ExplainerNotebook

May 7, 2021

1 Expose your parliament - Explainer notebook

Group: Sam Rahbar (s183670), Jonas Mærsk (s183635) and Mikkel Goldschmidt (s183966).

All members of the group contributed equally to the project.

1.1 Introduction (and a crash course in Danish politics)

The aim of this report is to get insight into the Danish power structure in the national parliament "Folketinget" (often abreviated FT). The parlament has 179 member of whom 2 are elected in Greenland and 2 on the Faroe Islands (both contries being part of The Kingdom of Denmark). The members of parlament are often revefered to as MF (Medlem af/Member of Folketinget).

To be able to properly understand this project and the analysis, one must understand the basics of Danish politics. As the report goes along, it will try to explain the domain knowledge, but to make the reading process easier, we have decided to write a very small crash course. As stated in the beginning The Kingdom of Denmark has a national parliament. The members are primarily elected through parties for a period of a maximum of 4 years. This report will focus on an election period between 2015 and 2019 in which the Danish prime minister was Lars Løkke Rasmussen the leader of the party Venstre. The reason for limiting ourselves to this period is that the power structure changed a lot after the election in 2019 when the new prime minister became Mette Frederiksen - the leader of Socialdemokraterne. Denmark has a lot of parties (as opposed to contries like the US with effectively only 2 parties). Each party will typically choose a logo and a color, that is distinct and easily recognizeable. To refer to a party in shorthand usually a few letters are used. Unfortuneatly there is no definte standard for what is used. Sometimes a set of letters abreviating the entire party name is used. Sometimes an official letter used for the election lists are used - this is handled a little ad hoc throughout the report, since there really isn't a standard to follow. In this period of time, the three biggest parties in Folketinget were:

- Venstre (meaning Left) - Has a *blue color*, a stylized V as logo and is commonly seen as a center-right party (yes, you read right, the party named Left is a right-leaning party). The usually use the letter 'V' as abreviation.
- Socialdemokraterne (meaning The Social Democrats) - Has a *red color*, a rose as logo and is commonly seen as s center-left party. They are usually abreviated by 'S' or 'A'.
- Dansk Folkeparti (meaning The Danish Peoples Party) - Has a *yellow color*, a logo with the letters DF encapsulated by two Danish flags and is commonly seen as a right-leaning party with focus on nationalistic values. They are usually abreviated as either 'DF' or 'O'.

Most of these parties have a youth organization - usually just named the same as the party suffixed with the word "Ungdom" (meaning youth).

1.2 The FT Odata dataset

To gain insight into the Danish political landscape, we have used the open data provided by Folketinget about the workings of parlament. The data is a relational database that can be found at https://oda.ft.dk/Home/OdataQuery. It exposes information on a lot of different objects, among them:

- Members of parlament
  - Biographies introducing the background of the members
  - Which commities they sit on
  - Which party they belong to
  - Their roles on different commities and governmental institutions (like minesterial titles)
- Meetings
  - Written transcripts from the meetings
  - Lists of who attended the meetings

The data does contain a lot more information than listed above, but these are the important points from the analysis done in this report. Another important thing about the dataset is that it is in Danish. This does give rise to some problems. Among these text analysis where the standard tools are defined in English.

1.3 Personal motivation

The reason that we chose this dataset was simply an interest in politics. Further the idea of finding someone powerfull who wasn't in the public eye intrigued us (though throughout the project we came to realize that such conclusion required more time than we had available). We were also motivated by the fact we hadn't seen anyone analyze this dataset before, making us feel like we had an oppertunity to actually discover something new.

1.4 Goal for the end user

The hope was to give some level of insight into the Danish political landscape that is not portrayed by the usual media. Where the media is very focused on qualitative data like interview and their analysis the motivations of a specific politician in a given situation, we were able to give a more qualitative look at what happens in parliament. It is sometimes seen in the Danish media that someone uses the data provided here for stuff like counting how much a specific politician talks in parliament (usually highlighting politicians that are deemed to not participate enough in the democratic process). We have however not seen a more quantitative analysis of language usage and how the politicians interacted with each other. The goal was therefore to provide the end user with some insight into the political landscape using this more quantitative approach.

1.5 Structure of the report

Even though the problem description wanted for the report to be structured with an initial look and preprocessing of the data first (section 2) and a description of the tools after (3), we decided to change the structure slightly. The text data that we are analyzing is very loosely coupled with the network data - hence the dataset are preprocessed seperately. We therefore decided to do preprocessing and data analysis of the network data first and then do those two again afterwards for the text analysis.

1.6 Network analysis

We have investigated to different ways of making networks from the dataset. Only the latter is presented on the website. The first is from a linking table linking different political actors together (defined in the section below). The latter is from the biographies that the politicians have written about themselves.

1.6.1 The predefined relations (Aktør-Aktør)

The terminology in the dataset defines an `Actor` ("Aktør" in the dataset) which is important to understand if you investigate the dataset. An `Actor` is some kind of political agent which can be anything from a MF to a ministerial title or the head of a commitee. The dataset provides a linking table relating the Actors to other actors in a variety of ways (the `AktørAktør` table). This table will be the starting point to make a network of politicians, as it provides a way to link politicians to one another. The exact details of linking will be explained as the data is loaded and cleaned.

The `.csv` files loaded into the program is simply dumps of the entire tables from the Odata database with corrosponding names. That is the table `Aktør` corrosponds to the file `data/ft/Aktør.csv`.

Data cleaning Since the data is mined from a relational database, we need to do some linking to get the information needed. Our aim is to build a network of people (primarily politicians) and how they relate to oneanother. Hence we will first find all Actors and then filter out those that are actual people.

[1]:
```
from collections import Counter
import networkx as nx
import netwulf as nw
import pandas as pd

# The table containing all actors
actors = pd.read_csv('data/ft/Aktør.csv')
# The table describing the different types of actors that exist in the data
# This could be something equivalent to "Minister" or "Member of parlament"
actor_types = pd.read_csv('data/ft/Aktørtype.csv').set_index('id', drop=False)

# The relation table between actors
# A lot of these are links from people to subcommities like the Comitee on␣
 ↪Health.
actor_relations = pd.read_csv('data/ft/AktørAktør.csv')
# Each of the links are tagged with a type of relation.
```

```
# This enables the distinction between for instance an Actor being the head of␣
 ↪a comitee
# and just being a member of the comiteee.
actor_relation_types = pd.read_csv('data/ft/AktørAktørRolle.csv').
 ↪set_index('id', drop=False)
```

To make it easier to query the data, the relations type-tables are joined onto the main table.

[2]:
```
actors = actors.merge(actor_types[['type']], left_on='typeid', right_on='id')
actor_relations = actor_relations.merge(actor_relation_types[['rolle']],␣
 ↪left_on='rolleid', right_on='id')
actors.sample(3)
```

[2]:
```
          id  typeid gruppenavnkort                              navn  \
1668     286       5            NaN             Ane Halsboe-Jørgensen
14852   9489      12            NaN  Niels Jørn Schak Krog, Hjørring
12710   2650      12            NaN       Benniy Dan Jensen, Rødovre

                     fornavn          efternavn  \
1668                     Ane  Halsboe-Jørgensen
14852  Niels Jørn Schak Krog,           Hjørring
12710      Benniy Dan Jensen,            Rødovre

                                              biografi  periodeid  \
1668   <member><url>/medlemmer/mf/a/ane-halsboe-joerg…        NaN
14852                                              NaN        NaN
12710                                              NaN        NaN

               opdateringsdato            startdato             slutdato  \
1668   2021-04-07T19:20:35.493  2015-10-08T00:00:00  2016-04-27T00:00:00
14852  2014-09-22T14:15:45.577                  NaN                  NaN
12710  2014-09-11T17:35:54.277                  NaN                  NaN

              type
1668        Person
14852  Privatperson
12710  Privatperson
```

[3]:
```
actor_relations.sample(3)
```

[3]:
```
             id  fraaktørid  tilaktørid            startdato slutdato  \
17099     71668       14387        9987  1979-03-14T00:00:00      NaN
93260  24935725         172       16917  2017-07-01T00:00:00      NaN
18892     75133       14085       10410                  NaN      NaN

               opdateringsdato  rolleid   rolle
17099  2015-03-19T11:02:52.277       15  medlem
93260   2021-04-07T19:20:53.95       15  medlem
```

```
18892  2015-04-08T10:36:33.217       15  medlem
```

To get access to the party of each politician, their biography has to be read. Here the sex of each politician can be seen as well. Note that the biography is XML-encoded, which gives rise to the method of extraction below.

[4]:
```
actors = actors.fillna('')
import re

# XML-decoding probably should be done using regex, but it works in this case
def getTag(bio,tag):
    if bio:
        results = re.search('<'+tag+'>(.*)</'+tag+'>', bio)
        if results: return results[1]
    else: return ''

actors['party'] = actors['biografi'].apply(getTag, args=('partyShortname',))
actors['køn'] = actors['biografi'].apply(getTag, args=('sex',))
```

The type of an Actor can be used to determine if this an actual person (which are the only Actors that we are intersted in linking). Two types of persons are defined in the data: "Person" (which translates directly to English) and "Privatperson" (which translates to a private citizen). Note here that by a person is meant a politician, that is registered as such in the dataset. These people are primarily current members of parlament.

[5]:
```
persons = actors[actors['type'].isin(['Person', 'Privatperson'])]
personIds = persons['id'].values
```

The actors in the dataset are all tagged with a `periode` (the precise definition of each is defined in the `Perioder` table of the database). This value specified as period of time that the parlament was in session. The need for this value comes from the fact that the actors in parlament change over time. For instance old commities are closed and new ones opened, elections change the politicians and some politicians decide to leave their party and maybe even join a new one. That means that a given indiviual person can be present many times in the dataset - once for each period of time. To make sure that we get consistent view of the parlament, we choose a single period thereby making sure that each person is only represented once.

[6]:
```
periods = [i for i in range(139, 148)] #period from the election in 139 to now
groups = actors[actors['periodeid'].isin(periods)]
groupIds = groups['id'].values
```

Looking into the data, one discovers, that almost all of the relations is from a person to some kind of group - most often commities on some subject like health, economy or transportation.

To build a network of relations we therefore extract what people that are in what commities (and other parlamentary groups).

[7]:
```
# It seems to be that the `tilaktørid` always specifies the Actor id of the␣
↪group
```

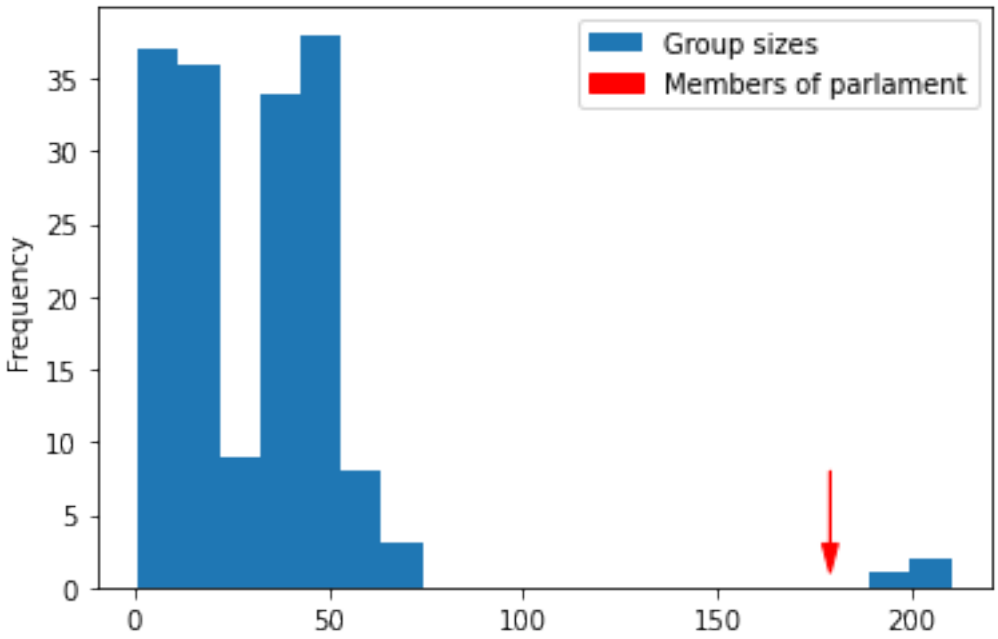
```
# and the `fraaktørid` always specifies the person when linking a person to a␣
 ↪group that they are a member of.
relations = actor_relations[(actor_relations['tilaktørid'].isin(groupIds))
                            & (actor_relations['fraaktørid'].isin(personIds))]
```

These relations can then be grouped by the individual, making a list for each group of all the people that are in the group.

[8]:
```
group_persons = relations[['tilaktørid','fraaktørid']]\
    .groupby(['tilaktørid'])\
    .agg(lambda tdf: tdf.unique().tolist())

all_persons_in_period = list(set(group_persons.fraaktørid.agg(sum)))
```

A network could then be specified by having an edge of weight 1 between all members of each group. This approach does however yield some problems as some of the groups are very big (one of them being the entire parlament including some suppleants making a group larger than the 179 members of parlament).

[9]:
```
import matplotlib.pyplot as plt
group_persons.fraaktørid.map(len).plot(kind='hist', bins=20, label="Group␣
 ↪sizes")
plt.arrow(x=179, y=8, dx=0, dy=-5, head_width=4, head_length=2, color="red",␣
 ↪label="Members of parlament")
plt.legend()
plt.show()
```



To make investigation of the network easier, we make some subsets of the groups that can be used to investigate the group sizes.

[10]:
```
medium_groups = group_persons[ group_persons.fraaktørid.map(len) < 55]
small_groups = group_persons[ group_persons.fraaktørid.map(len) < 26]
small_groups.sample(3)
```

[10]:
```
                                                  fraaktørid
tilaktørid
16095                                              [18, 366]
17613       [17827, 16728, 17628, 18278, 33, 15788, 199, 2…
17612         [86, 59, 123, 152, 112, 55, 141, 116, 191, 100]
```

To make the network possible to understand, we would rather show the name of a person that the persons corrosponding id. For that purpose, we define a mapping function from id to name.

