ((total_duration_in_initial_regime /␣
↪total_duration) <= 0.5):
    if ((initial_regime == 0) and ((total_duration_in_initial_regime /␣
↪total_duration) <= 0.5)) or ((initial_regime == 1) and␣
↪((total_duration_in_initial_regime / total_duration) >= 0.5)):
        if verbose:
            print('Flipping labels between regimes.')
        regimes = 1 - regimes
    return regimes

def make_regime_plots(regimes: pd.Series, tmv: pd.Series, T: pd.Series, ticker:␣
↪str, set_: str = 'test'):
    """Makes the normalized TMV versus normalized T plots, separated by regime.

    Args:
        regimes (pd.Series): regimes
        tmv (pd.Series): tmv
        T (pd.Series): T
        ticker (str): ticker
    """

    regime_df = (pd.DataFrame([regimes, tmv, T]).T)
    regime_df.columns = ['Regime','TMV','T']
    regime_df.Regime = regime_df.Regime.astype('category')

    # normalize
    regime_df[['TMV','T']] = (regime_df[['TMV','T']] - regime_df[['TMV','T']].
↪min())/(regime_df[['TMV','T']].max() - regime_df[['TMV','T']].min())

    fig, ax = plt.subplots(figsize=(10,5))
    colors = {0:'grey', 1:'red'}
    for c in colors:
        ax.scatter(regime_df[regime_df.Regime == c]['T'], regime_df[regime_df.
↪Regime == c]['TMV'], c=colors[c],label=f'regime {c}')
```

```
    ax.set_title(f"Regimes for {ticker} on the {set_} set")
    ax.set_xlabel("normalized T")
    ax.set_ylabel("normalized TMV")
    plt.legend()
    plt.show()

#%%
```

kmeans.py

[ ]:
```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans


class KMeansCluster():
    def __init__(self, k: int = 2, X_train=None, X_test=None, random_state=25):
        self.k = k
        self.model = KMeans(n_clusters=self.k, init='random',␣
 ↪random_state=random_state)
        self.X_train = X_train
        self.X_test = X_test
        self.labels_ = None
        self.y_train_pred = None
        self.y_test_pred = None
        self.sizes_train_df = None
        self.sizes_test_df = None

    def assign(self, X_train, X_test=None):
        self.X_train = X_train
        self.X_test = X_test

    def fit(self, X_train=None):
        if X_train is not None:
            self.assign(X_train)

        if self.X_train is not None:
            self.model.fit(self.X_train)
            self.labels_ = self.model.labels_
            self.sizes_train_df = pd.DataFrame.from_dict({
                'CLUSTER': [i for i in range(self.k)],
                'CLUSTER_SIZE': [np.sum(self.labels_ == i) for i in range(self.
 ↪k)],
            }).set_index('CLUSTER')
        else:
            print(f"Please assign a training set before fitting!")
```

```
    def predict(self):
        self.fit()
        self.y_train_pred = self.model.predict(self.X_train)
        if self.X_test is not None:
            self.y_test_pred = self.model.predict(self.X_test)
            self.sizes_test_df = pd.DataFrame.from_dict({
                'CLUSTER': [i for i in range(self.k)],
                'CLUSTER_SIZE': [np.sum(self.y_test_pred == i) for i in␣
 ↪range(self.k)],
            }).set_index('CLUSTER')
        else:
            self.y_test_pred = None
```

logistic_regression.py

[ ]:
```
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression

def train_LR(X, y):
    """
    params: X-> X is a pandas dataframe of features
            y-> y is a list of labels

    return: returns a learned Naive Bayes Classifier

    comments: Right now we are only considering gaussian kernels
    """
    model = LogisticRegression(penalty='none')
    model.fit( X, y )
    return model

def get_predict_probs_LR( model, X ):
    """
    params: model-> A fitted Naive Bayes Model
            X -> A pd dataframe of test variables

    return: Returns a 2-D matrix of n * classes size which has␣
 ↪predict_probabilities
    """

    return model.predict_proba(X)

def predict_LR( model, X, epsilon = 0.5 ):
    """
    params: model-> A fitted Naive Bayes Model
            X -> A pd dataframe of test variables
```

```
    return: Returns a list of predictions (highest probability) for each test point
    """


    if( epsilon != 0.5 ):
        ans = np.zeros(len(X))
        probs = get_predict_probs_LR( model, X )
        return np.where(probs[:, 1] >= epsilon, 1, 0)

    return model.predict(X)

def do_all_LR(X_train, y_train, X_valid, epsilon = 0.5):
    """
    Trains a naive bayes model with y_train as regimes after filteration and X_train as one of the three DC indicators

    params-> X_train: One of the three DC indicators, or a time series
             y_train: Regimes output from hmm model on the training data after filteration
             X_valid: Validation test for X, same DC indicator as X_train
    """

    model = train_LR( X_train, y_train )
    preds = predict_LR( model, X_valid, epsilon )
    return preds
```

NaiveBayesClassifier.py

[ ]:
```
import pandas as pd
import numpy as np
from sklearn.naive_bayes import GaussianNB


def train_NBC(X, y):
    """
    params: X-> X is a pandas dataframe of features
            y-> y is a list of labels

    return: returns a learned Naive Bayes Classifier

    comments: Right now we are only considering gaussian kernels
    """
    model = GaussianNB()
    model.fit( X, y )
    return model
```

```
def get_predict_probs_NBC( model, X ):
    """
    params: model-> A fitted Naive Bayes Model
            X -> A pd dataframe of test variables

    return: Returns a 2-D matrix of n * classes size which has␣
 ↪predict_probabilities
    """

    return model.predict_proba(X)

def predict_NBC( model, X, epsilon = 0.5 ):
    """
    params: model-> A fitted Naive Bayes Model
            X -> A pd dataframe of test variables

    return: Returns a list of predictions (highest probability) for each test␣
 ↪point
    """



    if( epsilon != 0.5 ):
        ans = np.zeros(len(X))
        probs = get_predict_probs_NBC( model, X )
        return np.where(probs[:, 1] >= epsilon, 1, 0)
        # for i in range(len(probs)):
        #     if probs[i][1] >= epsilon:
        #         ans[i] = 1
        # return ans

    return model.predict(X)

def do_all_NBC(X_train, y_train, X_valid, epsilon = 0.5):
    """
    Trains a naive bayes model with y_train as regimes after filteration and␣
 ↪X_train as one of the three DC indicators

    params-> X_train: One of the three DC indicators, or a time series
             y_train: Regimes output from hmm model on the training data after␣
 ↪filteration
             X_valid: Validation test for X, same DC indicator as X_train
    """

    model = train_NBC( X_train, y_train )
    preds = predict_NBC( model, X_valid, epsilon )
    return preds
```

svm.py

[ ]:
```
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC


def train_SVM(X, y):
    """
    params: X-> X is a pandas dataframe of features
            y-> y is a list of labels

    return: returns a learned SVM Classifier
    """
    model = make_pipeline(StandardScaler(), SVC(probability=True))
    model.fit(X, y)
    return model


def get_predict_probs_SVM(model, X):
    """
    params: model-> A fitted SVM Model
            X -> A pd dataframe of test variables

    return: Returns a 2-D matrix of n * classes size which has␣
 ↪predict_probabilities
    """
    return model.predict_proba(X)


def predict_SVM(model, X, epsilon=0.5):
    """
    params: model-> A fitted Naive Bayes Model
            X -> A pd dataframe of test variables

    return: Returns a list of predictions (highest probability) for each test␣
 ↪point
    """

    if (epsilon != 0.5):
        ans = np.zeros(len(X))
        probs = get_predict_probs_SVM(model, X)
        return np.where(probs[:, 1] >= epsilon, 1, 0)

    return model.predict(X)
```

```
def do_all_SVM(X_train, y_train, X_valid, epsilon=0.5):
    """
    Trains a SVM model with y_train as regimes after filteration and X_train as one of the three DC indicators

    params-> X_train: One of the three DC indicators, or a time series
             y_train: Regimes output from hmm model on the training data after filteration
             X_valid: Validation test for X, same DC indicator as X_train
    """
    model = train_SVM(X_train, y_train)
    preds = predict_SVM(model, X_valid, epsilon)
    return preds
```

text_preprocessing.py

[ ]:
```
import gensim
from gensim.corpora import Dictionary
from gensim.matutils import corpus2dense, corpus2csc
from gensim.models import TfidfModel
from gensim.models.nmf import Nmf
from gensim.models.coherencemodel import CoherenceModel
import numpy as np
import pandas as pd
import nltk

nltk.download("stopwords")
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS

FOMC_STOP_WORDS = ["federal", "reserve", "board", "meeting", "committee", "minutes", "members"]


def remove_names_from_minutes(text: str):
    """
    This function removes all names from the start of FED Minutes by relying on
    the fact that the phrases 'the manager' and 'unanimous' tend to appear at
    the end of the initial string of names.

    @param tet(str): text which needs to have names removed from the start
    @returns res(str): portion of text after first occurence of 'the manager'
                       or 'unanimous'
    """
    text = text.lower()
    split_by = ''
```

```
    if 'the manager' in text and 'unanimous' in text:
        if text.index('the manager') > text.index('unanimous'):
            split_by = 'unanimous'
        else:
            split_by = 'the manager'
    elif 'the manager' in text:
        split_by = 'the manager'
    elif 'unanimous' in text:
        split_by = 'unanimous'
    else:
        raise ValueError('Neither in text!')

    res = text.split(split_by)[1]
    return res


def tokenizer_wo_stopwords(text: str):
    """
    This function prepares raw text by tokenizing it and removing all stop
    words (based on nltk stopwords).

    @param text(str): raw text which needs to be prepared for analysis
    @return res(str): string representation of text without stopwords
    """
    tokens = nltk.word_tokenize(text)
    words = [word.lower() for word in tokens if word.isalpha()]
    words_wo_stop = [w.lower() for w in words if
                     w.lower() not in ENGLISH_STOP_WORDS and w.lower() not in␣
 ↪FOMC_STOP_WORDS]
    res = ' '.join(words_wo_stop)
    return res


class TF_IDF():
    def __init__(self, X_train: pd.Series = None, X_test: pd.Series = None):
        self.X_train = X_train
        self.X_test = X_test

        # Attributes needed for manual TF-IDF computations
        self.def_vectorizer = None
        self.tfidf_manual_train = None
        self.tfidf_manual_test = None

        # Attributes needed for gensim TF-IDF computations
        self.dict_gensim_statements = None
        self.tfidf_model_gensim = None
        self.tfidf_statements_train = None
```

```
        self.tfidf_statements_test = None
        self.tfidf_gensim_train = None
        self.tfidf_gensim_test = None

    def assign(self, X_train: pd.Series = None, X_test: pd.Series = None):
        self.X_train = X_train
        self.X_test = X_test

    def fit_manual_helper(self, train: bool = True):
        """
        This function manually computes the TF-IDF values for a column of train
        OR test documents, to avoid the incorrect computations performed by
        sklearn's native implementation.

        @param train: flag determining if function will fit/transform train
                      data, or only fit vectorizer to test data
        """
        if train:
            text = self.X_train
        else:
            text = self.X_test

        try:
            assert text is not None
        except Exception as e:
            print(f"assign() train/test data before fitting!")
            return

        # Get number of documents
        n_docs = text.shape[0]

        # Generate bag-of-words matrix
        if train:
            self.def_vectorizer = CountVectorizer(token_pattern='[a-zA-Z]+')
            word_bow_matrix = self.def_vectorizer.fit_transform(text)
        else:
            word_bow_matrix = self.def_vectorizer.transform(text)

        word_bow_df = pd.DataFrame(
            word_bow_matrix.toarray(),
            columns=self.def_vectorizer.get_feature_names_out()
        )

        # Create TF matrix
        tf_df = word_bow_df / word_bow_df.sum(axis=1).values.reshape(n_docs, 1)

        # Compute IDF values
```

```
        idf = np.log(n_docs / (word_bow_df / word_bow_df.values).sum(axis=0))

        # Manually create TF-IDF matrix
        if train:
            self.tfidf_manual_train = tf_df * idf
        else:
            self.tfidf_manual_test = tf_df * idf

    def fit_manual(self):
        """
        This function fits the manual TF-IDF model to train data and generates
        the values for the test data by calling the previously-defined helper
        function consecutively on train and test data.
        """
        self.fit_manual_helper(train=True)
        self.fit_manual_helper(train=False)

    def fit_gensim_helper(self, train: bool = True):
        """
        This function uses gensim to compute the TF-IDF values for a column of
        train or test documents, to avoid the incorrect computations performed
        by sklearn's native implementation.

        @param train: flag determining if function will fit/transform train
                      data, or only fit vectorizer to test data
        """
        if train:
            text = self.X_train
        else:
            text = self.X_test

        try:
            assert text is not None
        except Exception as e:
            print(f"assign() train/test data before fitting!")
            return

        gensim_statements = text.apply(lambda x: x.split(" ")).tolist()

        if train:
            self.dict_gensim_statements = Dictionary(gensim_statements)

        bow_gensim_statements = [self.dict_gensim_statements.doc2bow(d) for d␣
↪in gensim_statements]

        if train:
            self.tfidf_model_gensim = TfidfModel(bow_gensim_statements)
```

```
        tfidf_statements = self.tfidf_model_gensim[bow_gensim_statements]

        if train:
            self.tfidf_statements_train = tfidf_statements
        else:
            self.tfidf_statements_test = tfidf_statements

        num_terms = len(self.dict_gensim_statements.keys())
        num_docs = len(tfidf_statements)

        if train:
            self.tfidf_gensim_train = corpus2dense(
                tfidf_statements,
                num_terms,
                num_docs
            ).T
        else:
            self.tfidf_gensim_test = corpus2dense(
                tfidf_statements,
                num_terms,
                num_docs
            ).T

    def fit_gensim(self):
        """
        This function fits the gensim TF-IDF model to train data and generates
        the values for the test data by calling the previously-defined helper
        function consecutively on train and test data.
        """
        self.fit_gensim_helper(train=True)
        self.fit_gensim_helper(train=False)


if __name__ == "__main__":
    print(f"Please import this module as a library!")
```

topic_modeling.py

[ ]:
```
from sklearn.model_selection import KFold
import numpy as np
from gensim.models.nmf import Nmf
from gensim.models.coherencemodel import CoherenceModel


class TopicModel():
    """_summary_
```

```
    Class to generate
    1. topic probability distribution for a set of documents
    2. assign the topic with maximum probability to each document in the set
    """

    def __init__(
            self,
            tfidf_mat,
            dictionary,
            statements,
            bow_test=None,
            crossval=True,
            num_topics_list=[2, 5, 7, 10],
            num_topic=10,
            cv_score="c_v",
    ):
        """_summary_
        Constructor
        Args:
            tfidf_mat (_type_): TFIDF matrix
            dictionary (_type_): Gensim dictionary
            statements (_type_): documents
            bow_test (_type_): test bag of words
            crossval (bool, optional): If set to True, perform cross validation␣
↪to get number of topics. Defaults to True.
            num_topics_list (list, optional): List of number of topics to CV␣
↪over. Defaults to [2, 5, 7, 10].
            num_topic (int, optional): If no cross validation, use this for␣
↪model generation. Defaults to 10.
            cv_score (str, optional): Coherence score to use as metric for CV.␣
↪Defaults to "c_v".
        """
        self.tfidf_mat = tfidf_mat
        self.dictionary = dictionary
        self.statements = statements
        self.bow_test = bow_test
        self.crossval = crossval
        self.num_topic = num_topic
        self.cv_score = cv_score
        self.num_topics_list = num_topics_list
        self.num_docs = len(self.statements)

    def model(self):
        if self.crossval:
            if self.num_topics_list is None:
                self.num_topics_list = [2, 5, 7, 10]
            self.num_topic = self.cross_val(
```

```
                n_splits=5)
        self.cv_model = Nmf(
            corpus=self.tfidf_mat,
            id2word=self.dictionary,
            num_topics=self.num_topic,
            random_state=42
        )

    def cross_val(self, n_splits=5):
        kf = KFold(n_splits=n_splits)
        rank_dict = dict()
        for num_topics in self.num_topics_list:
            avg_coherence = 0
            for train_idx, _ in kf.split(self.tfidf_mat):
                train_stmts = [self.tfidf_mat[i] for i in train_idx]
                model = Nmf(
                    train_stmts,
                    num_topics=num_topics,
                    id2word=self.dictionary,
                    passes=5,
                    random_state=42,
                )
                coherence_model = CoherenceModel(
                    model=model,
                    texts=self.statements,
                    dictionary=self.dictionary,
                    coherence=self.cv_score,
                )
                coherence = coherence_model.get_coherence()
                avg_coherence += coherence
            avg_coherence /= kf.get_n_splits()
            rank_dict[num_topics] = avg_coherence
            # print(f"Num Topics: {num_topics}, Average Coherence:␣
↪{avg_coherence:.4f}")
        self.cv_topics_list = list(
            sorted(rank_dict.items(), key=lambda item: item[1], reverse=True))
        return self.cv_topics_list[0][0]

    def generate_topic_distribution_matrix(self):
        doc_mat = np.zeros(shape=(self.num_docs, self.num_topic))
        for i in range(self.num_docs):
            topic_list = self.cv_model.get_document_topics(
                self.tfidf_mat[i], minimum_probability=0)
            for tup in topic_list:
                doc_mat[i][tup[0]] = tup[1]
        self.doc_mat = doc_mat
        return doc_mat
```

```
    def assign_topic_to_documents(self):
        topic_mat = np.zeros(self.num_docs, dtype="int64")
        for i in range(self.num_docs):
            topic_list = self.cv_model.get_document_topics(
                self.tfidf_mat[i],
                minimum_probability=0
            )
            topic_list = sorted(topic_list, key=lambda x: x[1], reverse=True)
            topic_mat[i] = topic_list[0][0]
        self.topic_mat = topic_mat
        return topic_mat

    def fit(self):
        self.model()
        self.generate_topic_distribution_matrix()
        self.assign_topic_to_documents()

    def predict(self):
        pdf_test = []

        for i in range(len(self.bow_test)):
            topic_pdf = self.cv_model[self.bow_test][i]
            topic_pdf = {topic: pdf for topic, pdf in topic_pdf}
            cur_doc_pdf = []
            for i in range(self.num_topic):
                if i in topic_pdf:
                    cur_doc_pdf.append(topic_pdf[i])
                else:
                    cur_doc_pdf.append(0.0)
            pdf_test.append(cur_doc_pdf)

        self.pdf_test = np.array(pdf_test)

    def fit_predict(self):
        self.fit()
        if self.bow_test is not None:
            self.predict()
```

trading_strategy.py

[ ]:
```
import pandas as pd
import numpy as np
import modules.generate_data as gd

def sharpe(df):
    return (df.mean()*252/(df.std()*np.sqrt(252)))
```

```
def sortino(df):
    sd = np.sqrt(np.mean(df[df<0].values*df[df<0].values))
    return (df.mean()*252/(sd*np.sqrt(252)))

def mdd(df):
    return ((df.cumsum()-df.cumsum().cummax()).min())

def get_metrics(df):
    metrics = {'pnl':round((1+df).product() - 1,2),
               'sharpe':sharpe(df).round(2),
               'sortino':sortino(df).round(2),
               'volatility':round(df.std(),4),
               'mdd':mdd(df)}
    return pd.DataFrame(metrics, index=[''])

def RSI(price, lookback = 14):
    # based on: https://blog.quantinsti.com/
 ↪build-technical-indicators-in-python/
    ret = price.diff()

    high = ret.clip(lower=0)
    low = -ret.clip(upper=0)

    avg_up = high.ewm(alpha = 1/lookback, min_periods = lookback).mean()
    avg_down = low.ewm(alpha = 1/lookback, min_periods = lookback).mean()

    return (100 - (100/(1 + (avg_up / avg_down))).dropna()).shift(1).dropna()

##############################################################################
#### Trading strategy as explained in the book
##############################################################################

def regime_to_sign(val):
    if( val == 1):
        return 1
    elif( val == 0 ):
        return -1
    else:
        '''Never happens'''
        return 0

def strategy_regime_dependent(data, init_cap = 1, strat = "JC1", threshold = 0.
 ↪5):
    """
    JC1 strategy as explained in the book Chapter 6 -> Mean reversion during␣
 ↪normal time, momentum during abnormal time.
```

```
    It returns a dataframe after implementing JC1 strategy which is based on␣
↪regime change
    We get new columns at every time which are: position, asset capital, bank␣
↪capital, total capital and returns


    TODO: We assume that open to close and close to open times are same, useful␣
↪in sharpe calculation

    params-> data: Output of gd.generate_dataset_with_columns
             init_cap: Inital capital in the strategy
             strat = Name of strat for adding columns
             threshold: When to take buy/sell decisions

    returns-> pd.DataFrame which has columns appended to data
                'daily_ret_strat', 'position_strat', 'asset_cap_strat',␣
↪'bank_cap_strat', 'total_cap_strat'
    """

    '''All values are after trading'''
    position = 'position_'+strat # Position in asset at this time
    daily_ret = 'daily_ret_'+strat # Return on total_cap
    asset_cap = 'asset_cap_'+strat # Price of assets I have
    bank_cap = 'bank_cap_'+strat # Capital which I don't have invested
    total_cap = 'total_cap_'+strat # Sum of above two

    d = data.copy()

    d[daily_ret] = 0
    d[position] = 0
    d[asset_cap] = 0
    d[bank_cap] = 0
    d[bank_cap][0] = init_cap
    d[total_cap] = d[bank_cap]

    for i in range( 1,len(d) ):
        if( d['regime'][i] == -1 ):
            '''Don't do anything, regime detection has not started'''
            d[position][i] = d[position][i-1]
            d[asset_cap][i] = d[asset_cap][i-1]
            d[bank_cap][i] = d[bank_cap][i-1]

        else:
            if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= threshold ) ):
                '''I go against the market with all my money'''
                d[position][i] = regime_to_sign(d['regime'][i]) * np.
↪sign(d['TMV'][i]) * d[total_cap][i-1] / d['price'][i]
```