[11]:
```
actor_names = dict(zip(actors['id'], actors['navn']))
def getActorName(aid):
    return actor_names.get(aid, 'Unnamed ' + str(aid))
```

We then go through the groups to create links.

[12]:
```
import itertools
edges = []

for personIds in medium_groups['fraaktørid']:
    combos = list(itertools.combinations(personIds, 2))
    combos_weighted = [(min(n1,n2), max(n1,n2), 1) for n1, n2 in list(combos)]
    edges.extend(combos_weighted)

# Unidirectional graph
edges = [(min(n1,n2), max(n1,n2), w) for (n1,n2, w) in edges]
```

And combine multiple edges between people into a single higher weighted edge (by adding the weights of all edges connecting them).

[13]:
```
from itertools import groupby
weighted_edges = [
    (getActorName(edge[0]),
     getActorName(edge[1]),
     sum(j for n1, n2, j in data))
        for edge, data in groupby(edges, key=lambda v: (v[0], v[1]))
]
```

This enables us to define a network using almost all the groups defined in the dataset.

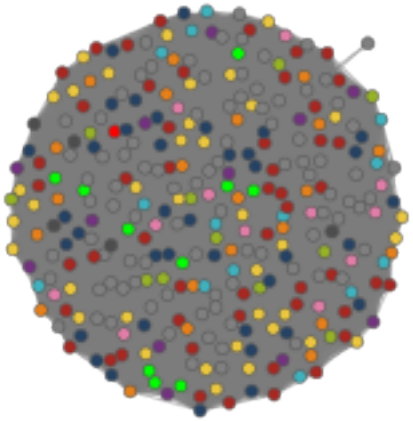
[14]:
```
G_medium = nx.Graph()
G_medium.add_weighted_edges_from(weighted_edges)
```

Each party in Folketinget has a color associated with them. We color people by their party color is avaiable.

[15]:
```
colorBy = 'party'
colors = {
    'S': '#a82721',
    'V': '#254264',
    'O': '#eac73e',
    'DF': '#eac73e',
    'RV': '#733280',
    'SF': '#e07ea8',
    'EL': '#e6801a',
    'KF': '#96b226',
    'NB': '#127b7f',
    'LA': '#3fb2be',
    'KD': '#8b8474',
    'ALT': '#00FF00',
    'CD': '#a70787',
    'IA': '#ff0000',
    'UFG': '#4d4d4d'
}

actorColors = { name: colors.get(colorId, 'grey') for name, colorId in␣
 ↪actors[['navn', colorBy]].values }
nx.set_node_attributes(G_medium, actorColors, 'group')
```

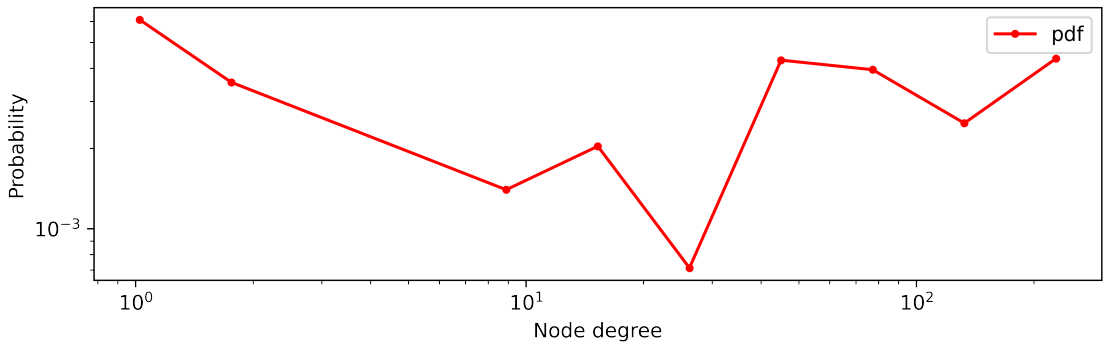
We can then visualize the network.

[16]:
```
_ = nw.visualize(G_medium)
```



The theory is that this network should represent some kind of social network, as people in the same groups might know one another. If that were the case, we would expect the degree distribution power law distributed. This is easy to check by plotting the data in log-log and checking if the line is linear (which it clearly is not).

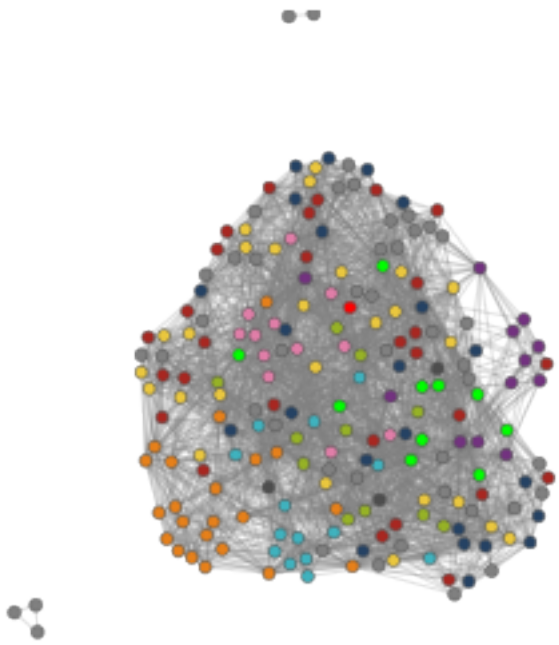
[17]:
```
from custom_plots import degree_distribution_histogram

degree_distribution_histogram([v for person, v in G_medium.
 ↪degree(weight='weight')])
```



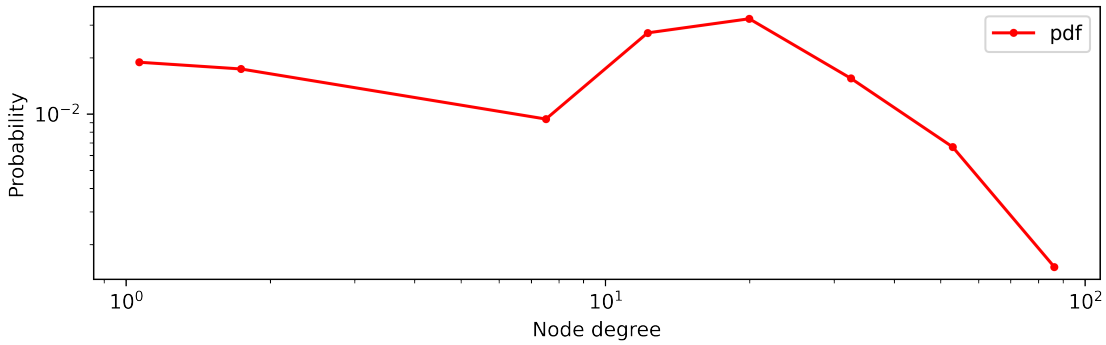
This might be due to several different reasons, but one of them is that a group of 50 people is not certain to know each other. To check if this might be the case, we repeat the plotting proces with all the big groups cutted off.

[18]:
```
import itertools
from itertools import groupby

edges = []

for personIds in small_groups['fraaktørid']:
    combos = list(itertools.combinations(personIds, 2))
    combos_weighted = [(min(n1,n2), max(n1,n2), 1) for n1, n2 in list(combos)]
    edges.extend(combos_weighted)

# Unidirectional graph
edges = [(min(n1,n2), max(n1,n2), w) for (n1,n2, w) in edges]


weighted_edges = [
    (getActorName(edge[0]),
     getActorName(edge[1]),
     sum(j for n1, n2, j in data))
        for edge, data in groupby(edges, key=lambda v: (v[0], v[1]))
]

colorBy = 'party'

G_small = nx.Graph()
G_small.add_weighted_edges_from(weighted_edges)

actorColors = { name: colors.get(colorId, 'grey') for name, colorId in␣
 ↪actors[['navn', colorBy]].values }
nx.set_node_attributes(G_small, actorColors, 'group')
```

[19]:
```
_ = nw.visualize(G_small)
```



[20]:
```
from custom_plots import degree_distribution_histogram

degree_distribution_histogram([v for person, v in G_small.
 ↪degree(weight='weight')])
```



This might seem somewhat more linear at the end, but still this is not anywhere near what a normal social network distribution looks like. We expect that this might be because the data about commities politicians work in severely lack information about anything that does not happen within the public scene in parlament. The next section will try to connect the politicians with more insight

into what they have done around parlament.

The work background based network Each member of parlament has the option to write a biography - a discription of them as a person. Most of them do this, and it is shown both in the `Aktør` table in the database and on the website of the Danish parlament www.ft.dk.

A description is an XML-encoded document with a lot of properties, where we are primarily interested in those that define what the politician has done in their career, as that might give us some insight into who they might have met and have some kind of relationship with. The documents define 4 types of "work" that a politician has done * `positionoftrust` - A title within a party of position of resposibility within the state * `parliamentarypositionoftrust` - Like `positionoftrust`, but specifically within FT * `occupation` - Some kind of "normal" not neccesarrily political work * `nomination` - A nomination from a party to some kind of election

We will first try to parse these biographies for relevant politicians.

[21]:
```
from collections import Counter
import networkx as nx
import netwulf as nw
import pandas as pd

# Extracting the people
persons = actors[(actors['type'].isin(['Person', 'Privatperson'])) & (actors.id.
 ↪isin(all_persons_in_period))]
```

The information we wan't about the jobs that each politician has held is: * What their title was * Where they worked * When they started working there (we will limit ourselves to a year) * When they stopped working there

As each individual politician is able to write in free text, we will need to parse this somehow. Due to timeconstraints we need to leave some of the jobs unparsed when the politicians get particularily imaginative about the text they write in their biographies. Fortuneatly it seems like most of these have been written by a few people (probably some secretary) as a lot of them are formatted in a similar way.

The "parsing" below is the result of looking a lot at different examples of how the profiles were written. We have tried to describe how each part of the code works - but to really understand the what happens one needs both a basic understanding of Danish politics and of the specifics of how the data looks.

[22]:
```
import re
from bs4 import BeautifulSoup
from timeout import timeout
import os
import string

def match_year(s):
    if (r := re.fullmatch(r'(\d{4})', s)):
        return r.group(1)
    else:
```

```
        return False

date_range_regex = re.compile('(\d{4})-(\d{4})')
def match_year_range(s, strict=True):
    if (r := date_range_regex.fullmatch(s) if strict else date_range_regex.
 ↪match(s)):
        return (r.group(1), r.group(2))
    else:
        return False


org_regex = re.compile(r'((?:[0-9a-zæøåA-ZÆØÅ/&-]+\s?)+)')

def match_organization(s):
    # The characters here has been found by trial and error
    # For instance the forward slash (/) is needed because some Danish companies
    # end their name with A/S if they are
    pattern = r'[^\.\s0-9a-zæøåA-ZÆØÅ/&-]'
    if not re.search(pattern, s):
        return s.strip()
    else:
        return False

title_regex = re.compile(r'((?:[0-9a-zæøåA-ZÆØÅ/&-]+\s?)+)')

def match_title(s):
    pattern = r'[^\.\s0-9a-zæøåA-ZÆØÅ\'/&-]'
    if not re.search(pattern, s):
        return s.strip()
    else:
        return False

def match_regex(s):
    r = re.search("^(.+)\s{1}(?:af|for|i|,)\s{1}(.+)\xa0(\d{4})-(\d{4})", s)
    if r:
        return {
        'organization': r.group(2),
        'title': r.group(1),
        'year_from': int(r.group(3)),
        'year_to': int(r.group(4)),
        'case_matched': 'match_regex'
    }
    return False

def comma_regex(s):
    r = re.match("^(.+)\s?,\s(.+),\xa0(\d{4})-(\d{4})", s)
    if r:
```

```
        return {
        'organization': r.group(2),
        'title': r.group(1),
        'year_from': int(r.group(3)),
        'year_to': int(r.group(4)),
        'case_matched': 'comma_regex'
    }
    return False

def current_position_regex(s):
    r = re.match("^(.+)\s(?:af|for|i)\s(.+)\xa0fra\s(\d{4})", s)
    if r:
        return {
        'organization': r.group(2),
        'title': r.group(1),
        'year_from': int(r.group(3)),
        'year_to': 2021,
        'case_matched': 'current_position_regex'
    }
    return False

def match_candidacy(s):
    """
        Trying to match things like
            "Kandidat for Socialdemokratiet i Åbenråkredsen 2000-2007."
        returning the party "Socialdemokratiet" and the district␣
 ↪"Åbenråkredsen" and the period (2000, 2007).
    """
    possible_initials = [
        ("Medlem af", "Medlem"),
        ("Kandidat for", "Kandidat"),
        ("Kandidat ved", "Kandidat"),
        ("Faglig sekretær", "Faglig sekretær")
    ]

    title = False
    for init in possible_initials:
        if s.startswith(init[0]):
            title = init[1]
            title_text = init[0]
    if not title:
        return False


    stripped_first_info = s.replace(title_text, "").strip()

    split_words = [
```

```
        " i ",
        " for ",
        " af ",
    ]

    party = False
    for split_word in split_words:
        if split_word in stripped_first_info:
            try:
                party, remaining_info = stripped_first_info.split(split_word)
            except:
                return False

    if not party:
        return False



    party = party.strip()
    remaining_info.strip(" .")
    if (r := re.search(r'(\d{4})-(\d{4})', remaining_info)):
        years = (int(r.group(1)), int(r.group(2)))
        data = f"{years[0]}-{years[1]}"
    elif (r := re.search(r'(?:fra\s)?(\d{4})', remaining_info)):
        years = (int(r.group(1)), int(r.group(1)))
        data = r.group(1)
    else:
        return False

    district = remaining_info.replace(data, "").strip(" .")

    return {
        'organization': party,# + " - " + district,
        'title': title,
        'year_from': years[0],
        'year_to': years[1],
        'case_matched': 'match_candidacy'
    }
```

With the parsing functions defined above, we can now try to parse the work lifes of all the politicians in the period we are investigating.