```
                d[asset_cap][i] = d[position][i] * d['price'][i]
                d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
            elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < threshold ) ):
                '''No position and threshold not crossed'''
                d[position][i] = d[position][i-1]
                d[asset_cap][i] = d[position][i] * d['price'][i]
                d[bank_cap][i] = d[bank_cap][i-1]
            elif( ( abs(d[position][i-1]) > 0) ):
                if( (d['regime'][i-1] == d['regime'][i]) and (d['type'][i] not␣
↪in ['DCC','EXT_DCC'] ) ):
                    '''No Action to be taken'''
                    d[position][i] = d[position][i-1]
                    d[asset_cap][i] = d[position][i] * d['price'][i]
                    d[bank_cap][i] = d[bank_cap][i-1]
                else:
                    '''I take the opposite position here'''
    #                 d['debug'][i] = 'Hello'
                    d[position][i] = 0
                    d[asset_cap][i] = 0
                    d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]
        d[total_cap][i] = d[bank_cap][i] + d[asset_cap][i]
        d[daily_ret][i] = (d[total_cap][i] - d[total_cap][i-1])/
↪d[total_cap][i-1]

    return d

    #     if( d['regime'][i] == 0 ):
    #         '''Normal Regime - Mean Reverting'''
    #         if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= threshold )␣
↪):
    #             '''I go against the market with all my money'''
    #             d[position][i] = -1 * np.sign(d['TMV'][i]) *␣
↪d[total_cap][i-1] / d['price'][i]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
    #         elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < threshold )␣
↪):
    #             '''No position and threshold not crossed'''
    #             d[position][i] = d[position][i-1]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[bank_cap][i-1]
    #         elif( ( abs(d[position][i-1]) > 0) ):
    #             if( (d['regime'][i-1] == 0) and (d['type'][i] not in␣
↪['DCC','EXT_DCC'] ) ):
    #                 '''No Action to be taken'''
    #                 d[position][i] = d[position][i-1]
```

```
    #                 d[asset_cap][i] = d[position][i] * d['price'][i]
    #                 d[bank_cap][i] = d[bank_cap][i-1]
    #             else:
    #                 '''I take the opposite position here'''
    # #                 d['debug'][i] = 'Hello'
    #                 d[position][i] = 0
    #                 d[asset_cap][i] = 0
    #                 d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]

    #     elif( d['regime'][i] == 1 ):
    #         '''Abnormal regime, momentum trading'''
    #         if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= threshold )
 ↪):
    #             '''I follow the market with all my money'''
    #             d[position][i] = np.sign(d['TMV'][i]) * d[total_cap][i-1] /
 ↪d['price'][i]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
    #         elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < threshold )
 ↪):
    #             '''No position and threshold not crossed'''
    #             d[position][i] = d[position][i-1]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[bank_cap][i-1]
    #         elif( ( abs(d[position][i-1]) > 0) ):
    #             if( (d['regime'][i-1] == 1) and (d['type'][i] not in
 ↪['DCC','EXT_DCC'] ) ):
    #                 '''No Action to be taken'''
    #                 d[position][i] = d[position][i-1]
    #                 d[asset_cap][i] = d[position][i] * d['price'][i]
    #                 d[bank_cap][i] = d[bank_cap][i-1]
    #             else:
    #                 '''I take the opposite position here'''
    #                 d[position][i] = 0
    #                 d[asset_cap][i] = 0
    #                 d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]
    #     else:
    #         d[position][i] = d[position][i-1]
    #         d[asset_cap][i] = d[asset_cap][i-1]
    #         d[bank_cap][i] = d[bank_cap][i-1]


def strategy_control(data, init_cap = 1, strat = "JC1", threshold = 0.5):
    """
    CT1 strategy as explained in the book Chapter 6 -> Mean reversion all the
 ↪time.
```

```
   It returns a dataframe after implementing JC1 strategy which is based on␣
↪regime change
   We get new columns at every time which are: position, asset capital, bank␣
↪capital, total capital and returns

   TODO: We assume that open to close and close to open times are same, useful␣
↪in sharpe calculation

   params-> data: Output of gd.generate_dataset_with_columns or any other␣
↪strategy
            init_cap: Inital capital in the strategy
            strat = Name of strat for adding columns
            threshold: When to take buy/sell decisions

   returns-> pd.DataFrame which has columns appended to data
               'daily_ret_strat', 'position_strat', 'asset_cap_strat',␣
↪'bank_cap_strat', 'total_cap_strat'
   """

   '''All values are after trading'''
   position = 'position_'+strat # Position in asset at this time
   daily_ret = 'daily_ret_'+strat # Return on total_cap
   asset_cap = 'asset_cap_'+strat # Price of assets I have
   bank_cap = 'bank_cap_'+strat # Capital which I don't have invested
   total_cap = 'total_cap_'+strat # Sum of above two

   d = data.copy()

   d[daily_ret] = 0
   d[position] = 0
   d[asset_cap] = 0
   d[bank_cap] = 0
   d[bank_cap][0] = init_cap
   d[total_cap] = d[bank_cap]

   for i in range( 1,len(d) ):
       if( d['regime'][i] == -1 ):
           '''Don't do anything, regime detection has not started'''
           d[position][i] = d[position][i-1]
           d[asset_cap][i] = d[asset_cap][i-1]
           d[bank_cap][i] = d[bank_cap][i-1]

       else:
           if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= threshold ) ):
               '''I go against the market with all my money'''
               d[position][i] = -1 * np.sign(d['TMV'][i]) * d[total_cap][i-1] /
↪ d['price'][i]
```

```
                d[asset_cap][i] = d[position][i] * d['price'][i]
                d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
            elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < threshold ) ):
                '''No position and threshold not crossed'''
                d[position][i] = d[position][i-1]
                d[asset_cap][i] = d[position][i] * d['price'][i]
                d[bank_cap][i] = d[bank_cap][i-1]
            elif( ( abs(d[position][i-1]) > 0) ):
                if( (d['type'][i] not in ['DCC','EXT_DCC'] ) ):
                    '''No Action to be taken'''
                    d[position][i] = d[position][i-1]
                    d[asset_cap][i] = d[position][i] * d['price'][i]
                    d[bank_cap][i] = d[bank_cap][i-1]
                else:
                    '''I take the opposite position here'''
    #                 d['debug'][i] = 'Hello'
                    d[position][i] = 0
                    d[asset_cap][i] = 0
                    d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]
        d[total_cap][i] = d[bank_cap][i] + d[asset_cap][i]
        d[daily_ret][i] = (d[total_cap][i] - d[total_cap][i-1])/
↪d[total_cap][i-1]

    return d

def strategy_control2(data, init_cap = 1, strat = "control2", threshold = 0.5):
    """
    Momentum trading all the time.
    It returns a dataframe after implementing JC1 strategy which is based on␣
↪regime change
    We get new columns at every time which are: position, asset capital, bank␣
↪capital, total capital and returns

    TODO: We assume that open to close and close to open times are same, useful␣
↪in sharpe calculation

    params-> data: Output of gd.generate_dataset_with_columns or any other␣
↪strategy
             init_cap: Inital capital in the strategy
             strat = Name of strat for adding columns
             threshold: When to take buy/sell decisions

    returns-> pd.DataFrame which has columns appended to data
                'daily_ret_strat', 'position_strat', 'asset_cap_strat',␣
↪'bank_cap_strat', 'total_cap_strat'
    """
```

```
    '''All values are after trading'''
    position = 'position_'+strat # Position in asset at this time
    daily_ret = 'daily_ret_'+strat # Return on total_cap
    asset_cap = 'asset_cap_'+strat # Price of assets I have
    bank_cap = 'bank_cap_'+strat # Capital which I don't have invested
    total_cap = 'total_cap_'+strat # Sum of above two

    d = data.copy()

    d[daily_ret] = 0
    d[position] = 0
    d[asset_cap] = 0
    d[bank_cap] = 0
    d[bank_cap][0] = init_cap
    d[total_cap] = d[bank_cap]

    for i in range( 1,len(d) ):
        if( d['regime'][i] == -1 ):
            '''Don't do anything, regime detection has not started'''
            d[position][i] = d[position][i-1]
            d[asset_cap][i] = d[asset_cap][i-1]
            d[bank_cap][i] = d[bank_cap][i-1]

        else:
            if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= threshold ) ):
                '''I go against the market with all my money'''
                d[position][i] = np.sign(d['TMV'][i]) * d[total_cap][i-1] /␣
↪d['price'][i]
                d[asset_cap][i] = d[position][i] * d['price'][i]
                d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
            elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < threshold ) ):
                '''No position and threshold not crossed'''
                d[position][i] = d[position][i-1]
                d[asset_cap][i] = d[position][i] * d['price'][i]
                d[bank_cap][i] = d[bank_cap][i-1]
            elif( ( abs(d[position][i-1]) > 0) ):
                if( (d['type'][i] not in ['DCC','EXT_DCC'] ) ):
                    '''No Action to be taken'''
                    d[position][i] = d[position][i-1]
                    d[asset_cap][i] = d[position][i] * d['price'][i]
                    d[bank_cap][i] = d[bank_cap][i-1]
                else:
                    '''I take the opposite position here'''
    #                 d['debug'][i] = 'Hello'
                    d[position][i] = 0
                    d[asset_cap][i] = 0
```

```
                    d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]
        d[total_cap][i] = d[bank_cap][i] + d[asset_cap][i]
        d[daily_ret][i] = (d[total_cap][i] - d[total_cap][i-1])/
↪d[total_cap][i-1]

    return d

    #     init_cap = 1
    # strat = 'JC2'



    # '''All values are after trading'''
    # position = 'position_'+strat # Position in asset at this time
    # daily_ret = 'daily_ret_'+strat # Return on total_cap
    # asset_cap = 'asset_cap_'+strat # Price of assets I have
    # bank_cap = 'bank_cap_'+strat # Capital which I don't have invested
    # total_cap = 'total_cap_'+strat # Sum of above two


    # d[daily_ret] = 0
    # d[position] = 0 # This is after trading on this event (time point)
    # d[asset_cap] = 0
    # d[bank_cap] = 0
    # d[bank_cap][0] = init_cap
    # d[total_cap] = d[bank_cap]

    # '''Threshold for TMV'''
    # thresh = 0.5

    # for i in range( 1,len(d) ):
    #     if( d['regime'][i] == 0 ):
    #         '''Normal Regime - Mean Reverting'''
    #         if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= thresh ) ):
    #             '''I go against the market with all my money'''
    #             d[position][i] = -1 * np.sign(d['TMV'][i]) *␣
↪d[total_cap][i-1] / d['price'][i]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
    #         elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < thresh ) ):
    #             '''No position and threshold not crossed'''
    #             d[position][i] = d[position][i-1]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[bank_cap][i-1]
    #         elif( ( abs(d[position][i-1]) > 0) ):
    #             if( (d['regime'][i-1] == 0) and (d['type'][i] not in␣
↪['DCC','EXT_DCC'] ) ):
```

```
    #                 '''No Action to be taken'''
    #                 d[position][i] = d[position][i-1]
    #                 d[asset_cap][i] = d[position][i] * d['price'][i]
    #                 d[bank_cap][i] = d[bank_cap][i-1]
    #             else:
    #                 '''I take the opposite position here'''
    # #                 d['debug'][i] = 'Hello'
    #                 d[position][i] = 0
    #                 d[asset_cap][i] = 0
    #                 d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]

    #     elif( d['regime'][i] == 1 ):
    #         if( (d[position][i-1] == 0) and (abs(d['TMV'][i]) >= thresh ) ):
    #             '''I go against the market with all my money'''
    #             d[position][i] = -1 * np.sign(d['TMV'][i]) *
↪d[total_cap][i-1] / d['price'][i]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[total_cap][i-1] - d[asset_cap][i]
    #         elif( (d[position][i-1] == 0) and (abs(d['TMV'][i]) < thresh ) ):
    #             '''No position and threshold not crossed'''
    #             d[position][i] = d[position][i-1]
    #             d[asset_cap][i] = d[position][i] * d['price'][i]
    #             d[bank_cap][i] = d[bank_cap][i-1]
    #         elif( ( abs(d[position][i-1]) > 0) ):
    #             if( (d['regime'][i-1] == 0) and (d['type'][i] not in
↪['DCC','EXT_DCC'] ) ):
    #                 '''No Action to be taken'''
    #                 d[position][i] = d[position][i-1]
    #                 d[asset_cap][i] = d[position][i] * d['price'][i]
    #                 d[bank_cap][i] = d[bank_cap][i-1]
    #             else:
    #                 '''I take the opposite position here'''
    # #                 d['debug'][i] = 'Hello'
    #                 d[position][i] = 0
    #                 d[asset_cap][i] = 0
    #                 d[bank_cap][i] = abs(d[position][i-1]) * d['price'][i]
    #     else:
    #         d[position][i] = d[position][i-1]
    #         d[asset_cap][i] = d[asset_cap][i-1]
    #         d[bank_cap][i] = d[bank_cap][i-1]