[23]:
```
types = ["parliamentarypositionoftrust", "occupation", "positionoftrust",␣
 ↪"nomination"]
parsed = []
unparsed = []
```

```
for index, a in persons[~persons.biografi.isna()].iterrows():
    xml = a["biografi"]
    soup = BeautifulSoup(xml) # from_encoding seemed to be specified by the␣
↪provided data being unicode
    for work_type in types:
        for obj in soup.findAll(work_type):
            add = True
            result = {
                "person_id": a.id,
                "name": a.navn,
                "work_type": work_type,
            }
            content = obj.contents[0].strip(" \t")
            splitted_comma = [ c.strip().strip(".") for c in content.split(",")␣
↪]
            splitted_nonbreakingspace = [ c.strip().strip(".") for c in content.
↪split("\xa0") ]

            if len(splitted_comma) == 3 and match_title(splitted_comma[0]) and␣
↪match_organization(splitted_comma[1]) and␣
↪match_year_range(splitted_comma[2], strict=False):
                result["title"] = match_title(splitted_comma[0])
                result["organization"] = match_title(splitted_comma[1])
                result["year_from"] = int(match_year_range(splitted_comma[2],␣
↪strict=False)[0])
                result["year_to"] = int(match_year_range(splitted_comma[2],␣
↪strict=False)[1])
                result["case_matched"] = "3-comma"
            elif (cancidacy_match := match_candidacy(content)):
                result.update(cancidacy_match)
            elif (regex_match := match_regex(content)):
                result.update(regex_match)
            elif (comma_regex_match := comma_regex(content)):
                result.update(comma_regex_match)
            elif (current_position_match := current_position_regex(content)):
                result.update(current_position_match)
            elif len(splitted_nonbreakingspace) == 2 and␣
↪match_title(splitted_nonbreakingspace[0]) and␣
↪match_year_range(splitted_nonbreakingspace[1], strict=False):
                result["title"] = match_title(splitted_nonbreakingspace[0])
                result["organization"] = soup.find("party").contents[0]
                result["year_from"] =␣
↪int(match_year_range(splitted_nonbreakingspace[1], strict=False)[0])
                result["year_to"] =␣
↪int(match_year_range(splitted_nonbreakingspace[1], strict=False)[1])
                result["case_matched"] = "2-vbspace"
```

```
            else:
                add = False
                result["content"] = obj.contents[0]
                unparsed.append(result.copy())
            if add:
                parsed.append(result.copy())
```

We can then check how many of the jobs that we were able to parse - and how many we weren't.

[24]:
```
len(unparsed), len(parsed)
```

[24]:
```
(316, 2184)
```

Some organizations have multiple different ways of being refered to. Especially in this dataset the political party "The Socialist Peoples Party" which in Danish is "Socialistisk Folkeparti" but is often shortened to just "SF". This applies both to their party organization and their youth organisation. Further a lot of them wrote very precisely what political position they held within a party - for instance one could have written "Formand for hovedbestyrelsen i sf ungdom" which would translate to "President of the main board in the Socialist Peoples Party Youth organization". As the youth organizations are fairly small in Denmark, we have found it to be a fair assumptions, that people who have volunteered there (and later became professional politicians) probably knew each other. We will therefor shorten any written organization containing the full string corrosponding to common fairly small Danish political group to just that group. That is in our example, the organization would be shortened to just "sf ungdom". Further we do things like removing commas and trimming to make sure that two organizations match even if the formatting is a little weird.

[45]:
```
import pandas as pd
jobs = pd.DataFrame(parsed)
jobs[jobs.case_matched == "match_regex"]

organizations = [
    "danmarks socialdemokratiske ungdom",
    "radikal ungdom",
    "venstres ungdom",
    "sf ungdom",
    "metal ungdom",
    "sfs ungdom",
    "dansk ungdoms fællesråd",
    "konservativ ungdom",
    "liberal alliances ungdom",
    "dansk folkepartis ungdom",
]

def organization_cleaner(s):
    # This special case is important to handle, as SF
    # is a major political party in Denmark
    if "socialistisk folkeparti" in s:
        s = s.replace("socialistisk folkeparti", "sf")
```

```
    for org in organizations:
        if org in s:
            # They simply couldn't decide if they had an s
            # at the end or not.. The trailing s in Danish
            # is simply a grammatical genetive construction
            if org == "sfs ungdom":
                return "sf ungdom"
            return org

    if (r := re.search(r"\(.+\)", s)):
        s = s.replace(r.group(), "")
    s = s.replace(",", "").strip()

    return s
```

We then run the cleaner over all columns.

[26]:
```
jobs["organization_original"] = jobs.organization
jobs.organization = jobs.organization.str.lower().map(organization_cleaner)
```

We will then define the network from the jobs that we just found. We will link to people if they worked the same place at the same time. The strength of the connection will be defined as the amount of years that they worked together that place. That is, if Person A worked at Org X from 2010 to 2012 and Person B worked there from 2011 to 2017, then we will make a connection between them with weight 2 since the worked together in 2011 and 2012 at Org X.

[27]:
```
def year_overlap(years_1, years_2):
    assert years_1[0] <= years_1[1]
    assert years_2[0] <= years_2[1]
    s1, e1 = years_1
    s2, e2 = years_2
    return max(0, 1 + min(e2, e1) - max(s2, s1))

def connection_strength(job1, job2):
    if job1.person_id == job_2.person_id:
        return 0
    if job1.organization == job2.organization:
        return year_overlap( (job1.year_from, job1.year_to), (job2.year_from,␣
 ↪job2.year_to))
```

[28]:
```
from itertools import combinations

links = []


for org in list(set(jobs.organization.values)):
```

```
    to_connect = combinations([ job for index, job in jobs[ jobs.organization␣
↪== org ].iterrows()], 2)
    for job_1, job_2 in to_connect:
        strength = connection_strength(job_1, job_2)
        if strength != 0:
            links.append((job_1["name"], job_2["name"], strength))
```

These can then be defined as a network.

[29]:
```
import networkx as nx
G = nx.Graph()
G.add_weighted_edges_from(links)
```

[33]:
```
len(G.nodes()), len(G.edges)
```

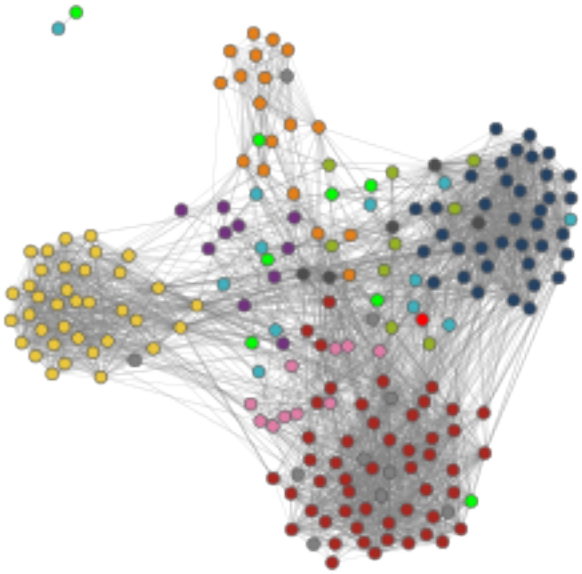
[33]:
```
(207, 3044)
```

We then add colors based on their party.

[30]:
```
colorBy = 'party'
colors = {
    'S': '#a82721',
    'V': '#254264',
    'O': '#eac73e',
    'DF': '#eac73e',
    'RV': '#733280',
    'SF': '#e07ea8',
    'EL': '#e6801a',
    'KF': '#96b226',
    'NB': '#127b7f',
    'LA': '#3fb2be',
    'KD': '#8b8474',
    'ALT': '#00FF00',
    'CD': '#a70787',
    'IA': '#ff0000',
    'UFG': '#4d4d4d'
}

actorColors = { name: colors.get(colorId, 'grey')
                    for name, colorId
                    in actors[['navn', colorBy]].values
              }
nx.set_node_attributes(G, actorColors, 'group')
```

And we can the visualize the network.

[31]:
```
import netwulf
_ = netwulf.visualize(G)
```



It here becomes very apparent that the big parties become very tightly connected in the graph. That really isn't a surprise, as a lot of the Danish politicians have very long political careers typically only in a single party. This will bind them very closely with all the other politicians with the same property.

We could then try to take a look at who the most influential people are. That is of course a very semantic question, but it is kind of like the initial question answered by Google when ranking web pages. There they defined a web page to be important if other important websited were linking to it. If we defined a similar metric and said someone to be well-connected if the were connected to other well-connected people, we could used the Page-Rank algorithm to apply some sort of score to measure how well placed the politician is (this algorithm i predefined in the NetworkX libary).

[108]:
```
page_ranks = list(nx.algorithms.link_analysis.pagerank_alg.pagerank(G).items())
page_ranks.sort(key=lambda v: v[1], reverse=True)
page_ranks[:10]
```

[108]:
```
[('Søren Espersen', 0.012987147848913128),
 ('Lars Løkke Rasmussen', 0.012219728059896693),
 ('Nick Hækkerup', 0.012068496306769946),
 ('Kristian Thulesen Dahl', 0.01153811749804906),
 ('Mette Frederiksen', 0.011006862806093481),
 ('Mogens Jensen', 0.010917006226369307),
```

```
('Lars Christian Lilleholt', 0.010763514592769929),
('Inger Støjberg', 0.010633842721172184),
('Peter Skaarup', 0.0102073396456307),
('Mogens Lykketoft', 0.00996430636679063)]
```

For the newcomer to the Danish politican system, these are very influential people that score well on the list. Lars Løkke Rasmussen was the Danish prime minister at the time, Kristian Thuelsen Dahl was (and is at the time of writing) the leader of the Danish Peoples Party (the yellow one in the graph) which was the party providing the votes for Løkke to become prime minister. Almost all the people on the list have at some point been (or are today) ministers and one of them, Mette Frederiksen, is at the time of writing the new Danish prime minister.

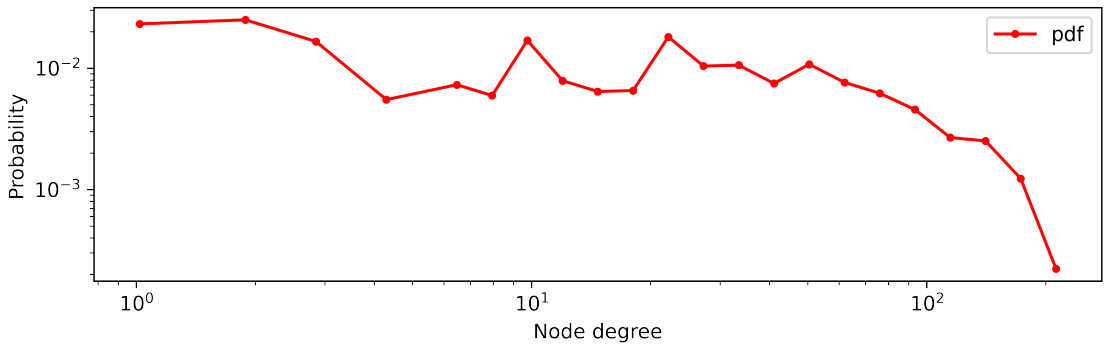
Using Page-Rank wasn't really neccesarry as we get a very similar result by just finding the degree of each node.

[107]:
```
dgs = list(G.degree(weight='weight'))
dgs.sort(key=lambda v: v[1], reverse=True)
dgs[:10]
```

[107]:
```
[('Lars Løkke Rasmussen', 213),
 ('Søren Espersen', 193),
 ('Nick Hækkerup', 190),
 ('Inger Støjberg', 178),
 ('Lars Christian Lilleholt', 177),
 ('Mette Frederiksen', 175),
 ('Kristian Jensen', 170),
 ('Kristian Thulesen Dahl', 168),
 ('Mogens Jensen', 165),
 ('Bertel Haarder', 158)]
```

We can then take a look into if the distribution is power law distributed. We again plot the degree histogram in log-log.

[44]:
```
degree_distribution_histogram(list(dict(G.degree(weight='weight')).values()),␣
↪bins=50)
```

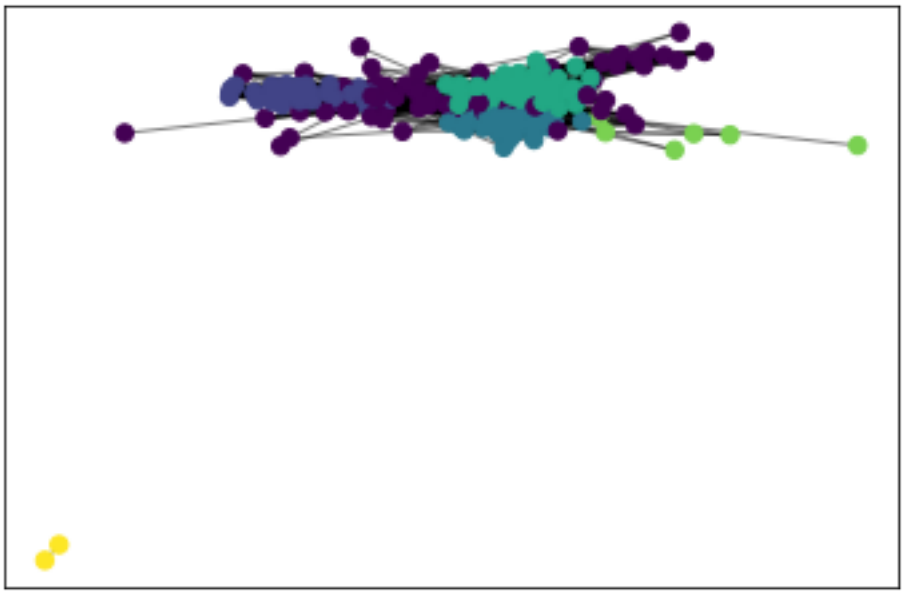


The line doesn't look quite linear, but it begins to look like it at the end. This can possibly be attributed to the fact that most people in the network are at least connected to the other people in their party - making it more unlikely for an individual to have a very low amount of connections.

Community detection As we have now found a network that seems to in some way measure how well-connected people are, we can now look at how they are distributed into communities. The hope of this exercise is to see if there were surprising communities, that could tell us something about how the people in power work.

For the purpose of community detection, we will just use the Louvain Partitioning. This choice is simply made to

[109]:
```
import community as community_louvain
import matplotlib.cm as cm
import matplotlib.pyplot as plt

# There is some level om randomness in this partitioning algorithm
# Hence we pin the random state to make the results repeatable.
partition = community_louvain.best_partition(G, random_state=420)
pos = nx.spring_layout(G)
cmap = cm.get_cmap('viridis', max(partition.values()) + 1)

nx.draw_networkx_nodes(G, pos, partition.keys(), node_size=40,
                       cmap=cmap, node_color=list(partition.values()))
nx.draw_networkx_edges(G, pos, alpha=0.5)
plt.show()
```



We need them in a another format than what the `best_partition` provides.

[79]:
```
louvain_partitions = [ [person for person, group in partition.items() if group
 ↪== i]
    for i in set(partition.values())
]
```

We can then check what modularity this partitioing gives rise to.

[80]:
```
import networkx.algorithms.community.quality as nxquality

nxquality.modularity(G, louvain_partitions)
```

[80]:
```
0.5515707658787116
```

To give some context for this we will first calculate the more natural partitioning of political parties.

[81]:
```
import numpy as np
network_degree = pd.DataFrame(G.degree, columns=["name", "degree"])

party_lookup = dict(zip(persons.navn, persons.party))


party_partitions = [ [person for person in sum(louvain_partitions, start=[]) if
 ↪party_lookup[person] == i]
    for i in set(party_lookup.values())
]

nxquality.modularity(G, party_partitions)
```

[81]:
```
0.4475969255799385
```

We can then compare these two partitions further by making a confusion matrix.

[118]:
```
D = np.zeros((len(louvain_partitions), len(party_partitions)))

i = 0
for c1 in louvain_partitions:
    j=0
    for c2 in party_partitions:
        overlap = len(set(c1).intersection(c2))
        D[i][j] = overlap
        j += 1
    i += 1

Ddf = pd.DataFrame(D, columns=set(party_lookup.values()))

s = Ddf.sum()
s = s[(s.index.notnull()) & (s.index != "") & (s.values != 0)]
Ddf[s.sort_values(ascending=False).index]
```

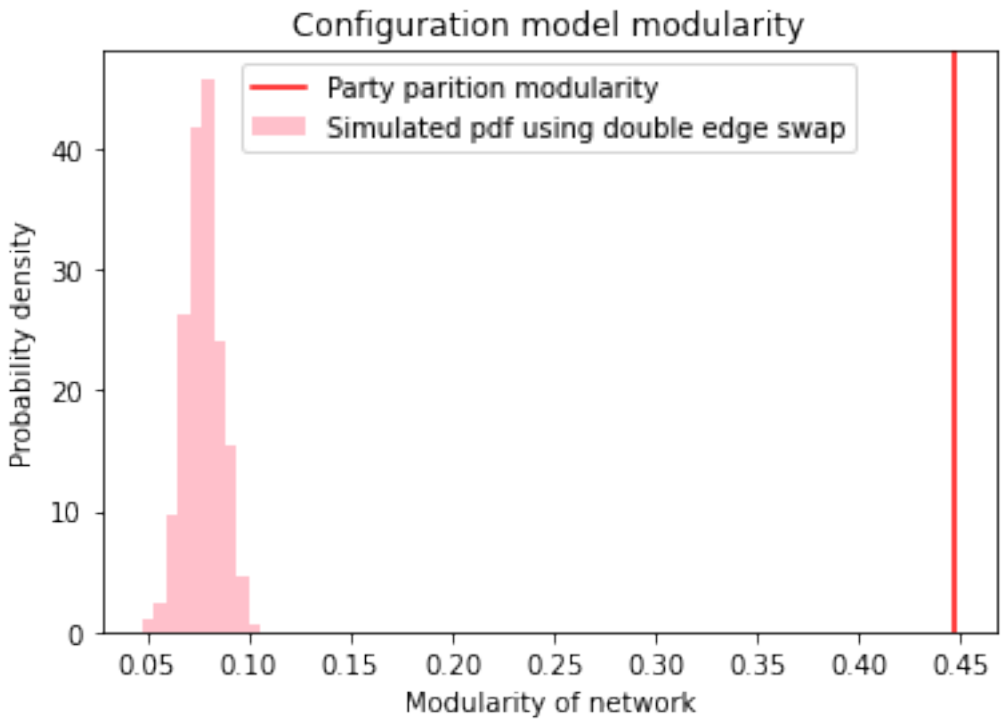
[118]:

	S	V	DF	EL	LA	SF	RV	KF	ALT	UFG	JF	IA
0	5.0	1.0	0.0	19.0	6.0	10.0	10.0	7.0	4.0	3.0	0.0	1.0
1	0.0	0.0	37.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	36.0	0.0	0.0	1.0	0.0	0.0	2.0	0.0	2.0	0.0	0.0
3	51.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0
4	0.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0

Here it becomes clear that a lot of the partitions are simply primarily the political parties. One of them though seems to be a collection of a lot of different parties (group 0). Later we will try to investigate this group and see what is characteristic about them.

To check if these two partitions are indeed better than random (as measured by modularity), we can try to define a configuration model and check how well the same partitions work on networks. A double-edge-swap configuration seems appropriate for this purpose.

[132]:
```
def shuffle_graph(G: nx.classes.graph.Graph):
    G_copy = G.copy()
    return nx.double_edge_swap(G_copy, len(G.edges), max_tries=10000)

scores = []
for i in range(300):
    G_swapped = shuffle_graph(G)
    scores.append(nxquality.modularity(G_swapped, [ c for c in party_partitions␣
 ↪]))
sum(scores)/len(scores)
plt.hist(scores, bins=10, density=True, color='pink', label='Simulated pdf␣
 ↪using double edge swap')
plt.axvline(nxquality.modularity(G, party_partitions), color='r', label='Party␣
 ↪partition modularity')
plt.ylabel("Probability density")
plt.xlabel("Modularity of network")
plt.title('Configuration model modularity')
plt.legend()
plt.show()
```



As expected the party partitioning is way better than random when using the double-edge-swap configuration model.

[134]:
```
# For export to the website
Ddf[s.sort_values(ascending=False).index].columns.tolist(), Ddf[s.
 ↪sort_values(ascending=False).index].values.tolist()
```

[134]:
```
(['S', 'V', 'DF', 'EL', 'LA', 'SF', 'RV', 'KF', 'ALT', 'UFG', 'JF', 'IA'],
 [[5.0, 1.0, 0.0, 19.0, 6.0, 10.0, 10.0, 7.0, 4.0, 3.0, 0.0, 1.0],
  [0.0, 0.0, 37.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
  [0.0, 36.0, 0.0, 0.0, 1.0, 0.0, 0.0, 2.0, 0.0, 2.0, 0.0, 0.0],
  [51.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0],
  [0.0, 0.0, 0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 0.0],
  [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]])
```

[135]:
```
# Export for display on the website
import pickle
with open("louvain_partitions.pkl", "wb") as f:
    pickle.dump(louvain_partitions, f)
```

[42]:
```
# Export for display on the website
from graph2json import graph2json
graph2json(G, 'network')
```

1.7 Language Analysis

Having now analyzed how the politicians are grouped into communities, we can move on to analyzing each group. As the topic is politics, the most important thing that the politicians are doing is speaking. Hence we find it inherently important to analyze the language that they are using.

Since the language in the speeches are almost exclusively Danish, we needed to adapt our analysis tools for the purpose of analysing Danish texts. Fortuneatly the sentiment analysis package Sentida (https://tidsskrift.dk/lwo/article/view/115711) is created just for this purpose. For stopwords we are using the ones built in to `nltk` for the Danish language. Sometimes we will add some stop words such as "Hr." and "Fru" which mean "Mr." and "Mrs." respectively. These words are extremely common in the speeches, as the language in the parliament perscribes that the members speak formally epspecially when adressing one another.

The data analyzed here are speeches from within parlament. As all speeches are transcribed and marked with who the speaker was, we have been able to analyze it without having to do a lot of manual tagging. As with the rest of the report, we are limiting the timeframe to speeches held within the election period between 2015 and 2019.

[136]:
```
# To install the sentida module
# `pip install sentida`
```

[2]:
```
from sentida import Sentida
import re
import pandas as pd
```

In this notebook we limit ourselves to the first year of the election period (as this makes the code way more readable). The data presented on out website uses all of the data from the election period.

[3]:
```
votingString = "2015"
speechString = "20151"
```

[4]:
```
meetings = pd.read_csv("data/speeches/" + speechString + ".csv")
grouped = meetings[(meetings.role == "medlem")].groupby("party")
party_docs = dict(grouped.aggregate("text").sum())
party_docs_count = dict(grouped.aggregate("meeting").count())
```

[5]:
```
grouped_person = meetings.groupby("name")
person_docs = dict(grouped_person.aggregate("text").sum())
person_docs_count = dict(grouped_person.aggregate("meeting").count())
```

In Denmark people usually vote for a party rather than an individual politician within that party. That makes it insteresting to understand what that party is actually talking about in parliament - and more importantly what makes them different from the other parties. Further it is interesting to understand how other metrics about how they speak. Here we have chosen to investigate the LIX-numbers for their speeches and the sentiment.

We will however also investigate each individual politician. To follow up on the former section on network analysis, we will think of the louvain partitioning as another type of party. This might give

us some insight into how these grops can be intepreted (perhaps they have a commonality that has bound their working life together, that is prevalent in the words that they use in their speeches).

1.7.1 Sentiment Analysis Per Party - Sentida

This section will go into sentiment analysis on a political party by party basis.

First we define a sentiment analysis function to be used throughout the entire section on sentiment analysis.

[6]:
```
def _sentiment_analysis_helper(text, output):
    return Sentida().sentida(text, output=str(output), normal = True, speed =␣
↪"normal")
```

[7]:
```
def SentAnalysis(collection, columnName: str, filename: str, party: bool,␣
↪analyse: bool):
    if(analyse):
        SentAnalysis = dict([(person, _sentiment_analysis_helper(doc, "total"))␣
↪for (person, doc) in collection.items()])
        pd.DataFrame(list(SentAnalysis.items()),columns =␣
↪[str(columnName),'Sentiment score']).to_csv("data/AppData/" + speechString +␣
↪"/"+str(filename)+".csv")

    person_sent_df = pd.read_csv("data/AppData/" + speechString + "/
↪"+str(filename)+".csv", index_col = False)

    #By using the output of "total" each partys sentiment becomes accumulative.
    #This way parties that speak more are typically rewarded as such.
    #Therefore the average sentiment of each party is found below:

    if(party):
        meetingsPartyText = meetings.groupby(meetings["party"]).
↪aggregate("text").sum()
        person_sent_df["Percentage sentiment"] = person_sent_df.apply(lambda␣
↪row: (row["Sentiment score"]/len(meetingsPartyText[row["party"]]))*100,␣
↪axis=1)

    if(party== False):
        meetingsPartyText = meetings.groupby(meetings["name"]).
↪aggregate("text").sum()
        person_sent_df["Percentage sentiment"] = person_sent_df.apply(lambda␣
↪row: (row["Sentiment score"]/
↪len(meetingsPartyText[row[str(columnName)]]))*100, axis=1)

    person_sent_avg =␣
↪dict(zip(person_sent_df[str(columnName)],person_sent_df["Percentage␣
↪sentiment"]))
```

```
    person_sent =␣
↪dict(zip(person_sent_df[str(columnName)],person_sent_df["Sentiment score"]))
    return(person_sent_avg, person_sent)
```

Analysing sentiment for each party:

[11]:
```
[party_sent_avg, party_sent] = SentAnalysis(party_docs, "party", "partySent",␣
↪True, True)
```

```
IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.

Current values:
NotebookApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
NotebookApp.rate_limit_window=3.0 (secs)


7184.7924733564105
9816.98418920359
11673.655360098595
1081.7798095238113
275.70485978835956
3637.448697178312
5573.353110486202
4968.206200274834
10581.238756697141
5492.370928078793
594.6681777777777
362.2343121693122
62.18165343915343
7562.340034730559
Analysis done, saving…
Loading analysis…
Party
```

Average sentiment of each party

[12]:
```
dict(sorted(party_sent_avg.items(), key=lambda item: item[1], reverse=True))
```

[12]:
```
{'UFG': 0.6062362624466553,
 'IA': 0.4623405560002441,
 'SIU': 0.43676465283745336,
 'T': 0.37293378238596553,
 'ALT': 0.34253651555958825,
 'JF': 0.3314357874476884,
 'RV': 0.3136295814831661,
```

```
 'KF': 0.29638391830185806,
 'V': 0.27963521243779244,
 'S': 0.2676054883733499,
 'LA': 0.2618698810682469,
 'SF': 0.2574555308925206,
 'DF': 0.24408342731783503,
 'EL': 0.217731632712758}
```

Total sentiment of each political party

[13]:
```
dict(sorted(party_sent.items(), key=lambda item: item[1], reverse=True))
```

[13]:
```
{'EL': 11673.655360098595,
 'S': 10581.23875669714,
 'DF': 9816.98418920359,
 'V': 7562.340034730559,
 'ALT': 7184.79247335641,
 'LA': 5573.353110486202,
 'SF': 5492.3709280787925,
 'RV': 4968.206200274834,
 'KF': 3637.4486971783117,
 'IA': 1081.779809523811,
 'SIU': 594.6681777777778,
 'T': 362.2343121693122,
 'JF': 275.7048597883596,
 'UFG': 62.181653439153436}
```

1.7.2 Sentiment Analysis Per Person - Sentida

The same sentiment analysis is done, but for every individual politician in Folketinget.