    #     d[total_cap][i] = d[bank_cap][i] + d[asset_cap][i]
    #     d[daily_ret][i] = (d[total_cap][i] - d[total_cap][i-1])/
↪d[total_cap][i-1]
```

```
def get_sharpe(data, column):
    """
    params -> data: an ouput from one of the strategies
                column: Sharpe of which column
    returns -> sharpe for that column from the data
    """

    df = data.copy()
    df = df[ df['regime'] >= 0 ]

    '''We use half days thats why to annualize sharpe we get this'''
    return np.sqrt(2*252) * (df[column].mean()/(df[column].std( ddof = 1 ) ) )

def get_profit( data, column):
    """
    params -> data: an ouput from one of the strategies
                column: profit of which column
    returns -> Profit from this column in percent
    """

    return ( data[column].iloc[-1] - data[column].iloc[0] ) / data[column].
 ↪iloc[0]

def get_drawdown( data, column ):
    """
    params -> data: an ouput from one of the strategies
                column: drawdown of which column
    TODO: Need to check this function, might be incorrect
    returns -> Max negative sum of return ( ie, minimum sum of continuous␣
 ↪return (daily_ret) )
    """



    arr = np.array( data[column] )
    curr_min = arr.copy()
    curr = curr_min[0]

    for i in range(1, len(arr) ):
        curr_min[i] = np.minimum( arr[i], curr_min[i-1] + arr[i] )
        curr = min( curr, curr_min[i] )

    '''Very good strategy, never loses money'''
    if( curr > 0 ):
        return -10000
```

```
    '''We would like to minimize drawdown'''
    return -1 * curr

def get_metrics_trading_strategy( data, strategies ):
    """
    params -> data: an ouput from one of the strategies
              strategies: a list of strategy names

    returns -> Dict of dict of metric
    """

    ans = {}

    for strategy in strategies:
        ans[strategy] = {}
        ans[strategy]['drawdown'] = get_drawdown( data, 'daily_ret_'+strategy)
        ans[strategy]['profit'] = get_profit( data, 'total_cap_'+strategy)
        ans[strategy]['sharpe'] = get_sharpe( data, 'daily_ret_'+strategy)

    return ans


def get_loss_function_for_pipeline( data, DC, regimes, theta, init_cap = 1,␣
 ↪strat = 'JC1', threshold = 1):
    """
    Function to get the loss
    Params-> data: Time series( pd.Series ) for the full thing
             DC: List of tuples for DC indicators ( output for get_DC_data_v2 )
             regimes: Filtered regimes, output from hmm model (for train, if␣
 ↪used) /Naive Bayes Classifier(valid)
             theta: Theta value for TMV Calculation
             init_cap: Initial Capital
             strat: Name of strat, "control" for control strategy and any other␣
 ↪string for regime dependent strategy
             threshold: threshold for Trading on TMV

    Returns: A dict of dict with keys (strat, {drawdown, profit, sharpe})
    """
    df = gd.generate_dataset_with_columns( data, DC, regimes, theta )
    if( strat == "control" ):
        df1 = strategy_control(df, init_cap=init_cap, strat=strat, threshold =␣
 ↪threshold)
    elif( strat == "control2"):
        df1 = strategy_control2(df, init_cap=init_cap, strat=strat, threshold =␣
 ↪threshold)
    else:
```

```
        df1 = strategy_regime_dependent(df, init_cap=init_cap, strat=strat,␣
↪threshold = threshold)

    return get_metrics_trading_strategy( df1, [strat]  )
```

visualization.py

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from wordcloud import WordCloud


def wordcloud_clusters(model, vectors, features, n_top_words=50):
    num_clouds = len(np.unique(model.labels_))
    fig, axs = plt.subplots(num_clouds, 1, figsize=(6.4, 6.4 * num_clouds))
    axs = axs.flatten()

    for i, cluster in enumerate(np.unique(model.labels_)):
        ax = axs[i]
        size = {}
        words = vectors[model.labels_ == cluster].sum(axis=0)
        largest = words.argsort()[::-1]
        for j in range(0, n_top_words):
            size[features[largest[j]]] = abs(words[largest[j]])
        wc = WordCloud(
            background_color="white",
            max_words=100,
            width=500,
            height=300
        )
        wc.generate_from_frequencies(size)
        ax.imshow(wc, interpolation="bilinear")
        ax.axis("off")
        ax.set_title(f'Cluster = {cluster}')

    fig.tight_layout()
```

Install External Modules

[1]:
```
# !pip install finbert-embedding
# !pip install gensim
# !pip install hmmlearn
# !pip install numpy==1.21.4
# !pip install pandas==1.5.1
# !pip install pyldavis
# !pip install seaborn
```

```
# !pip install torch torchvision torchaudio
# !pip install transformers
# !pip install yahoofinance
```

Imports/Settings

Macro Variables

[2]:
```
# Set LDA_IMPORT flag to True only if you have the correct version of Pandas␣
↪installed!
LDA_IMPORT = False
```

Import External Modules

[3]:
```
import warnings
warnings.filterwarnings('ignore')
```

[4]:
```
from datetime import datetime
from finbert_embedding.embedding import FinbertEmbedding
import matplotlib.cm as cm
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter
import nltk
nltk.download("stopwords")
from nltk.corpus import stopwords
import numpy as np
import os
import pandas as pd
if LDA_IMPORT:
    import pyLDAvis
import seaborn as sns
from sklearn import preprocessing
from sklearn.cluster import KMeans
from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS
import sys
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\dhruv\AppData\Roaming\nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

Import Internal Modules

[5]:
```
sys.path.append(os.getcwd() + '/modules')
from modules.cross_validation import Pipeline, CustomCrossValidation
from modules.data import *
from modules.directional_change import *
```

```
from modules.hidden_markov_model import make_regime_plots, fit_hmm
from modules.kmeans import *
from modules.logistic_regression import do_all_LR
from modules.NaiveBayesClassifier import *
from modules.svm import do_all_SVM
from modules.text_preprocessing import *
from modules.topic_modeling import *
from modules.trading_strategy import *
from modules.visualization import *
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\dhruv\AppData\Roaming\nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

[6]:
```
plt.style.use('seaborn')
sns.set_theme()
```

Assign Train/Test Dates

[7]:
```
period_start = datetime(1985, 1, 1)
period_end = datetime(2023, 6, 30)

train_start = datetime(1985, 1, 1)
train_end = datetime(2019, 12, 31)
test_start = datetime(2020, 1, 1)
test_end = datetime(2023, 6, 30)
```

Unsupervised Learning

Natural Language Processing

Text Data - Reading

[8]:
```
FOMC_FPATH = '../fomc_documents/fomc_documents.csv'
# FOMC_PATH = 'data/fomc_documents.csv'
```

[ ]:
```
fomc_data = get_text_data(fpath=FOMC_FPATH)
fomc_data
```

Text Data - Pre-Processing

[ ]:
```
# Remove names
fomc_data.text = fomc_data.text.apply(remove_names_from_minutes)

# Remove stop-words
fomc_data.text = fomc_data.text.apply(tokenizer_wo_stopwords)
```

```
# Set index as meeting_date
fomc_data.set_index('meeting_date', inplace=True)

fomc_data
```

[11]:
```
# Define train and test data
train_data = fomc_data[(fomc_data.index >= train_start) & (fomc_data.index <=␣
 ↪train_end)]
test_data = fomc_data[(fomc_data.index >= test_start) & (fomc_data.index <=␣
 ↪test_end)]
```

Label Generation

TF-IDF Values Computation

[12]:
```
# Compute TF-IDF values
tfidf_class = TF_IDF(X_train=train_data.text, X_test=test_data.text)

tfidf_class.fit_manual()
tfidf_class.fit_gensim()
```

K-Means Clustering on TF-IDF Values

[13]:
```
# Train KMeans Clustering
model_kmeans = KMeansCluster(
    k=2,
    X_train=preprocessing.normalize(tfidf_class.tfidf_gensim_train),
    X_test=preprocessing.normalize(tfidf_class.tfidf_gensim_test),
)
model_kmeans.fit()
model_kmeans.predict()
```

[14]:
```
display(pd.merge(
    left=model_kmeans.sizes_train_df,
    right=model_kmeans.sizes_test_df,
    left_index=True,
    right_index=True,
    suffixes=('_TRAIN', '_TEST'),
)['CLUSTER_SIZE_TRAIN'])
```

```
CLUSTER
0    177
1    104
Name: CLUSTER_SIZE_TRAIN, dtype: int32
```

[ ]:
```
assert model_kmeans.labels_.shape[0] == train_data.shape[0]
assert model_kmeans.y_test_pred.shape[0] == test_data.shape[0]

nlp_regimes_train = pd.DataFrame.from_dict({
    'NLP_Regimes': model_kmeans.labels_
```

```
}).set_index(train_data.index)
nlp_regimes_test = pd.DataFrame.from_dict({
    'NLP_Regimes': model_kmeans.y_test_pred
}).set_index(pd.to_datetime(test_data.index))

display(nlp_regimes_train)
# display(nlp_regimes_test)
```

Wordclouds using Training Labels

[16]:
```
wordcloud_clusters(
    model_kmeans.model,
    preprocessing.normalize(tfidf_class.tfidf_gensim_train),
    tfidf_class.dict_gensim_statements,
)
```


Cluster = 0
currencies
transactions
financial
recent
participants
exchange
obligations
securities
manager
previous
funds
economic
account
november
september
march
october
reported
period
foreign
markets
developments
august
required
inflation
guaranteed
approved
range
soma
held
june
july
agencies
prices
operations
government
policy
rate
percent
fomc
treasury
domestic
vote
issued
fully
february
conditions
agency
growth


Cluster = 1
official
february
serve
august
davis
discontinuance
division
bank
selected
actions
accepted
cease
alan
examination
october
event
selection
following
vice
connection
economist
scheduled
vote
elected
assistant
chairman
july
taken
held
november
counsel
thomas
december
michael
deputy
economists
september
successors
officers
understanding
june
general
january
regularly
approved
march
david
election
secretary
governors

Feature Generation

Topic Modeling (CV Scoring)

[17]:
```
X_test = tfidf_class.X_test.apply(tokenizer_wo_stopwords).apply(lambda x: x.
 ↪split(" "))
bow_test = [tfidf_class.dict_gensim_statements.doc2bow(text) for text in X_test]

topicmod = TopicModel(
    tfidf_class.tfidf_statements_train,
    tfidf_class.dict_gensim_statements,
    tfidf_class.X_train.apply(tokenizer_wo_stopwords).apply(lambda x: x.split("␣
 ↪")).tolist(),
    bow_test,
)

topicmod.fit_predict()
```

[18]:
```
topicmod.num_topic
```

[18]:
```
10
```

[19]:
```
topicmod.cv_topics_list
```

[19]:
```
[(10, 0.4962291321541573),
 (5, 0.48200517385359387),
 (7, 0.45150784480669054),
 (2, 0.3560655730463756)]
```

[20]:
```
# pdf_test = topicmod.pdf_test
print(topicmod.pdf_test.shape)
```

```
(28, 10)
```