[ ]:
```
[person_sent_avg, person_sent] = SentAnalysis(person_docs, "person",␣
↪"personSent", True, True)
```

```
IOPub data rate exceeded.
The notebook server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--NotebookApp.iopub_data_rate_limit`.

Current values:
NotebookApp.iopub_data_rate_limit=1000000.0 (bytes/sec)
NotebookApp.rate_limit_window=3.0 (secs)


1081.7798095238113
656.8498312169315
327.29735767195695
135.79622751322756
```

41.67362219576721
338.3614743306881
18.952199999999998
38.109582010582
89.02599206349217
337.7573201058201
3.017023809523805
1012.9513675724913
567.1573535132258
225.19033068783074
175.67103650793644
945.094861997354
157.10117106746046
103.95293650793653
761.4346534285705
699.2572931216929
7.181455026455027
1741.7137698197869
355.3946117619045
824.6077788359763
162.3049874232804
393.68878306878304
13.729576719576716
727.6611873015873
160.99596560846567
431.7516682539676
37.07246031746031
614.8048244603174
27.366269841269855
373.04711904761893
115.41019841269852
292.1935296296295
728.3800192354527
716.9013927671948
654.8289111111119
360.5897779920635
1883.8315982844053
344.9801190476187
696.8405898730173
684.9660444444436
12.63841269841271
421.59314524867756
353.9652241507939
992.499542537037
55.45617724867727
178.17100793650792
681.0235742989469
46.704576719576714

655.8272544801582
81.73535714285711
482.7746114021155
924.54933533598
785.1026531111107
853.4364120714298
322.46726810026365
283.93728359788327
337.0871957671954
694.8309142169313
47.255158730158705
100.05843915343925
42.95738730158731
480.43420105820036
222.9903042328044
552.0277153439151
367.34202169312107
306.1911153439155
108.6643857142858
399.1663884973545
721.5515933862437
451.5038288346563
191.06306746031726
595.8334830910043
1475.3543878511923
104.99571428571431
958.9020920634929
175.99841269841264
911.854452380953
445.52269591005233
171.01475861640202
230.24276455026418
454.93768493915275
38.25330680158727
546.5822966349216
1107.6136026904771
157.87144179894182
746.3809044695751
443.34806288359766
40.193756613756605
1296.0930402161366
1452.4253266402154
414.04195925925865
5.301190476190473
836.6596915343904
697.0593399841283
477.20786766402153
299.35437301587257

598.9726542962953
331.18689682539684
136.21573174603193
1771.8442117484233
67.32354497354497
155.85261640211647
252.15904682539653
347.84840857142893
362.2343121693122
315.6955767195766
134.83783068783083
52.4854100529101
29.142089947089932
57.28041005291003
590.3118137566139
915.4288347592611
377.49577632804215
124.10582433862432
436.08923708068795
767.8962852857154
1637.8114593187852
465.48081704298903
217.4230232804233
54.66232275132275
399.66127174603173
896.7716439550244
1525.218986225997
275.3105515899471
49.826756769841275
168.14217989418003
389.44410029497317
283.50378350740755
314.4916957671957
151.52330687830698
67.85578042328044
316.1860582010579
37.08328042328042
352.292588888889
270.35041967724885
101.31556750317462
49.21305335714287
474.059325396825
388.1612728529085
562.0581386291002
206.65898148148136
-10.586666666666702
263.0518111111114
1345.1144307333318

```
41.26330158730159
170.42615873015868
179.510444015873
100.33296666666668
733.8186263015867
213.10624514814793
573.5575078015875
225.33176076455004
393.02088306878295
69.51180846560851
326.1544682539674
2446.5055529603037
512.1551975925929
125.42394814814826
52.65625661375659
210.262113904762
773.9950999455027
1507.2940070661384
745.3100410492058
18.175634920634916
1147.2368131851847
```

[12]:
```
person_sent = dict([(person, SentAnal(doc, "total")) for (person, doc) in␣
↪person_docs.items()])
```

```
14.805555555555564
1081.7798095238113
656.8498312169315
327.29735767195695
135.79622751322756
41.67362219576721
338.3614743306881
18.952199999999998
38.109582010582
89.02599206349217
337.7573201058201
3.017023809523805
1012.9513675724913
567.1573535132258
225.19033068783074
175.67103650793644
945.094861997354
157.10117106746046
103.95293650793653
761.4346534285705
699.2572931216929
7.181455026455027
1741.7137698197869
```

355.3946117619045
824.6077788359763
162.3049874232804
393.68878306878304
13.729576719576716
727.6611873015873
160.99596560846567
431.7516682539676
37.07246031746031
614.8048244603174
27.366269841269855
373.04711904761893
115.41019841269852
292.1935296296295
728.3800192354527
716.9013927671948
654.8289111111119
360.5897779920635
1883.8315982844053
344.9801190476187
696.8405898730173
684.9660444444436
12.63841269841271
421.59314524867756
353.9652241507939
992.499542537037
55.45617724867727
178.17100793650792
681.0235742989469
46.704576719576714
655.8272544801582
81.73535714285711
482.7746114021155
924.54933533598
785.1026531111107
853.4364120714298
322.46726810026365
283.93728359788327
337.0871957671954
694.8309142169313
47.255158730158705
100.05843915343925
42.95738730158731
480.43420105820036
222.9903042328044
552.0277153439151
367.34202169312107
306.1911153439155

108.6643857142858
399.1663884973545
721.5515933862437
451.5038288346563
191.06306746031726
595.8334830910043
1475.3543878511923
104.99571428571431
958.9020920634929
175.99841269841264
911.854452380953
445.52269591005233
171.01475861640202
230.24276455026418
454.93768493915275
38.25330680158727
546.5822966349216
1107.6136026904771
157.87144179894182
746.3809044695751
443.34806288359766
40.193756613756605
1296.0930402161366
1452.4253266402154
414.04195925925865
5.301190476190473
836.6596915343904
697.0593399841283
477.20786766402153
299.35437301587257
598.9726542962953
331.18689682539684
136.21573174603193
1771.8442117484233
67.32354497354497
155.85261640211647
252.15904682539653
347.84840857142893
362.2343121693122
315.6955767195766
134.83783068783083
52.4854100529101
29.142089947089932
57.28041005291003
590.3118137566139
915.4288347592611
377.49577632804215
124.10582433862432

436.08923708068795
767.8962852857154
1637.8114593187852
465.48081704298903
217.4230232804233
54.66232275132275
399.66127174603173
896.7716439550244
1525.218986225997
275.3105515899471
49.826756769841275
168.14217989418003
389.44410029497317
283.50378350740755
314.4916957671957
151.52330687830698
67.85578042328044
316.1860582010579
37.08328042328042
352.292588888889
270.35041967724885
101.31556750317462
49.21305335714287
474.059325396825
388.1612728529085
562.0581386291002
206.65898148148136
-10.586666666666702
263.0518111111114
1345.1144307333318
41.26330158730159
170.42615873015868
179.510444015873
100.33296666666668
733.8186263015867
213.10624514814793
573.5575078015875
225.33176076455004
393.02088306878295
69.51180846560851
326.1544682539674
2446.5055529603037
512.1551975925929
125.42394814814826
52.65625661375659
210.262113904762
773.9950999455027
1507.2940070661384

```
745.3100410492058
18.175634920634916
1147.2368131851847
1081.4472940793648
67.75931216931217
839.8975811957665
27.113571428571436
813.9996050396837
396.11733756613694
4.908571428571432
275.70485978835956
864.716444706349
782.4300343915346
54.123094179894174
409.2408778783066
138.88915767195783
965.3125153439166
223.24845090742798
58.556957671957626
497.2072165767193
664.2327394894177
716.6037915202393
468.94821428571356
9.32611111111111
6.095074074074067
578.2209976375658
424.56865911904714
287.79772126984074
552.069612142857
589.3919841269836
48.41011353968252
16.373333333333328
355.70735392857137
342.29642063492076
419.68111711640233
109.65145026455026
303.4652759878302
```

[13]:
```
pd.DataFrame(list(person_sent.items()),columns = ['Person','Sentiment score']).
 ↪to_csv("data/AppData/" + speechString + "/PersonSentimentTotal")
```

[14]:
```
person_sent_df = pd.read_csv("data/AppData/" + speechString + "/
 ↪PersonSentimentTotal", index_col = False)
meetingsPersonText = meetings.groupby(meetings["name"]).aggregate("text").sum()
person_sent_df["Percentage sentiment"] = person_sent_df.apply(lambda row:␣
 ↪(row["Sentiment score"]/len(meetingsPersonText[row["Person"]]))*100, axis=1)
```

```
person_sent_avg = dict(zip(person_sent_df["Person"],person_sent_df["Percentage␣
↪sentiment"]))
person_sent = dict(zip(person_sent_df["Person"],person_sent_df["Sentiment␣
↪score"]))
```

Average sentiment of persons in the danish parliment

[15]:
```
sorted(person_sent_avg.items(), key=lambda item: item[1], reverse=True)[:5]
```

[15]:
```
{'Jan Erik Messmann': 0.8252734671700788,
 'Sisse Marie Welling': 0.6770443349753699,
 'Daniel Toft Jakobsen': 0.5550600436810947,
 'Roger Matthisen': 0.5362402039356772,
 'Lars Christian Lilleholt': 0.5246535010154891,
 'Jan Rytkjær Callesen': 0.518871691044659,
 'Malou Lunderød': 0.5106376370613269,
 'Hans Christian Thoning': 0.5027212688310545,
 'Anders Johansson': 0.48940868386934644,
 'Pia Kjærsgaard': 0.48691522598473563,
 'Thomas Jensen': 0.4852295063013064,
 'Jan Johansen': 0.4844506592109966,
 'Jens Henrik Thulesen Dahl': 0.48028237574103344,
 'Merete Dea Larsen': 0.47429347289650975,
 'Anni Matthiesen': 0.46941381194087817,
 'Nikolaj Amstrup': 0.4662083763392717,
 'Aaja Chemnitz Larsen': 0.4623405560002441,
 'Kristian Pihl Lorentzen': 0.4492500237056033,
 'Aleqa Hammond': 0.44863727287543986,
 'Ulla Tørnæs': 0.44216917137292294,
 'Annette Lind': 0.4387462030628957,
 'Ane Halsboe-Jørgensen': 0.43688796680497927,
 'Maja Panduro': 0.42967998405984525,
 'Morten Marinus': 0.4185944285278295,
 'Ellen Trane Nørby': 0.41747478574679253,
 'Mette Hjermind Dencker': 0.4134711550046231,
 'Rasmus Helveg Petersen': 0.41256958876248995,
 'Brian Mikkelsen': 0.4091302037642062,
 'Erling Bonnesen': 0.4091149188574646,
 'Mette Bock': 0.40626030120901613,
 'Leif Mikkelsen': 0.4032520751809919,
 'Rasmus Nordqvist': 0.4027656364075446,
 'Lea Wermelin': 0.40161310072160855,
 'Hans Andersen': 0.39997967493385755,
 'Henrik Dam Kristensen': 0.39988466239919374,
 'Karin Gaardsted': 0.39739452204396997,
 'Per Husted': 0.39611501955746947,
 'Jens Joel': 0.3952484608594183,
```

'Uffe Elbæk': 0.39472838315654113,
'Carsten Bach': 0.39466595163124607,
'Orla Hav': 0.3934937479416618,
'Laura Lindahl': 0.39027108154013995,
'Anders Samuelsen': 0.3874093352771889,
'Jakob Engel-Schmidt': 0.38485880911108244,
'Torsten Gejl': 0.3845833040489297,
'Marianne Jelved': 0.38201501394298776,
'Karen Ellemann': 0.38136630073397676,
'Bertel Haarder': 0.3783121893117694,
'Mikkel Dencker': 0.37689404236088797,
'Eva Kjer Hansen': 0.3759785813257255,
'Thomas Danielsen': 0.37381582498522403,
'Christine Antorini': 0.37339071850902134,
'Magni Arge': 0.37293378238596553,
'Jørn Neergaard Larsen': 0.3693013722404489,
'Carl Holst': 0.36839229041015137,
'Kasper Roug': 0.3682476307969066,
'Dorthe Ullemose': 0.3680735688133134,
'Trine Torp': 0.36575919393582595,
'Karen J. Klint': 0.3629158012333233,
'Helle Thorning-Schmidt': 0.36144718079321336,
'Sophie Løhde': 0.3598636923207809,
'René Christensen': 0.35978085595277864,
'Mai Mercado': 0.3565534830574367,
'Christian Poll': 0.35649927535093023,
'Morten Løkkegaard': 0.35274449797074603,
'Troels Lund Poulsen': 0.3520397463457512,
'Alex Ahrendtsen': 0.35158483830184867,
'Flemming Møller Mortensen': 0.34864084795110534,
'Claus Kvist Hansen': 0.3484534891856929,
'Rasmus Prehn': 0.34693337974994215,
'Karin Nødgaard': 0.34679815720544077,
'Ib Poulsen': 0.34433735157289086,
'Julie Skovsby': 0.343909971456647,
'Pernille Bendixen': 0.33681059037580774,
'Kaare Dybvad': 0.3360479878914949,
'Martin Lidegaard': 0.3359192920103916,
'Magnus Heunicke': 0.335782059520067,
'Jane Heitmann': 0.33394097440584447,
'Lars Løkke Rasmussen': 0.33296839218337415,
'Carolina Magdalene Maier': 0.3327439654897942,
'Kirsten Brosbøl': 0.3327022317172139,
'Sjúrður Skaale': 0.3314357874476884,
'Christina Egelund': 0.3267642060314763,
'Merete Riisager': 0.32622746857075485,
'Simon Kollerup': 0.3255509201207608,

'Torsten Schack Pedersen': 0.3249739061126908,
'Sofie Carsten Nielsen': 0.323776526444685,
'Benny Engelbrecht': 0.32210973485009614,
'Karina Adsbøl': 0.32184280949895505,
'Johannes Lebech': 0.3211197960643326,
'Jacob Jensen': 0.31934603762141117,
'Marlene Harpsøe': 0.3179049083518167,
'Kristian Thulesen Dahl': 0.31749492692932135,
'Hans Christian Schmidt': 0.3173386910378387,
'Erik Christensen': 0.31721893110446037,
'Søren Pape Poulsen': 0.3168983776572801,
'Rasmus Vestergaard Madsen': 0.3155492173721341,
'Karsten Lauritzen': 0.3155072958572308,
'Esben Lunde Larsen': 0.314054396607218,
'Eva Flyvholm': 0.31240943060366205,
'Kristian Jensen': 0.31186669623480157,
'Mette Abildgaard': 0.31061114326907935,
'Mette Gjerskov': 0.31040840250337204,
'Mette Frederiksen': 0.3076301781012661,
'René Gade': 0.3072267264993096,
'Andreas Steenberg': 0.30608482910189344,
'Ida Auken': 0.3057741733922675,
'Carsten Kudsk': 0.30572392620072486,
'Yildiz Akdogan': 0.30492616870008415,
'Kim Christiansen': 0.30406500571549905,
'Louise Schack Elholm': 0.30354853532595855,
'Leif Lahn Jensen': 0.3025496422761941,
'Morten Bødskov': 0.3024550052296877,
'Søren Egge Rasmussen': 0.3011735774838516,
'Bjarne Laustsen': 0.30115723189319143,
'Jakob Sølvhøj': 0.30107286022685853,
'Hans Kristian Skibby': 0.2999318065555499,
'Mattias Tesfaye': 0.2995931139292332,
'Josephine Fock': 0.29901427786674517,
'Susanne Eilersen': 0.2983655374263326,
'Morten Østergaard': 0.2974790914907951,
'Mette Reissmann': 0.29627236648330385,
'Jakob Ellemann-Jensen': 0.2961512665542528,
'Claus Hjort Frederiksen': 0.29514694749843323,
'Peter Christensen': 0.29137155983558427,
'Christian Juhl': 0.28837657537431977,
'Peter Juel Jensen': 0.28681299420160633,
'Lise Bech': 0.28651974708918143,
'Henrik Sass Larsen': 0.2853756368054303,
'Villum Christensen': 0.2848810852145714,
'Pia Olsen Dyhr': 0.28466825498579473,
'Jacob Mark': 0.2842788612247485,

'May-Britt Kattrup': 0.28361991035797457,
'Marcus Knuth': 0.2826974310901635,
'Mogens Jensen': 0.28210326719970763,
'Ulla Sandbæk': 0.2816145625275682,
'Peter Hummelgaard Thomsen': 0.2802941487690224,
'Lennart Damsbo-Andersen': 0.27939969180569796,
'Dennis Flydtkjær': 0.279345000390897,
'Stine Brix': 0.279265246740393,
'Søren Gade': 0.2780350300173668,
'Malte Larsen': 0.27750792138418695,
'Simon Emil Ammitzbøll': 0.2744259825027,
'Jeppe Jakobsen': 0.27367570485061404,
'Martin Henriksen': 0.2721164465884162,
'Bent Bøgsted': 0.2708850054034091,
'Lotte Rod': 0.2685228279613619,
'Jonas Dahl': 0.2650493027571071,
'Anne Paulin': 0.2626616721385484,
'Pernille Schnoor': 0.2613586357203019,
'Joachim B. Olsen': 0.25946599975772094,
'Ole Birk Olesen': 0.2579668343258216,
'Inger Støjberg': 0.2562945678807497,
'Rasmus Horn Langhoff': 0.2546501882119948,
'Jesper Petersen': 0.2501904719654985,
'Maria Reumert Gjerding': 0.24986457263402062,
'Jeppe Bruus': 0.24985853740519728,
'Dan Jørgensen': 0.24980135633018644,
'Karsten Hønge': 0.2488707099077617,
'Finn Sørensen': 0.2465141559779014,
'Zenia Stampe': 0.24641922532507524,
'Christian Rabjerg Madsen': 0.24483713840986018,
'Henrik Dahl': 0.24482783402933456,
'Peter Kofod Poulsen': 0.24200644273667216,
'Rune Lund': 0.24059559862493343,
'Pia Adelsteen': 0.24055882658020042,
'Kenneth Kristensen Berth': 0.24023718701243227,
'Nicolai Wammen': 0.23917697004832264,
'Jan E. Jørgensen': 0.23825687742967355,
'Lisbeth Bech Poulsen': 0.23674906558050351,
'Britt Bager': 0.23474563843682456,
'Henrik Brodersen': 0.23321492597955032,
'Michael Aastrup Jensen': 0.23211188995292945,
'Pelle Dragsted': 0.23009968382944507,
'Søren Søndergaard': 0.2293212853957225,
'Pernille Rosenkrantz-Theil': 0.227211153603363,
'Troels Ravn': 0.22534149578588894,
'Jesper Kiel': 0.2250432541819281,
' ': 0.2234800838574425,

```
 'Liselott Blixt': 0.2226720430936628,
 'Peter Skaarup': 0.2220824551616885,
 'Rasmus Jarlov': 0.21872115135811518,
 'Holger K. Nielsen': 0.21244189656931226,
 'Henning Hyllested': 0.21244114629165067,
 'Emrah Tuncer': 0.21185901498430199,
 'Peder Hvelplund': 0.21171512709348353,
 'Christian Langballe': 0.20728035867039035,
 'Naser Khader': 0.20601580430795932,
 'Søren Espersen': 0.193955371195736,
 'Steen Holm Iversen': 0.19199394884673351,
 'Pernille Skipper': 0.18230205185255888,
 'Sarah Glerup': 0.18178727072458226,
 'Marie Krarup': 0.1807436565574745,
 'Nick Hækkerup': 0.17532284817466368,
 'Nikolaj Villumsen': 0.17156223524002479,
 'Søren Pind': 0.16746488994791695,
 'Karina Due': 0.16421252114868973,
 'Johanne Schmidt-Nielsen': 0.1490136236025863,
 'Trine Bramsen': 0.1455567520406635,
 'Preben Bang Henriksen': 0.14295119405070522,
 'Astrid Krag': 0.13826873554187927,
 'Tilde Bork': 0.1276188038960232,
 'Lars Aslan Rasmussen': 0.10591789163217728,
 'Pause Pause': -0.3338589298854211}
```

Total sentiment of persons in the danish parliment

[16]:
```
sorted(person_sent.items(), key=lambda item: item[1], reverse=True)[:5]
```

[16]:
```
{'Pia Kjærsgaard': 2446.505552960304,
 'Finn Sørensen': 1883.8315982844051,
 'Lisbeth Bech Poulsen': 1771.844211748423,
 'Christian Juhl': 1741.7137698197866,
 'Martin Lidegaard': 1637.8114593187854,
 'Mette Bock': 1525.218986225997,
 'Rasmus Nordqvist': 1507.2940070661386,
 'Josephine Fock': 1475.3543878511925,
 'Kristian Pihl Lorentzen': 1452.4253266402154,
 'Pelle Dragsted': 1345.1144307333318,
 'Kristian Jensen': 1296.0930402161364,
 'René Christensen': 1147.2368131851847,
 'Karsten Lauritzen': 1107.6136026904774,
 'Aaja Chemnitz Larsen': 1081.779809523811,
 'René Gade': 1081.4472940793648,
 'Benny Engelbrecht': 1012.9513675724913,
 'Henning Hyllested': 992.4995425370371,
 'Søren Egge Rasmussen': 965.3125153439166,
```

```
'Jørn Neergaard Larsen': 958.9020920634928,
'Brian Mikkelsen': 945.094861997354,
'Inger Støjberg': 924.5493353359801,
'Maria Reumert Gjerding': 915.4288347592613,
'Karen Ellemann': 911.854452380953,
'Mette Abildgaard': 896.7716439550244,
'Sofie Carsten Nielsen': 864.7164447063491,
'Jacob Mark': 853.4364120714298,
'Rune Lund': 839.8975811957664,
'Lars Christian Lilleholt': 836.6596915343904,
'Christian Poll': 824.6077788359762,
'Simon Emil Ammitzbøll': 813.9996050396837,
'Jacob Jensen': 785.1026531111107,
'Sophie Løhde': 782.4300343915346,
'Rasmus Jarlov': 773.9950999455028,
'Martin Henriksen': 767.8962852857153,
'Carolina Magdalene Maier': 761.4346534285705,
'Kenneth Kristensen Berth': 746.3809044695752,
'Rasmus Prehn': 745.3100410492058,
'Pernille Skipper': 733.8186263015867,
'Erling Bonnesen': 728.3800192354528,
'Claus Hjort Frederiksen': 727.6611873015872,
'Joachim B. Olsen': 721.5515933862438,
'Esben Lunde Larsen': 716.9013927671948,
'Søren Søndergaard': 716.6037915202393,
'Carsten Bach': 699.2572931216928,
'Lars Løkke Rasmussen': 697.0593399841283,
'Hans Andersen': 696.8405898730173,
'Jan E. Jørgensen': 694.8309142169313,
'Hans Christian Schmidt': 684.9660444444436,
'Henrik Dam Kristensen': 681.0235742989469,
'Søren Pind': 664.2327394894178,
'Aleqa Hammond': 656.8498312169315,
'Holger K. Nielsen': 655.8272544801582,
'Eva Flyvholm': 654.8289111111119,
'Dennis Flydtkjær': 614.8048244603174,
'Leif Lahn Jensen': 598.9726542962952,
'Jonas Dahl': 595.8334830910043,
'Marcus Knuth': 590.3118137566139,
'Troels Lund Poulsen': 589.3919841269836,
'Torsten Gejl': 578.2209976375658,
'Peter Hummelgaard Thomsen': 573.5575078015876,
'Bent Bøgsted': 567.1573535132258,
'Ole Birk Olesen': 562.0581386291002,
'Trine Torp': 552.069612142857,
'Jens Joel': 552.0277153439151,
'Karsten Hønge': 546.5822966349216,
```

'Pia Olsen Dyhr': 512.1551975925928,
'Søren Pape Poulsen': 497.2072165767193,
'Ida Auken': 482.77461140211545,
'Jane Heitmann': 480.43420105820036,
'Laura Lindahl': 477.20786766402153,
'Nikolaj Amstrup': 474.05932539682505,
'Thomas Danielsen': 468.9482142857137,
'Mattias Tesfaye': 465.480817042989,
'Karina Adsbøl': 454.93768493915286,
'Johanne Schmidt-Nielsen': 451.5038288346563,
'Karen J. Klint': 445.52269591005233,
'Kim Christiansen': 443.3480628835977,
'Marlene Harpsøe': 436.089237080688,
'Dan Jørgensen': 431.7516682539676,
'Torsten Schack Pedersen': 424.56865911904714,
'Hans Kristian Skibby': 421.5931452486776,
'Villum Christensen': 419.6811171164023,
'Kristian Thulesen Dahl': 414.04195925925865,
'Stine Brix': 409.24087787830666,
'Merete Riisager': 399.66127174603173,
'Jesper Petersen': 399.1663884973545,
'Simon Kollerup': 396.11733756613694,
'Christina Egelund': 393.68878306878304,
'Peter Kofod Poulsen': 393.02088306878295,
'Mette Reissmann': 389.4441002949732,
'Nikolaj Villumsen': 388.1612728529085,
'Marianne Jelved': 377.4957763280422,
'Ellen Trane Nørby': 373.0471190476189,
'Jeppe Bruus': 367.342021693121,
'Magni Arge': 362.2343121693122,
'Eva Kjer Hansen': 360.58977799206355,
'Ulla Sandbæk': 355.7073539285714,
'Christian Langballe': 355.3946117619045,
'Helle Thorning-Schmidt': 353.96522415079386,
'Morten Østergaard': 352.292588888889,
'Louise Schack Elholm': 347.84840857142893,
'Flemming Møller Mortensen': 344.9801190476187,
'Ulla Tørnæs': 342.2964206349208,
'Andreas Steenberg': 338.3614743306881,
'Anni Matthiesen': 337.7573201058201,
'Jakob Sølvhøj': 337.08719576719534,
'Leif Mikkelsen': 331.1868968253968,
'Alex Ahrendtsen': 327.29735767195695,
'Pia Adelsteen': 326.15446825396737,
'Jakob Ellemann-Jensen': 322.46726810026365,
'Morten Løkkegaard': 316.1860582010579,
'Magnus Heunicke': 315.6955767195766,

'Mikkel Dencker': 314.4916957671957,
'Jeppe Jakobsen': 306.1911153439155,
'Zenia Stampe': 303.4652759878302,
'Lea Wermelin': 299.35437301587257,
'Erik Christensen': 292.1935296296295,
'Trine Bramsen': 287.79772126984074,
'Jakob Engel-Schmidt': 283.93728359788327,
'Michael Aastrup Jensen': 283.50378350740755,
'Sjúrður Skaale': 275.7048597883596,
'Mette Frederiksen': 275.31055158994707,
'Naser Khader': 270.35041967724885,
'Peder Hvelplund': 263.0518111111114,
'Lotte Rod': 252.15904682539653,
'Karin Nødgaard': 230.24276455026416,
'Peter Juel Jensen': 225.33176076455,
'Bertel Haarder': 225.19033068783074,
'Søren Espersen': 223.248450907428,
'Jens Henrik Thulesen Dahl': 222.99030423280442,
'May-Britt Kattrup': 217.4230232804233,
'Peter Christensen': 213.10624514814796,
'Rasmus Horn Langhoff': 210.262113904762,
'Orla Hav': 206.65898148148136,
'Johannes Lebech': 191.06306746031726,
'Pernille Rosenkrantz-Theil': 179.510444015873,
'Henrik Dahl': 178.17100793650792,
'Kaare Dybvad': 175.99841269841264,
'Bjarne Laustsen': 175.67103650793644,
'Karin Gaardsted': 171.01475861640205,
'Pernille Bendixen': 170.4261587301587,
'Mette Hjermind Dencker': 168.14217989418003,
'Christian Rabjerg Madsen': 162.3049874232804,
'Claus Kvist Hansen': 160.9959656084657,
'Kasper Roug': 157.87144179894182,
'Britt Bager': 157.10117106746046,
'Liselott Blixt': 155.85261640211647,
'Mogens Jensen': 151.52330687830695,
'Susanne Eilersen': 138.88915767195783,
'Lennart Damsbo-Andersen': 136.2157317460319,
'Anders Johansson': 135.79622751322756,
'Mai Mercado': 134.83783068783086,
'Preben Bang Henriksen': 125.42394814814827,
'Marie Krarup': 124.1058243386243,
'Emrah Tuncer': 115.41019841269852,
'Yildiz Akdogan': 109.65145026455026,
'Jesper Kiel': 108.6643857142858,
'Julie Skovsby': 104.99571428571433,
'Carl Holst': 103.95293650793651,

'Nick Hækkerup': 101.31556750317463,
'Pernille Schnoor': 100.33296666666668,
'Jan Johansen': 100.05843915343924,
'Annette Lind': 89.02599206349217,
'Ib Poulsen': 81.73535714285711,
'Peter Skaarup': 69.51180846560851,
'Morten Bødskov': 67.85578042328044,
'Roger Matthisen': 67.75931216931217,
'Lise Bech': 67.32354497354497,
'Søren Gade': 58.55695767195762,
'Malte Larsen': 57.280410052910035,
'Henrik Brodersen': 55.45617724867727,
'Merete Dea Larsen': 54.66232275132275,
'Steen Holm Iversen': 54.12309417989418,
'Rasmus Helveg Petersen': 52.65625661375659,
'Maja Panduro': 52.4854100529101,
'Mette Gjerskov': 49.82675676984128,
'Nicolai Wammen': 49.21305335714287,
'Troels Ravn': 48.41011353968252,
'Jan Erik Messmann': 47.25515873015871,
'Henrik Sass Larsen': 46.70457671957672,
'Jan Rytkjær Callesen': 42.95738730158732,
'Anders Samuelsen': 41.67362219576721,
'Per Husted': 41.26330158730159,
'Kirsten Brosbøl': 40.19375661375661,
'Karina Due': 38.25330680158727,
'Anne Paulin': 38.109582010581995,
'Morten Marinus': 37.08328042328042,
'Daniel Toft Jakobsen': 37.07246031746031,
'Malou Lunderød': 29.14208994708993,
'Dorthe Ullemose': 27.36626984126985,
'Sarah Glerup': 27.113571428571444,
'Ane Halsboe-Jørgensen': 18.9522,
'Rasmus Vestergaard Madsen': 18.175634920634923,
'Uffe Elbæk': 16.373333333333328,
' ': 14.805555555555566,
'Christine Antorini': 13.729576719576714,
'Hans Christian Thoning': 12.638412698412711,
'Thomas Jensen': 9.326111111111109,
'Carsten Kudsk': 7.181455026455027,
'Tilde Bork': 6.095074074074067,
'Lars Aslan Rasmussen': 5.301190476190473,
'Sisse Marie Welling': 4.908571428571432,
'Astrid Krag': 3.0170238095238053,
'Pause Pause': -10.586666666666703}

1.7.3 LIX analysis

LIX is a readibility score commonly used in Scandinavia. In Danish schools this is introduced at a very early stage to determine pupils reading level. We decided to measure the speeches for each party and parliment member using LIX to try to give some insight into how complicated language they are using.