[21]:
```
topic_models_train = pd.DataFrame(
    topicmod.doc_mat,
    columns=[f"Topic_{i}" for i in range(10)],
    index=tfidf_class.X_train.index
)

topic_models_test = pd.DataFrame(
    topicmod.pdf_test,
    columns=[f"Topic_{i}" for i in range(10)],
    index=tfidf_class.X_test.index
)
```

Top 10 Words by Topic

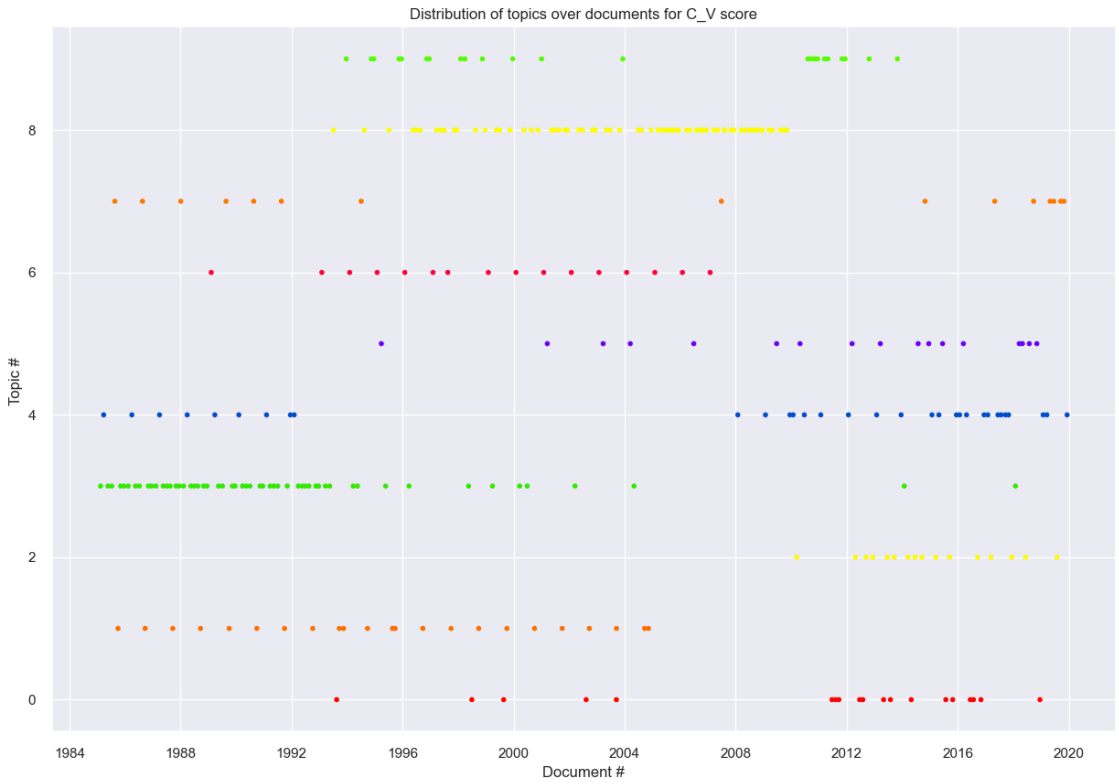
[22]:
```
print("Top 10 words for topics")
topicmod.cv_model.show_topics(num_words=10)
```

```
Top 10 words for topics
```

[22]:
```
[(0,
  '0.031*"june" + 0.011*"inflation" + 0.009*"economic" + 0.009*"labor" +
0.008*"policy" + 0.007*"pace" + 0.007*"conditions" + 0.006*"participants" +
0.006*"growth" + 0.006*"quarter"'),
 (1,
  '0.067*"august" + 0.036*"september" + 0.018*"april" + 0.013*"s" +
0.009*"inflation" + 0.009*"participants" + 0.008*"business" + 0.008*"selection"
+ 0.008*"agency" + 0.007*"obligations"'),
 (2,
  '0.027*"participants" + 0.021*"projections" + 0.018*"financial" + 0.017*"rate"
+ 0.017*"percent" + 0.016*"inflation" + 0.014*"domestic" + 0.011*"october" +
0.010*"appropriate" + 0.009*"unemployment"'),
 (3,
  '0.089*"taken" + 0.081*"actions" + 0.064*"march" + 0.057*"february" +
0.056*"approved" + 0.052*"held" + 0.033*"vote" + 0.021*"chairman" +
0.018*"required" + 0.016*"august"'),
 (4,
  '0.014*"deputy" + 0.011*"secretary" + 0.011*"assistant" + 0.011*"counsel" +
0.010*"economist" + 0.010*"rate" + 0.009*"general" + 0.009*"rates" +
0.008*"continued" + 0.008*"policy"'),
 (5,
  '0.018*"soma" + 0.013*"inflation" + 0.013*"january" + 0.011*"participants" +
0.011*"rate" + 0.010*"economic" + 0.008*"policy" + 0.007*"financial" +
0.007*"growth" + 0.007*"guaranteed"'),
 (6,
  '0.039*"connection" + 0.037*"official" + 0.020*"discontinuance" +
0.019*"officers" + 0.019*"cease" + 0.019*"successors" + 0.019*"event" +
0.018*"elected" + 0.017*"election" + 0.016*"governors"'),
 (7,
  '0.022*"taken" + 0.020*"actions" + 0.020*"july" + 0.016*"november" +
0.014*"rate" + 0.009*"participants" + 0.009*"economic" + 0.009*"funds" +
0.008*"policy" + 0.007*"range"'),
 (8,
  '0.074*"foreign" + 0.060*"account" + 0.055*"previous" + 0.052*"exchange" +
0.052*"currencies" + 0.050*"recent" + 0.048*"period" + 0.046*"reported" +
0.044*"operations" + 0.044*"markets"'),
 (9,
  '0.050*"november" + 0.033*"september" + 0.019*"securities" +
0.017*"transactions" + 0.017*"treasury" + 0.016*"july" + 0.015*"desk" +
0.013*"met" + 0.013*"fomc" + 0.013*"required"')]
```

Topic Distribution by Time

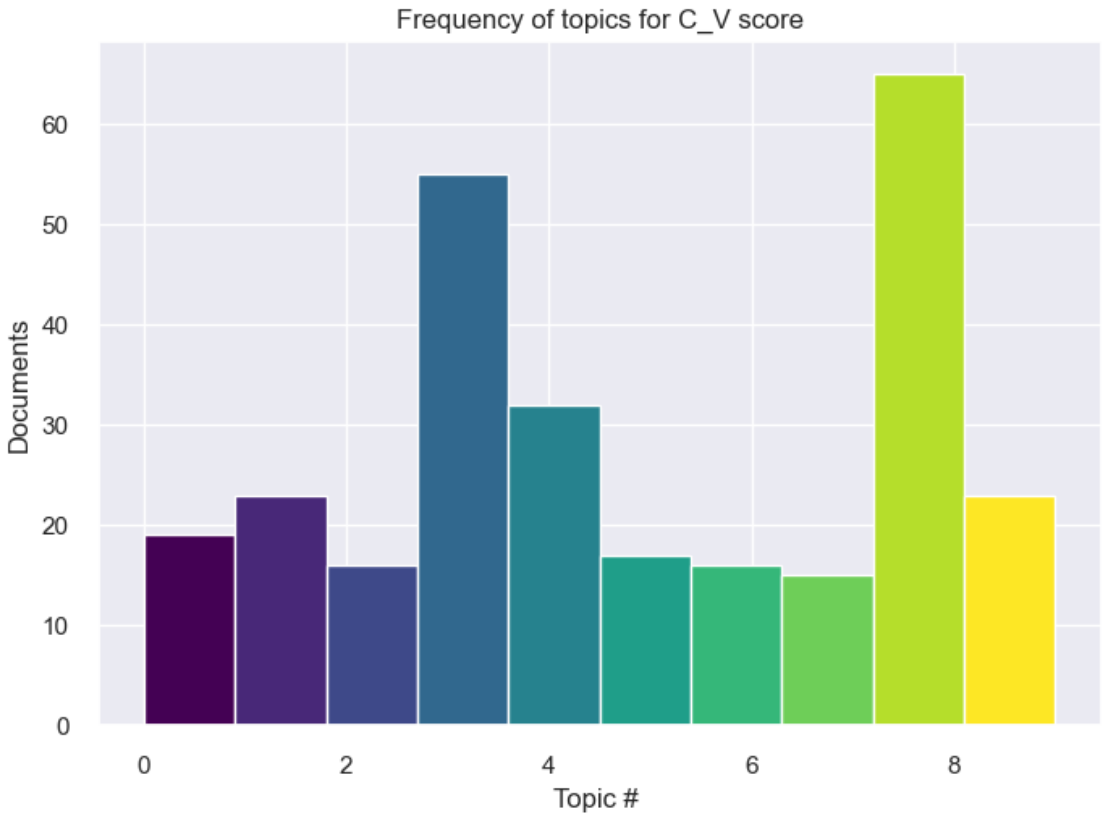
[23]:
```
fig, ax =plt.subplots(1, figsize=(15, 10))
ax.scatter(y=topicmod.topic_mat, x=train_data.index, marker=".", c=topicmod.
 ↪topic_mat, cmap="prism")
ax.set_xlabel("Document #")
ax.set_ylabel("Topic # ")
plt.title("Distribution of topics over documents for C_V score")
plt.show()
```



Topic Frequency across Documents

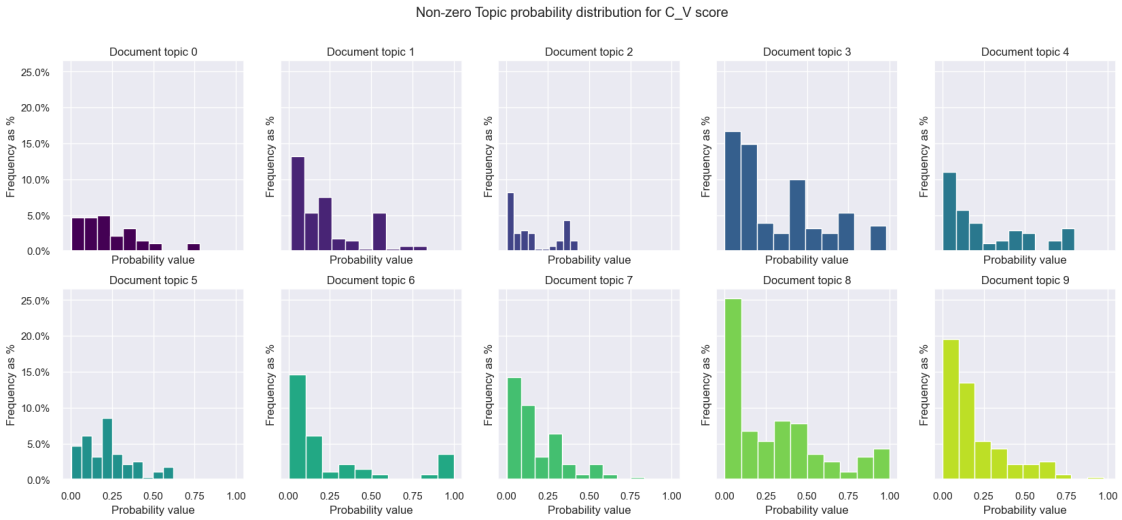
[24]:
```
fig, ax = plt.subplots()
counts, bins, patches = ax.hist(topicmod.topic_mat, bins=10)

# Use a colormap
cmap = plt.get_cmap('viridis')
colors = cmap(np.linspace(0, 1, len(patches)))

for i, patch in enumerate(patches):
    patch.set_facecolor(colors[i])
ax.set_ylabel("Documents")
ax.set_xlabel("Topic # ")
plt.title("Frequency of topics for C_V score")
```

```
plt.show()
```



Probability Distributions by Topic

[25]:
```
fig, ax = plt.subplots(2, 5, figsize=(20, 8),sharey=True, sharex=True)
k = 0
cmap = cm.get_cmap('viridis')
for i in range(2):
    for j in range(5):
        df = topicmod.doc_mat[:, k][topicmod.doc_mat[:, k].nonzero()]
        ax[i, j].hist(df, weights= np.ones_like(df)/len(topicmod.doc_mat[:,k]),␣
 ↪color=cmap(k/10))
        ax[i, j].set_title("Document topic " + str(k))
        ax[i, j].set_xlabel("Probability value")
        ax[i, j].set_ylabel("Frequency as %")
        k=k+1
        ax[i, j].yaxis.set_major_formatter(PercentFormatter(1))
plt.suptitle("Non-zero Topic probability distribution for C_V score")
plt.show()
```



LDA Visualization

[26]:
```
if LDA_IMPORT:
    topic_term_dists = topicmod.cv_model.get_topics()  # transpose to make␣
 ↪shape (num_terms, num_topics)
    doc_topic_dists = topicmod.doc_mat # cv_model.get_document_topics(topicmod.
 ↪tfidf_mat, minimum_probability=0)
    # doc_topic_dists = [[tup[1] for tup in lst] for lst in doc_topic_dists]  #␣
 ↪convert list of tuples to just list
    doc_lengths = [len(doc) for doc in gensim_statements]
    vocab = list(dict_gensim_statements.token2id.keys())
    term_frequency = dict_gensim_statements.cfs

    # Use pyLDAvis
    vis_data = pyLDAvis.prepare(
        topic_term_dists=topic_term_dists,
        doc_topic_dists=doc_topic_dists,
        doc_lengths=doc_lengths,
        vocab=vocab,
        term_frequency=list(term_frequency.values())
    )

    print("Intertopic distance map for C_V Score\n\n")
    pyLDAvis.display(vis_data)
else:
    print(f"Please see attached PDF for LDA Visualization!")
```

```
Please see attached PDF for LDA Visualization!
```

Topic Modeling (UMass Scoring)

[27]:
```
um_topicmod = TopicModel(
    tfidf_class.tfidf_statements_train,
    tfidf_class.dict_gensim_statements,
    tfidf_class.X_train.apply(tokenizer_wo_stopwords).apply(lambda x: x.split("␣
 ↪")).tolist(),
    cv_score="u_mass",
)

um_topicmod.fit()
```

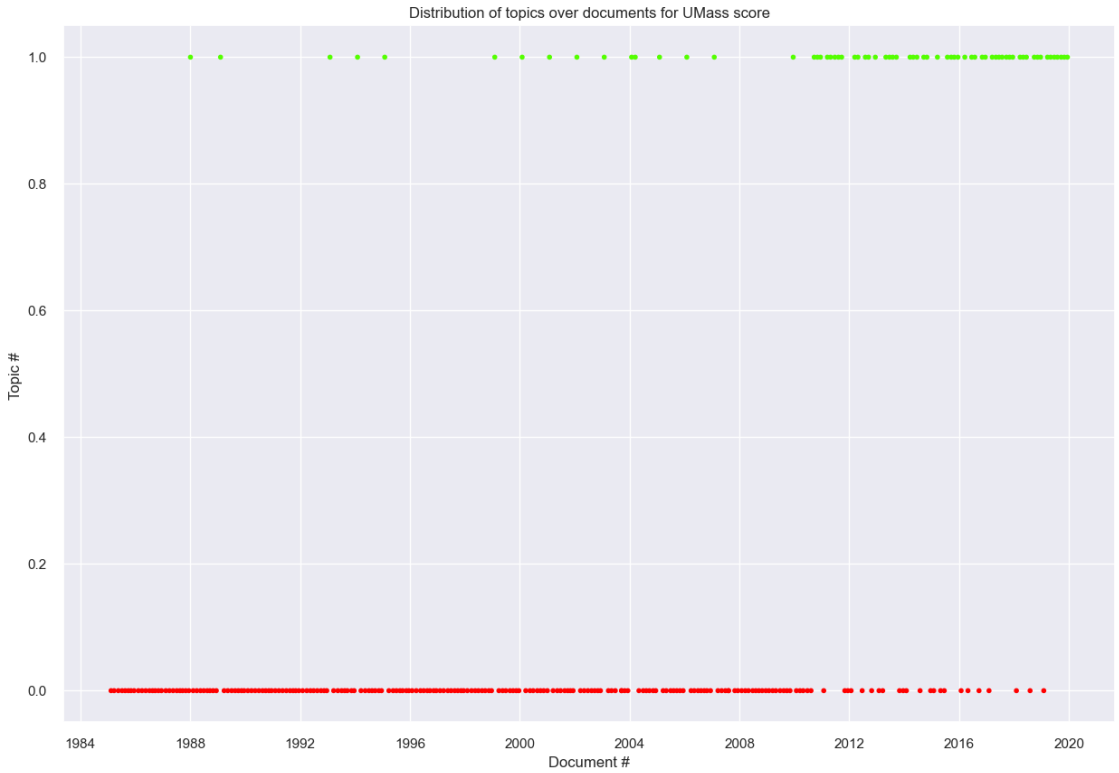
Top 10 Words by Topic

[28]:
```
um_topicmod.cv_model.show_topics(num_words=10)
```

[28]:
```
[(0,
  '0.017*"foreign" + 0.014*"recent" + 0.014*"account" + 0.012*"taken" +
0.012*"reported" + 0.012*"markets" + 0.011*"actions" + 0.011*"developments" +
0.011*"period" + 0.010*"approved"'),
 (1,
  '0.012*"participants" + 0.012*"inflation" + 0.011*"rate" + 0.009*"s" +
0.009*"economic" + 0.008*"policy" + 0.007*"securities" + 0.007*"percent" +
0.005*"funds" + 0.005*"growth"')]
```

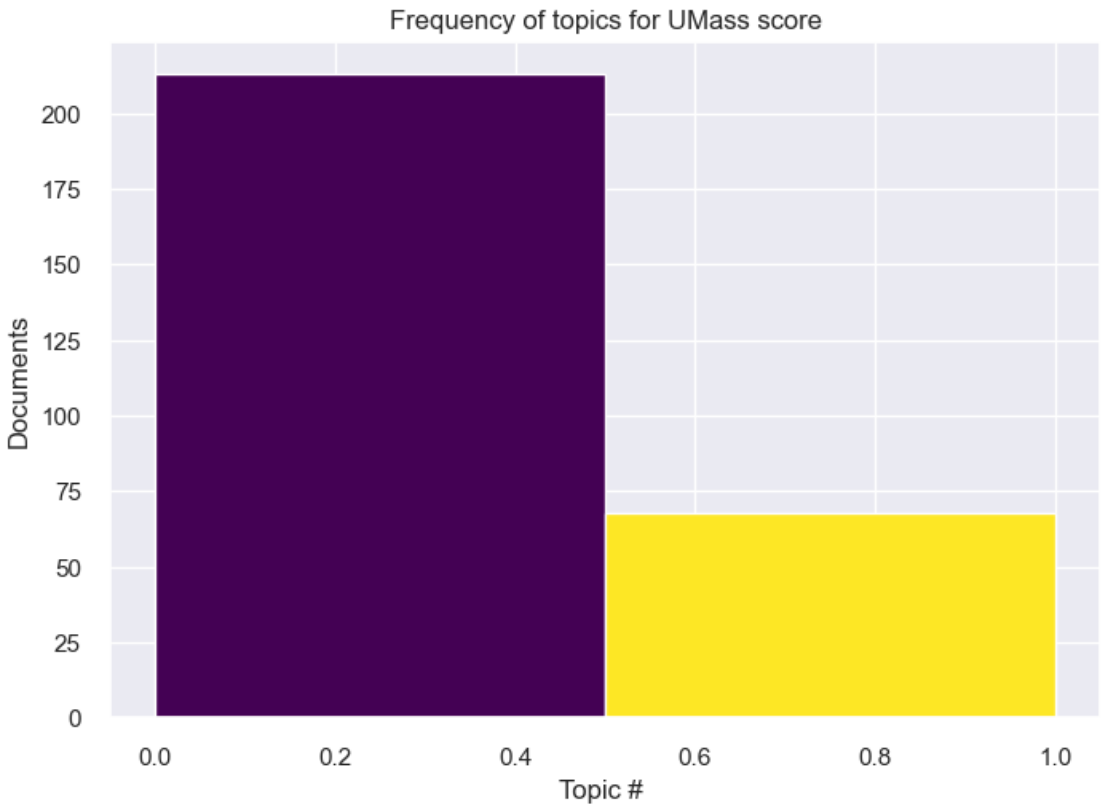
Topic Distribution by Time

[29]:
```
fig, ax =plt.subplots(1, figsize=(15, 10))
ax.scatter(y=um_topicmod.topic_mat, x=train_data.index, marker="."
 ↪",c=um_topicmod.topic_mat, cmap="prism")
ax.set_xlabel("Document #")
ax.set_ylabel("Topic # ")
plt.title("Distribution of topics over documents for UMass score")
plt.show()
```



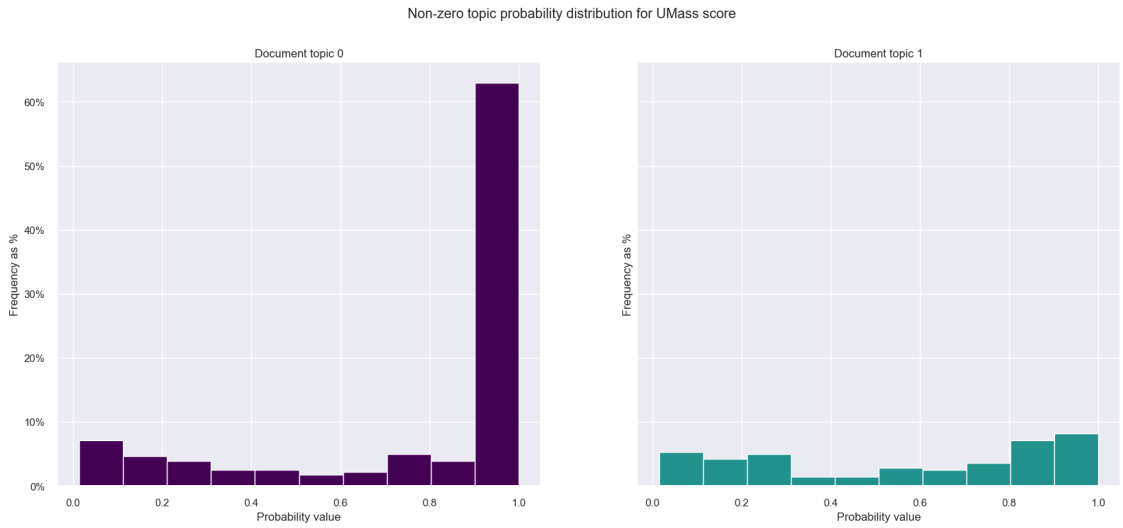
Topic Frequency across Documents

[30]:
```
fig, ax = plt.subplots()
counts, bins, patches = ax.hist(um_topicmod.topic_mat, bins=2)

# Use a colormap
cmap = plt.get_cmap('viridis')
colors = cmap(np.linspace(0, 1, len(patches)))

for i, patch in enumerate(patches):
    patch.set_facecolor(colors[i])
ax.set_ylabel("Documents")
ax.set_xlabel("Topic # ")
plt.title("Frequency of topics for UMass score")
plt.show()
```