The formula for LIX is as follows

$$LIX = \frac{A}{B} + \frac{C \cdot 100}{A}$$

$$A : Number\ of\ words\ in\ text \tag{1}$$
$$B : Number\ of\ periods\ (all\ ending\ characters\ are\ converted\ to\ periods) \tag{2}$$
$$C : Number\ of\ long\ words\ (more\ than\ 6\ letters) \tag{3}$$

Function for LIX is defined as a function

[17]:
```
def LIX(text: str, longWordBoundary = 7):
    # replaces whitespace with space
    text = re.sub(r'\s\s+', ' ', text)
    # replaces ending characters with period ": ? ! ;"
    text = re.sub('(\:|\;|\?|\!)\w*', '.', text)
    # removes all special characters except for periods
    text = re.sub('[^0-9a-zA-Z.æøåÆØÅé ]+', '', text)
    # extracts sentences
    sentences = text.split('.')
    # removes periods
    text = re.sub('\.','',text)
    # extracts words
    words = text.split(' ')
    long_words = [w for w in words if len(w) >= longWordBoundary]

    lix = len(words) / len(sentences) + (len(long_words) * 100 / len(words))

    return lix
```

Find LIX for each party:

[18]:
```
party_LIX = dict([(party, LIX(doc)) for (party, doc) in party_docs.items()])
```

[19]:
```
dict(sorted(party_LIX.items(), key=lambda item: item[1], reverse=True))
```

[19]:
```
{'UFG': 50.84985758849858,
 'T': 46.097804939769,
 'SIU': 45.63326029413298,
 'IA': 43.00548172488698,
 'RV': 40.520201094212894,
 'ALT': 40.25360764209356,
```

```
 'S': 39.69870616775746,
 'LA': 39.577172121364484,
 'V': 39.45486847110512,
 'EL': 38.87013834606552,
 'JF': 38.51028457149269,
 'SF': 38.25531931820791,
 'KF': 37.824341857440004,
 'DF': 37.484111161511635}
```

Find LIX for each parliment member

[20]:
```
person_LIX = dict([(person, LIX(doc)) for (person, doc) in person_docs.items()])
```

[21]:
```
sorted(person_LIX.items(), key=lambda item: item[1], reverse=True)[:5]
```

[21]:
```
{'Hans Christian Thoning': 55.225820962663065,
 'Carsten Kudsk': 53.49299719887955,
 'Kirsten Brosbøl': 50.04360183841315,
 'Steen Holm Iversen': 49.19016453079425,
 'Jakob Sølvhøj': 48.378575575413365,
 'Carsten Bach': 48.36037226055852,
 'Daniel Toft Jakobsen': 47.090495449949444,
 'Tilde Bork': 47.0279219616968,
 'Karen Ellemann': 46.90617917118062,
 'Ellen Trane Nørby': 46.683879723002946,
 'Orla Hav': 46.20326760995481,
 'Magni Arge': 46.097804939769,
 'Aleqa Hammond': 45.9966523158828,
 'Dorthe Ullemose': 45.49625332366449,
 'Ulla Tørnæs': 45.44485774835078,
 'Emrah Tuncer': 45.22205728343836,
 'Torsten Schack Pedersen': 45.18258099801241,
 'Jørn Neergaard Larsen': 44.94186332669097,
 'Christina Egelund': 44.837193635749315,
 'Erik Christensen': 44.57682345771351,
 'Lars Christian Lilleholt': 44.4533694534525,
 'Britt Bager': 44.44452788893113,
 'Nikolaj Amstrup': 44.4052398926537,
 'Esben Lunde Larsen': 44.34370912691912,
 'Trine Bramsen': 44.11760930471553,
 'Søren Pind': 43.99443997132232,
 'Uffe Elbæk': 43.98841354723707,
 'Christian Rabjerg Madsen': 43.70426640650652,
 'Laura Lindahl': 43.69780438724074,
 'Sophie Løhde': 43.69682690258361,
 'Mogens Jensen': 43.61752149928378,
 ' ': 43.55348361088246,
 'Peter Christensen': 43.46409579955379,
```

'Jane Heitmann': 43.299651542094544,
'Flemming Møller Mortensen': 43.28433277621967,
'Troels Ravn': 43.15193996035117,
'Morten Bødskov': 43.14507055459191,
'Peter Juel Jensen': 43.119905958874185,
'Malte Larsen': 43.09590792838875,
'Troels Lund Poulsen': 43.05824580422566,
'Johannes Lebech': 43.01471573274429,
'Aaja Chemnitz Larsen': 43.00548172488698,
'Simon Kollerup': 42.998056571506794,
'Carl Holst': 42.900013155705174,
'Anders Johansson': 42.738344984151766,
'Eva Kjer Hansen': 42.73567148822853,
'Roger Matthisen': 42.73169049951028,
'Nikolaj Villumsen': 42.684439684456976,
'Jan Rytkjær Callesen': 42.64483388292025,
'Mikkel Dencker': 42.520703118520274,
'Trine Torp': 42.45572440325368,
'Bertel Haarder': 42.41976682508574,
'Martin Lidegaard': 42.195956238748096,
'Claus Kvist Hansen': 42.19328607172644,
'Henrik Dahl': 42.13548506231321,
'Mette Reissmann': 42.13390223041799,
'Nick Hækkerup': 42.064462523528626,
'Mattias Tesfaye': 42.00016365304888,
'Torsten Gejl': 41.993553946924244,
'Merete Dea Larsen': 41.98309172124634,
'Julie Skovsby': 41.96911594159377,
'Martin Henriksen': 41.946991897767056,
'Preben Bang Henriksen': 41.908195134592006,
'Jeppe Jakobsen': 41.72346907801974,
'Anne Paulin': 41.690508958381116,
'Kristian Jensen': 41.66741271515346,
'Josephine Fock': 41.647976269805866,
'Thomas Danielsen': 41.63639481690515,
'May-Britt Kattrup': 41.59323930346281,
'Peter Hummelgaard Thomsen': 41.56815937025754,
'Karina Adsbøl': 41.49180853461064,
'Morten Østergaard': 41.27816874672805,
'Peder Hvelplund': 41.1583128033811,
'Zenia Stampe': 41.0919996653986,
'Yildiz Akdogan': 40.97067602856328,
'Karsten Lauritzen': 40.96563685026232,
'Marcus Knuth': 40.94907191284929,
'Rasmus Nordqvist': 40.899948114097214,
'Peter Kofod Poulsen': 40.85289190058888,
'Jeppe Bruus': 40.740748518273065,

'Jonas Dahl': 40.61615006690815,
'Astrid Krag': 40.58923884514436,
'Pernille Skipper': 40.58248442202358,
'Jan Johansen': 40.575667974840066,
'Inger Støjberg': 40.52095726545788,
'Henning Hyllested': 40.519293237698236,
'Anni Matthiesen': 40.51490173962084,
'Maria Reumert Gjerding': 40.43779320176495,
'Kenneth Kristensen Berth': 40.40509965740142,
'Karin Gaardsted': 40.3936918359538,
'Mette Frederiksen': 40.3346174051821,
'Jesper Petersen': 40.22198070256799,
'Kaare Dybvad': 40.138181978270936,
'Mai Mercado': 40.11067613033537,
'Sofie Carsten Nielsen': 40.01438709561431,
'Hans Christian Schmidt': 39.990475422811734,
'Jakob Engel-Schmidt': 39.93173996175908,
'Henrik Brodersen': 39.92122304276645,
'Morten Marinus': 39.819182389937104,
'Jan Erik Messmann': 39.80343861682604,
'Jens Joel': 39.78597856196555,
'Rune Lund': 39.770140262945574,
'Eva Flyvholm': 39.623468842004385,
'Rasmus Prehn': 39.57993071843709,
'Pernille Schnoor': 39.562164804224246,
'Merete Riisager': 39.555896065089826,
'Susanne Eilersen': 39.448951362903685,
'Simon Emil Ammitzbøll': 39.44268275788703,
'Marianne Jelved': 39.428768658020374,
'Karsten Hønge': 39.3552956374151,
'Claus Hjort Frederiksen': 39.348862315657065,
'Carolina Magdalene Maier': 39.3357157151691,
'Peter Skaarup': 39.31424451539809,
'Søren Egge Rasmussen': 39.29100053402858,
'Henrik Sass Larsen': 39.29071459157074,
'Karin Nødgaard': 39.19562823765965,
'Leif Mikkelsen': 39.17857588317694,
'Kasper Roug': 39.093201696461705,
'René Gade': 39.08810861882246,
'Rasmus Jarlov': 39.0785138813422,
'Jens Henrik Thulesen Dahl': 39.051300960593,
'Michael Aastrup Jensen': 39.02406201867589,
'Hans Andersen': 38.94569165070243,
'Rasmus Helveg Petersen': 38.9173376734624,
'Sarah Glerup': 38.90822318148913,
'Hans Kristian Skibby': 38.75728931283553,
'Lotte Rod': 38.65142776601094,

'Dennis Flydtkjær': 38.607115239422654,
'Mette Abildgaard': 38.58316431992896,
'Maja Panduro': 38.56058391620559,
'Ulla Sandbæk': 38.537851186666074,
'Rasmus Horn Langhoff': 38.53762545057509,
'Pelle Dragsted': 38.51292535784826,
'Sjúrður Skaale': 38.51028457149269,
'Brian Mikkelsen': 38.467484323562175,
'Lea Wermelin': 38.396671845346106,
'Finn Sørensen': 38.36272578650629,
'Louise Schack Elholm': 38.31519513677371,
'Andreas Steenberg': 38.29813770058873,
'Pernille Rosenkrantz-Theil': 38.26328106575981,
'Ida Auken': 38.23874343244056,
'Marie Krarup': 38.16713895854773,
'Christian Poll': 38.15566474858619,
'Søren Søndergaard': 38.14848576821101,
'Lisbeth Bech Poulsen': 38.10500339762278,
'Dan Jørgensen': 37.9467760921699,
'Morten Løkkegaard': 37.894089441667255,
'Jan E. Jørgensen': 37.88173474515539,
'Søren Gade': 37.86371856388494,
'Lennart Damsbo-Andersen': 37.68460557101692,
'Marlene Harpsøe': 37.669998611374425,
'Pia Olsen Dyhr': 37.561568880000905,
'Magnus Heunicke': 37.476774062179956,
'Lise Bech': 37.425548976067105,
'Lars Løkke Rasmussen': 37.4168027630125,
'Johanne Schmidt-Nielsen': 37.2855633328268,
'Nicolai Wammen': 37.22942409532215,
'Per Husted': 37.022544482155716,
'Naser Khader': 36.868648176013366,
'Villum Christensen': 36.78770239133822,
'Thomas Jensen': 36.75079264426125,
'Anders Samuelsen': 36.74976853842767,
'Holger K. Nielsen': 36.744574974852114,
'Rasmus Vestergaard Madsen': 36.73990043391974,
'Bjarne Laustsen': 36.649022731836595,
'Jacob Mark': 36.50489646210693,
'Mette Gjerskov': 36.439887312265625,
'Pernille Bendixen': 36.27522022131937,
'Kristian Thulesen Dahl': 36.12023599100385,
'Christian Langballe': 36.05271562803452,
'Jacob Jensen': 35.894143717678304,
'Malou Lunderød': 35.84532629655942,
'Joachim B. Olsen': 35.77575537141689,
'Karina Due': 35.60948676572127,

```
'Ib Poulsen': 35.4779539292083,
'Liselott Blixt': 35.44723011469402,
'Lars Aslan Rasmussen': 35.4106529209622,
'Jesper Kiel': 35.11106602422585,
'Jakob Ellemann-Jensen': 35.099657196275245,
'Ole Birk Olesen': 34.98710710500459,
'Mette Hjermind Dencker': 34.84962128500021,
'Sisse Marie Welling': 34.80487804878049,
'Annette Lind': 34.792645085091536,
'Ane Halsboe-Jørgensen': 34.549516539440205,
'Kristian Pihl Lorentzen': 34.41602114560915,
'Christine Antorini': 34.3416149068323,
'Søren Espersen': 34.239488550592355,
'Pia Adelsteen': 34.100367267395946,
'Alex Ahrendtsen': 34.053263263882144,
'Kim Christiansen': 33.70935249370439,
'Søren Pape Poulsen': 33.53471553266289,
'Mette Bock': 33.514117847304085,
'Pia Kjærsgaard': 33.440634140611,
'Karen J. Klint': 33.07011764907721,
'Henrik Dam Kristensen': 32.799046007987855,
'Benny Engelbrecht': 32.75787295352336,
'René Christensen': 32.38701757337345,
'Stine Brix': 31.98426853958378,
'Leif Lahn Jensen': 31.204880101960804,
'Erling Bonnesen': 31.154387989439577,
'Christian Juhl': 30.999894777709077,
'Helle Thorning-Schmidt': 30.340514429006326,
'Bent Bøgsted': 29.37088538634748,
'Pause Pause': 22.969136670416198}
```

1.7.4 Language analysis using TF-IDF

In Danish the word "politikersnak" meaning "politician speak" is a real word (https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjJv
til-politikersnak-udenomssnak-floskler-og-papegoejesaetninger&usg=AOvVaw0PyBFX3PVHqIF-
ftjcJPLK). It is used to refer to when a person is avoiding answering a question or is directing the conversation onto a topic that they would like to brand themself on.