Probability Distributions by Topic

[31]:
```
fig, ax = plt.subplots(1, 2, figsize=(20, 8), sharey=True, sharex=True)
cmap = cm.get_cmap('viridis')
k = 0
for j in range(2):
        df = um_topicmod.doc_mat[:, k][um_topicmod.doc_mat[:, k].nonzero()]
        ax[j].hist(df, weights= np.ones_like(df)/len(um_topicmod.doc_mat[:,k]),␣
 ↪color=cmap(k/2))
        ax[j].set_title("Document topic " + str(k))
        ax[j].set_xlabel("Probability value")
        k=k+1
        ax[j].yaxis.set_major_formatter(PercentFormatter(1))
        ax[j].set_ylabel("Frequency as %")
plt.suptitle("Non-zero topic probability distribution for UMass score")
plt.show()
```



LDA Visualization

[32]:
```
if LDA_IMPORT:
    topic_term_dists = um_topicmod.cv_model.get_topics() # transpose to make␣
 ↪shape (num_terms, num_topics)
    doc_topic_dists = um_topicmod.doc_mat # cv_model.
 ↪get_document_topics(topicmod.tfidf_mat, minimum_probability=0)
    # doc_topic_dists = [[tup[1] for tup in lst] for lst in doc_topic_dists]  #␣
 ↪convert list of tuples to just list
    doc_lengths = [len(doc) for doc in gensim_statements]
    vocab = list(dict_gensim_statements.token2id.keys())
    term_frequency = dict_gensim_statements.cfs

    # Use pyLDAvis
    vis_data = pyLDAvis.prepare(
        topic_term_dists=topic_term_dists,
        doc_topic_dists=doc_topic_dists,
        doc_lengths=doc_lengths,
        vocab=vocab,
        term_frequency=list(term_frequency.values())
    )
    print("Intertopic distance map for UMass score\n\n")
    pyLDAvis.display(vis_data)
else:
    print(f"Please see attached PDF for LDA Visualization!")
```

```
Please see attached PDF for LDA Visualization!
```

FinBERT Word Embeddings

[33]:
```
finbert = FinbertEmbedding()
```

[34]:
```
def word_embedding_to_np(text: str):
    sentence_embedding = finbert.sentence_vector(text)
    res = np.array(list(map(lambda x: x.detach().numpy(), sentence_embedding)))
    return res
```

[35]:
```
finbert_embeddings_train = train_data.text.apply(word_embedding_to_np)
finbert_embeddings_test = test_data.text.apply(word_embedding_to_np)
```

[36]:
```
finbert_embeddings_train = pd.DataFrame(
    np.array(list(map(lambda x: list(x), finbert_embeddings_train.values))),
    columns=[f"Word {i}" for i in range(len(finbert_embeddings_train.
 ↪values[0]))],
    index=train_data.index,
)

finbert_embeddings_test = pd.DataFrame(
    np.array(list(map(lambda x: list(x), finbert_embeddings_test.values))),
    columns=[f"Word {i}" for i in range(len(finbert_embeddings_test.
 ↪values[0]))],
    index=test_data.index,
)
```

Hidden Markov Models

Price Data - Reading

[37]:
```
epsilon = 0.5
theta = 0.01
trading_day = {'equity':12, 'fx':12,'bond':12}
```

[38]:
```
df_ts = get_ts_data(
    '^GSPC',
    start_date=period_start,
    end_date=period_end,
    delta=trading_day['equity']
)
```

```
[**********100%%**********]  1 of 1 completed
```

Price Data - Pre-Processing

[ ]:
```
df_ts_train = df_ts[(df_ts.index.date >= train_start.date()) & (df_ts.index.
 ↪date <= train_end.date())]
df_ts_test = df_ts[(df_ts.index.date >= test_start.date()) & (df_ts.index.date␣
 ↪<= test_end.date())]

df_ts_train
```

[40]:
```
def get_r_values(data):
    r_values = get_R(
        get_TMV(get_DC_data_v2(data, theta), theta),
        get_T(get_DC_data_v2(data, theta)),
        theta
    )
    return r_values

r_values_train = get_r_values(df_ts_train)
r_values_test = get_r_values(df_ts_test)
```

Label Generation

[41]:
```
hmm_regimes_train, hmm_model = fit_hmm(
    2,
    df_ts_train,
    r_values_train,
    '^GSPC',
    plot=False,
    verbose=False
)

hmm_regimes_test = hmm_model.predict(r_values_test.values.reshape(-1, 1))
```

[42]:
```
hmm_regimes_train = pd.DataFrame.from_dict({
    'HMM_Regimes': list(hmm_regimes_train.values),
}).set_index(r_values_train.index)

hmm_regimes_test = pd.DataFrame.from_dict({
    'HMM_Regimes': hmm_regimes_test,
}).set_index(r_values_test.index)
```

[ ]:
```
display(hmm_regimes_train)
```

Supervised Learning

Fill Labels Across Entire Time Period

[ ]:
```
train_regimes = pd.DataFrame(index=pd.date_range(
    start=period_start,
    end=train_end + timedelta(1),
    freq='12H'
)[:-1])

train_regimes = pd.merge(
    left=train_regimes,
    right=nlp_regimes_train,
```

```
    how="left",
    left_index=True,
    right_index=True,
).bfill()

train_regimes = pd.merge(
    left=train_regimes,
    right=hmm_regimes_train,
    how="left",
    left_index=True,
    right_index=True,
).bfill()

train_regimes.NLP_Regimes = train_regimes.NLP_Regimes.ffill()
train_regimes.HMM_Regimes = train_regimes.HMM_Regimes.ffill()

train_regimes
```

Equalize Indexes for Features

[45]:
```
index_train = r_values_train.index
X_train = pd.DataFrame(index=index_train)

# ffill() topic model PDFs to account for dates on which we have text data but␣
 ↪no DC data
topic_models_train_new = pd.merge(
    left=X_train,
    right=topic_models_train,
    how='outer',
    left_index=True,
    right_index=True
).ffill()
topic_models_train_new = topic_models_train_new[topic_models_train_new.index.
 ↪isin(index_train)]

# ffill() word embeddings to account for dates on which we have text data but␣
 ↪no DC data
finbert_embeddings_train_new = pd.merge(
    left=X_train,
    right=finbert_embeddings_train,
    how='outer',
    left_index=True,
    right_index=True
).ffill()
finbert_embeddings_train_new =␣
 ↪finbert_embeddings_train_new[finbert_embeddings_train_new.index.
 ↪isin(index_train)]
```

```
# Add name to R Values Series
r_values_train.name = 'R_Values_Train'
```

[46]:
```
index_test = r_values_test.index
X_test = pd.DataFrame(index=index_test)

# ffill() topic model PDFs to account for dates on which we have text data but␣
 ↪no DC data
topic_models_test_new = pd.merge(
    left=X_test,
    right=topic_models_test,
    how='outer',
    left_index=True,
    right_index=True
).ffill()
topic_models_test_new = topic_models_test_new[topic_models_test_new.index.
 ↪isin(index_test)]

# ffill() word embeddings to account for dates on which we have text data but␣
 ↪no DC data
finbert_embeddings_test_new = pd.merge(
    left=X_test,
    right=finbert_embeddings_test,
    how='outer',
    left_index=True,
    right_index=True
).ffill()
finbert_embeddings_test_new =␣
 ↪finbert_embeddings_test_new[finbert_embeddings_test_new.index.
 ↪isin(index_test)]

# Add name to R Values Series
r_values_test.name = 'R_Values_Test'
```

Construct Covariates and Labels

[47]:
```
# Flags to change covariates used in NB Classifier
USE_TOPIC_MODEL_PDF = True
USE_WORD_EMBEDDINGS = True
USE_R_VALUES = True
```

[48]:
```
def make_X(
    train: bool = True,
    topic_model: bool = USE_TOPIC_MODEL_PDF,
    word_embeddings: bool = USE_WORD_EMBEDDINGS,
    price_data: bool = USE_R_VALUES,
```

```
):
    """
    This function creates a new DataFrame of covariates based on the flags which
    determine specifically which covariates will be included.

    @param train: flag for deciding if making train or test data
    @param topic_model: flag for including NMF Topic Models (loadings)
    @param word_embeddings: flag for including Finbert Word Embeddings
    @param topic_model: flag for including R Indicator Values (price data)
    @return X: pd.DataFrame indexed by DC Indicators containing specified␣
 ↪covariates
    """
    if train:
        index = r_values_train.index
        topic_models_df = topic_models_train_new.copy()
        word_embeddings_df = finbert_embeddings_train_new.copy()
        r_values_df = r_values_train.copy()
    else:
        index = r_values_test.index
        topic_models_df = topic_models_test_new.copy()
        word_embeddings_df = finbert_embeddings_test_new.copy()
        r_values_df = r_values_test.copy()

    X = pd.DataFrame(index=index)

    # Add topic model PDFs to covariates DataFrame
    if topic_model:
        X = pd.merge(
            left=X,
            right=topic_models_df,
            how='inner',
            left_index=True,
            right_index=True
        )

    # Add word embeddings to covariates DataFrame
    if word_embeddings:
        X = pd.merge(
            left=X,
            right=word_embeddings_df,
            how='inner',
            left_index=True,
            right_index=True
        )

    # Add DC Indicator (price data) to covariates DataFrame
    if USE_R_VALUES:
```

```
        X = pd.merge(
            left=X,
            right=r_values_df,
            how='inner',
            left_index=True,
            right_index=True
        )

    # TODO: either bfill() here, or remove the NA rows from y_train and X_train
    X = X.bfill()

    if train:
        print(f"X_train: {X.shape}")
    else:
        print(f"X_test: {X.shape}")

    return X
```

[49]:
```
def make_Y(
    train: bool = True,
):
    """
    This function constructs the NLP and HMM y-labels.

    @param train: flag for deciding if making train or test data
    @return (y_nlp, y_hmm): tuple containing NLP and HMM y-labels
    """
    if train:
        regimes = train_regimes.copy()
        index = index_train.copy()
    else:
        regimes = test_regimes.copy()
        index = index_test.copy()

    regimes = regimes[regimes.index.isin(index)]

    y_nlp = regimes.NLP_Regimes
    y_hmm = regimes.HMM_Regimes

    if train:
        print(f"y_train_nlp: {y_nlp.shape}")
        print(f"y_train_hmm: {y_hmm.shape}")
    else:
        print(f"y_test_nlp: {y_nlp.shape}")
        print(f"y_test_hmm: {y_hmm.shape}")

    return y_nlp, y_hmm
```

[50]:
```
def make_data(
    topic_model: bool = USE_TOPIC_MODEL_PDF,
    word_embeddings: bool = USE_WORD_EMBEDDINGS,
    price_data: bool = USE_R_VALUES,
):
    """
    This function creates all X and y data for the classification model.

    @param topic_model: flag for including NMF Topic Models (loadings)
    @param word_embeddings: flag for including Finbert Word Embeddings
    @param topic_model: flag for including R Indicator Values (price data)
    @return (X_train, X_test, y_train_nlp, y_train_hmm, y_test_nlp, y_test_nlp):
 ↪ tuple containing all train and test data
    """
    X_train = make_X(
        train=True,
        topic_model=topic_model,
        word_embeddings=word_embeddings,
        price_data=price_data
    )
    X_test = make_X(
        train=False,
        topic_model=topic_model,
        word_embeddings=word_embeddings,
        price_data=price_data
    )
    y_train_nlp, y_train_hmm = make_Y(train=True)
    # y_test_nlp, y_test_hmm = make_Y(train=False)
    return X_train, X_test, y_train_nlp, y_train_hmm
```

[51]:
```
X_train, X_test, y_train_nlp, y_train_hmm = make_data()
```

```
X_train: (4932, 779)
X_test: (563, 779)
y_train_nlp: (4932,)
y_train_hmm: (4932,)
```

Classification and Performance on Trading Strategies

[67]:
```
def make_test_regimes(X_train, X_test, do_all_func):
    test_regimes = pd.DataFrame(index=pd.date_range(
        start=test_start,
        end=period_end + timedelta(1),
        freq='12H'
    )[:-1])

    def add_to_regime_dataframe(X_train, y_train, X_test, label, test_regimes):
```

```
        y_pred = pd.DataFrame.from_dict({
            label: do_all_func(
                X_train.values,
                y_train.values,
                X_test.values,
            )
        }).set_index(index_test)

        test_regimes = pd.merge(
            left=test_regimes,
            right=y_pred,
            how="left",
            left_index=True,
            right_index=True,
        ).bfill()

        test_regimes[label] = test_regimes[label].ffill()
        return test_regimes

    test_regimes = add_to_regime_dataframe(
        X_train.iloc[:,-1:],
        pd.DataFrame(y_train_nlp),
        X_test.iloc[:,-1:],
        'Kmeans_labels_DC_indicators',
        test_regimes
    )
    test_regimes = add_to_regime_dataframe(
        X_train.iloc[:,:10],
        pd.DataFrame(y_train_nlp),
        X_test.iloc[:,:10],
        'Kmeans_labels_NMF_loadings',
        test_regimes
    )
    test_regimes = add_to_regime_dataframe(
        X_train.iloc[:,10:-1],
        pd.DataFrame(y_train_nlp),
        X_test.iloc[:,10:-1],
        'Kmeans_labels_finBERT_embeddings',
        test_regimes
    )
    test_regimes = add_to_regime_dataframe(
        X_train.iloc[:,-1:],
        pd.DataFrame(y_train_hmm),
        X_test.iloc[:,-1:],
        'HMM_labels_DC_indicators',
        test_regimes
    )
```

```
    test_regimes = add_to_regime_dataframe(
        X_train.iloc[:,:10],
        pd.DataFrame(y_train_hmm),
        X_test.iloc[:,:10],
        'HMM_labels_NMF_loadings',
        test_regimes
    )
    test_regimes = add_to_regime_dataframe(
        X_train.iloc[:,10:-1],
        pd.DataFrame(y_train_hmm),
        X_test.iloc[:,10:-1],
        'HMM_labels_finBERT_embeddings',
        test_regimes
    )

    return test_regimes
```

[68]:
```
result_labels = [
    'Control 1',
    'Control 2',
    'K-means labels, only DC covariates',
    'K-means labels, NMF loading covariates',
    'K-means labels, FinBERT embeddings',
    'HMM labels, only DC covariates',
    'HMM labels, NMF loading covariates',
    'HMM labels, FinBERT embeddings'
]

def make_results(test_regimes):
    results = []

    strat_test = Pipeline(
        df_ts=df_ts,
        to_test=True,
        strat='control',
        start_date=str(train_start)[:10],
        train_end=str(train_end)[:10],
        test_start=str(test_start)[:10],
        theta=theta,
        epsilon=0.5,
        provide_labels=True,
        labels=test_regimes[test_regimes.columns[0]] # labels are a placeholder␣
 ↪for control
    )
    strat_test.fit(verbose=False)
    results.append(strat_test.trading_metrics_test)
```

```
    strat_test = Pipeline(
        df_ts=df_ts,
        to_test=True,
        strat='control2',
        start_date=str(train_start)[:10],
        train_end=str(train_end)[:10],
        test_start=str(test_start)[:10],
        theta=theta,
        epsilon=0.5,
        provide_labels=True,
        labels=test_regimes[test_regimes.columns[0]]
    )
    strat_test.fit(verbose = False)
    results.append(strat_test.trading_metrics_test)

    for label, column in zip(result_labels[2:], test_regimes.columns):
        strat_test = Pipeline(
            df_ts=df_ts,
            to_test=True,
            start_date=str(train_start)[:10],
            train_end=str(train_end)[:10],
            test_start=str(test_start)[:10],
            theta=theta,
            epsilon=0.5,
            provide_labels=True,
            labels=test_regimes[column]
        )
        strat_test.fit(verbose = False)
        results.append(strat_test.trading_metrics_test)

    return results
```

[69]:
```
do_all_map = {
    'nbc': do_all_NBC,
    'svm': do_all_SVM,
    'lr': do_all_LR,
}

def train_predict_eval(classifier):
    assert classifier in list(do_all_map.keys())
    do_all_func = do_all_map[classifier]
    test_regimes = make_test_regimes(X_train, X_test, do_all_func)
    results = make_results(test_regimes)
    results = pd.DataFrame(
        results,
        index=result_labels
    ).round(8).sort_values('sharpe')
```

```
    return test_regimes, results
```

Naive Bayes Classifier

[70]:
```
test_regimes_nbc, results_nbc = train_predict_eval('nbc')
```

[71]:
```
results_nbc
```

[71]:

	drawdown	profit	sharpe
K-means labels, NMF loading covariates	0.154341	0.530849	0.540878
Control 1	0.154341	0.530849	0.541172
K-means labels, only DC covariates	0.154341	0.530849	0.541172
K-means labels, FinBERT embeddings	0.154341	0.530849	0.541172
HMM labels, FinBERT embeddings	0.154341	0.530849	0.541172
Control 2	0.154564	0.529506	0.616659
HMM labels, NMF loading covariates	0.158274	0.530849	0.618412
HMM labels, only DC covariates	0.222633	0.901848	0.710041

Support Vector Machine Classifier

[72]:
```
test_regimes_svm, results_svm = train_predict_eval('svm')
```

[ ]:
```
results_svm
```

Logistic Regression Classifier

[74]:
```
test_regimes_lr, results_lr = train_predict_eval('lr')
```

[ ]:
```
results_lr
```

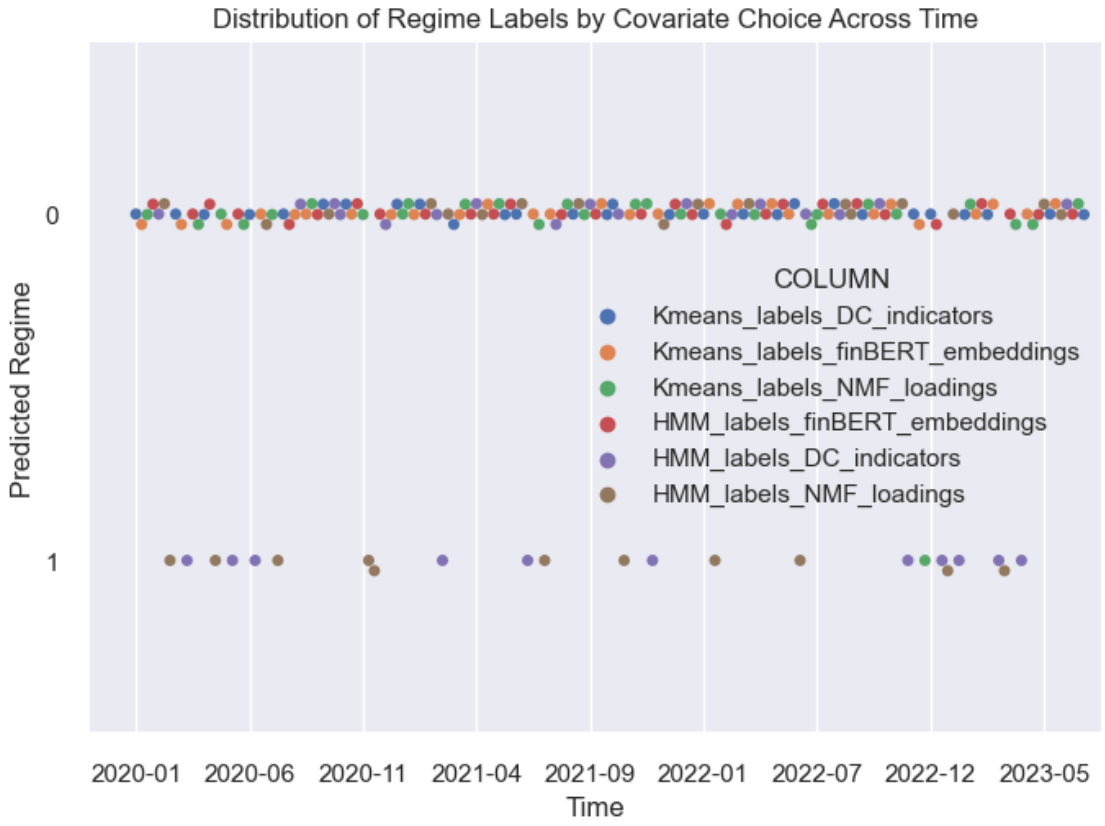
Additional Visualizations

[ ]:
```
test_regimes_nbc_plot = test_regimes_nbc[pd.DatetimeIndex(test_regimes_nbc.
↪index.date).isin(test_data.index)]
```

[ ]:
```
label_list = []
column_list = []
date_list = []
test_regimes = test_regimes_nbc_plot

for i, col in enumerate(test_regimes.columns):
    col_vals = test_regimes.loc[:, test_regimes.columns[i]]
    label_list += list(col_vals.values)
    column_list += [col for _ in range(col_vals.shape[0])]
    date_list += list(test_regimes.index.date)
```

```
df = pd.DataFrame.from_dict({
    'LABEL': label_list,
    'COLUMN': column_list,
    'DATE': date_list,
}).sort_values(by='DATE')

df['LABEL'] = df['LABEL'].astype("int").astype("category")
df.DATE = pd.to_datetime(df.DATE)
df = df.drop_duplicates(subset=('COLUMN', 'DATE'))

df.reset_index(inplace=True)
df['x'] = df.index.astype("int")

df
```

[246]:
```
ax = sns.swarmplot(
    data=df.drop_duplicates(subset=('COLUMN', 'DATE')),
    x="x",
    y="LABEL",
    hue="COLUMN"
)

labels = ax.get_xticklabels()

def get_date(label):
    txt = str(label)
    if '-' in txt or '-' in txt:
        return txt
    txt = int(txt)
    if txt > max(df.x.astype("int")):
        return str(txt)
    my_date = pd.to_datetime(df[df.x == txt].DATE.values[0]).date()
    return f"{my_date.year}-{'0' + str(my_date.month) if my_date.month < 10␣
 ↪else my_date.month}"

ax.set_xticklabels(list(map(get_date, [-20] + [i * 20 for i in range(9)])))

xlim = ax.get_xlim()
ax.set_xlim(xlim[0], xlim[1] - 5)

ax.set_xlabel('Time')
ax.set_ylabel('Predicted Regime')
ax.set_title('Distribution of Regime Labels by Covariate Choice Across Time')

plt.show()
```


Distribution of Regime Labels by Covariate Choice Across Time
Predicted Regime
0
1
COLUMN
Kmeans_labels_DC_indicators
Kmeans_labels_finBERT_embeddings
Kmeans_labels_NMF_loadings
HMM_labels_finBERT_embeddings
HMM_labels_DC_indicators
HMM_labels_NMF_loadings
2020-01
2020-06
2020-11
2021-04
2021-09
2022-01
2022-07
2022-12
2023-05
Time