To cut through all of this "politikersnak" that are in these thousands of speeches made from the podium in Folketinget, we would like to determine which words are actually special about a certain politician. They are all using a lot of the same words, but if you for instance were interested in green energy - you would probably be interested in which politicians were actually talking about "solceller" (solar power) and "vindmøller" (wind mills). Or if you were very interested in your specific little city and there was a single politician who actually mentioned it multiple times, it would be interesting for you to know that they were actually fighting your cause from the most important podium in Denmark. The same idea can be applied to the political parties. The TF-IDF

will hopefully also help us understand what the communities created by the Louvain Partitioning have in common. By looking at their speeches, we will be able to see words that are important to that group.

For the purpose of finding words that are close to unique but still prevalent for some kind of document, the TF-IDF method is ideal. In our GitHub repo we have provided a file `tf_idf.py`, that defines the exact flavour of TF-IDF we are using.

[22]:
```
from tf_idf import TfIdf
from nltk.corpus import stopwords
```

[23]:
```
def SpeechTFIDF(doc, column1: str):
    doc_counted = TfIdf(doc, stop_words=stopwords.words('danish') + ["hr",␣
 ↪"fru", "fordi", "hen", "får"])
    doc_counted.docs_fd.keys()
    TF_IDF = pd.DataFrame()

    for p in doc_counted.docs_fd.keys():
        tfids = doc_counted.all_tf_idf(p)
        tfids_sorted = sorted(tfids, key=tfids.get, reverse=True)[:100] # Top 5

        #print(f"{p}", tfids)
        TF_IDF = TF_IDF.append([[f"{p}", tfids]],ignore_index=True)
    TF_IDF = TF_IDF.rename(columns={0: str(column1), 1:"Words"})
    return(TF_IDF)
```

TF_IDF is analysed for each party

[24]:
```
Party_TF_IDF_ALL = SpeechTFIDF(party_docs, "Party")
```

[25]:
```
Party_TF_IDF_ALL
```

[25]:

	Party	Words
0	ALT	{'tak': -0.0001620893789640562, 'hørte': 1.095…
1	DF	{'tak': -0.00013362092276703437, 'talen': 8.27…
2	EL	{'tak': -0.00013045472639642896, 'socialdemokr…
3	IA	{'arktis': -0.00010335540258316482, 'hot': 2.0…
4	JF	{'fornemt': 0.0002723333224501143, 'store': -6…
5	KF	{'sidder': 0.0, 'millioner': 1.904683111902463…
6	LA	{'retorik': 5.989939538166984e-06, 'benny': 1…
7	RV	{'tak': -9.589612384965261e-05, 'ordførerens':…
8	S	{'tak': -0.0001290966010152699, 'formand': -1…
9	SF	{'tak': -8.222107688316156e-05, 'gerne': -0.00…
10	SIU	{'tak': -3.066574778115699e-05, 'rigsfællesska…
11	T	{'godt': -6.349723228827407e-05, 'stå': 0.0, '…
12	UFG	{'tak': -3.488151731949151e-05, 'formand': -3…
13	V	{'tak': -0.00013404959303226743, 'formand': -1…

TF_IDF is analysed for each member

[26]:
```
Person_TF_IDF_ALL = SpeechTFIDF(person_docs, "Person")
```

[27]:
```
Person_TF_IDF_ALL.sample(5)
```

[27]:

	Person	Words
0		{'punktet': 0.0061550261181423475, 'udgået': 0…
1	Aaja Chemnitz Larsen	{'arktis': 0.003086036992207079, 'hot': 8.3852…
2	Aleqa Hammond	{'tak': 8.891363930646234e-06, 'formand': 6.06…
3	Alex Ahrendtsen	{'lovforslag': 0.00017100373687430138, 'måske'…
4	Anders Johansson	{'tak': 6.979399987128776e-05, 'beslutningsfor…
..	…	…
199	Ulla Sandbæk	{'tak': 6.192193439774338e-05, 'undskyld': 3.4…
200	Ulla Tørnæs	{'gerne': 7.600299990855947e-05, 'takke': 8.91…
201	Villum Christensen	{'tak': 3.285573726855341e-05, 'beslutningsfor…
202	Yildiz Akdogan	{'tak': 8.440406916708445e-05, 'ordføreren': 0…
203	Zenia Stampe	{'tak': 3.0070530636075416e-05, 'svært': 0.000…

[204 rows x 2 columns]

1.7.5 Member information

Connecting name and id for each Person in data

[39]:
```
actors = pd.read_csv("data/ft/Aktør.csv", index_col = False)
```

[40]:
```
ActorNameDict = dict(zip(actors["navn"], actors["id"]))
```

[41]:
```
import re
def getTag(bio,tag):
    if bio:
        results = re.search('<'+tag+'>(.*)</'+tag+'>', bio)
        if results: return results[1]
    else: return ''
```

[42]:
```
actor_types = pd.read_csv('data/ft/Aktørtype.csv').set_index('id', drop=False)
```

[43]:
```
actors = actors.fillna('')
actors['parti'] = actors['biografi'].apply(getTag, args=('partyShortname',))
```

[44]:
```
IDPartyDict = dict(zip(actors["id"],actors['biografi'].apply(getTag,␣
↪args=('partyShortname',))))
```

1.7.6 Combine Information

Information is combined to have profiles for each party and each member. This is primarily used for showing data on the website more easily.

Party

[45]:
```
#Sentiment
PartyData = pd.DataFrame.from_dict(party_sent, orient='index')
PartyData["avgSent"] = pd.DataFrame.from_dict(party_sent_avg, orient='index')[0]
PartyData = PartyData.rename(columns={0: "totalSent",})

#LIX
PartyData["lix"] = pd.DataFrame.from_dict(party_LIX, orient='index')[0]

#TF-IDF
Party_TF_IDF_Dict =␣
↪dict(zip(Party_TF_IDF_ALL["Party"],Party_TF_IDF_ALL["Words"]))
PartyData["tfIdf"] = PartyData.apply(lambda row: Party_TF_IDF_Dict[row.name],␣
↪axis=1)

#Votes
PartyData["votes"] = pd.DataFrame.from_dict(PartyVotesDict, orient='index')[0]
PartyData["mandates"] = pd.DataFrame.from_dict(PartyMandatesDict,␣
↪orient='index')[0]
```

[46]:
```
PartyData
```

[46]:

	totalSent	avgSent	lix	tfIdf	votes	mandates
ALT	7184.792473	0.342537	40.253608	{'tak': -0.0001620893789640562, 'hørte': 1.095…	104278.0	5.0
DF	9816.984189	0.244083	37.484111	{'tak': -0.00013362092276703437, 'talen': 8.27…	308513.0	16.0
EL	11673.655360	0.217732	38.870138	{'tak': -0.00013045472639642896, 'socialdemokr…	245100.0	13.0
IA	1081.779810	0.462341	43.005482	{'arktis': -0.00010335540258316482, 'hot': 2.0…	NaN	NaN
JF	275.704860	0.331436	38.510285	{'fornemt': 0.0002723333224501143, 'store': -6…	NaN	NaN
KF	3637.448697	0.296384	37.824342	{'sidder': 0.0, 'millioner': 1.904683111902463…	233865.0	12.0
LA	5573.353110	0.261870	39.577172	{'retorik': 5.989939538166984e-06, 'benny': 1…	82270.0	4.0
RV	4968.206200	0.313630	40.520201	{'tak': -9.589612384965261e-05, 'ordførerens':…	304714.0	16.0
S	10581.238757	0.267605	39.698706			
SF	5492.370928	0.257456	38.255319			
SIU	594.668178	0.436765	45.633260			
T	362.234312	0.372934	46.097805			
UFG	62.181653	0.606236	50.849858			
V	7562.340035	0.279635	39.454868			

```
S    {'tak': -0.0001290966010152699, 'formand': -1…  914882.0      48.0
SF   {'tak': -8.222107688316156e-05, 'gerne': -0.00…  272304.0      14.0
SIU  {'tak': -3.066574778115699e-05, 'rigsfællesska…       NaN       NaN
T    {'godt': -6.349723228827407e-05, 'stå': 0.0, '…       NaN       NaN
UFG  {'tak': -3.488151731949151e-05, 'formand': -3…    2774.0       0.0
V    {'tak': -0.00013404959303226743, 'formand': -1…  826161.0      43.0
```

Saving Data

[47]:
```
PartyData.to_csv("data/AppData/" + speechString + "/PartyData"+ speechString +".
↪csv")
```

1.7.7 People

Actor name to id dict

[48]:
```
#Sentiment
PersonData = pd.DataFrame.from_dict(person_sent, orient='index')
PersonData["avgSent"] = pd.DataFrame.from_dict(person_sent_avg,␣
↪orient='index')[0]
PersonData = PersonData.rename(columns={0: "totalSent",})

#LIX
PersonData["lix"] = pd.DataFrame.from_dict(person_LIX, orient='index')[0]

#TF-IDF
Person_TF_IDF_Dict =␣
↪dict(zip(Person_TF_IDF_ALL["Person"],Person_TF_IDF_ALL["Words"]))
PersonData["tfIdf"] = PersonData.apply(lambda row: Person_TF_IDF_Dict[row.
↪name], axis=1)
#PersonData["tfIdf"] = pd.DataFrame.from_dict(Person_TF_IDF_Dict,␣
↪orient='index')[0]

#Votes
PersonData["votes"] = pd.DataFrame.from_dict(StemmerDict, orient='index')[0].
↪astype(int)
PersonData["personalVotes"] = pd.DataFrame.from_dict(PStemmerDict,␣
↪orient='index')[0]

#Convert index to id
PersonData["id"] = pd.DataFrame.from_dict(ActorNameDict, orient='index')[0]
PersonData["name"] = PersonData.index
PersonData = PersonData.set_index("id")

#Party
PersonData["party"] = pd.DataFrame.from_dict(IDPartyDict, orient='index')[0]

#Rearrange columns
```

```
PersonData = PersonData[["name","party", "votes", "personalVotes", "totalSent",␣
↪"avgSent", "lix", "tfIdf"]]
```

Saving data

[50]:
```
PersonData.sort_values(by="votes", ascending=False)
```

[50]:

id	name	party	votes	personalVotes	totalSent	avgSent	lix	tfIdf
16374.0	Anders Johansson	KF	935.0	695.000	135.796228	0.489409	42.738345	{'tak': 6.979399987128776e-05, 'beslutningsfor…
16074.0	Steen Holm Iversen	LA	926.0	451.000	54.123094	0.191994	49.190165	{'tak': 6.648609365738912e-05, 'ganske': 0.000…
77.0	Kristian Thulesen Dahl	DF	90.0	57.371	414.041959	0.317495	36.120236	{'tak': 3.5780803764293e-05, 'socialdemokratis…
144.0	Helle Thorning-Schmidt	S	68.0	42.412	353.965224	0.361447	30.340514	{'ordføreren': 0.004790339002940531, 'rasmus':…
252.0	Pia Kjærsgaard	DF	50.0	26.583	2446.505553	0.486915	33.440634	{'udvalget': 0.000267922011908464, 'valgs': 0…
…	…	…	…	…	…	…	…	…
15795.0	May-Britt Kattrup	LA	NaN	NaN	217.423023	0.283620	41.593239	{'tak': 6.279490492828702e-05, 'periode': 4.99…
NaN	Pause Pause	NaN	NaN	NaN	-10.586667	-0.333859	22.969137	{'mødet': 0.1337599248496442, 'udsat': 0.04283…
262.0	Sjúrður Skaale	JF	NaN	NaN	275.704860	0.331436	38.510285	{'fornemt': 0.0011334213418015505, 'store': 0…
15784.0	Søren Pape Poulsen	KF	NaN	NaN	497.207217	0.316898	33.534716	{'tak': 3.279578684057815e-05, 'rigtig': 0.000…
296.0	Ulla Tørnæs	V	NaN	NaN	342.296421	0.442169	45.444858	{'gerne': 7.600299990855947e-05, 'takke': 8.91…

```
[204 rows x 8 columns]
```

[51]:
```
PersonData.to_csv("data/AppData/" + speechString + "/PersonData"+ speechString␣
↪+".csv")
```

1.7.8 Export to json

[52]:
```
import json
```

Parties

[53]:
```
PartyData.apply(lambda row: PartyData.loc[str(row.name)].to_json('data/AppData/
↪'+ speechString + '/party/ ' + row.name + '.json',␣
↪orient="table",force_ascii=False), axis=1)
```

[53]:
```
ALT    None
DF     None
EL     None
IA     None
JF     None
KF     None
LA     None
RV     None
S      None
SF     None
SIU    None
T      None
UFG    None
V      None
dtype: object
```

Persons

[54]:
```
PersonDataNoNaN = PersonData[~PersonData.index.duplicated(keep='first')].
↪dropna(how='any', thresh=6)
```

[55]:
```
PersonDataNoNaN.apply(lambda row: PersonData.loc[int(row.name)].to_json('data/
↪AppData/'+ speechString + '/persons/' + str(int(row.name)) + '.json',␣
↪orient="table",force_ascii=False), axis=1)
```

[55]:
```
id
15757.0    None
15758.0    None
18.0       None
16374.0    None
148.0      None
123.0      None
```

```
286.0      None
16503.0    None
181.0      None
168.0      None
38.0       None
120.0      None
183.0      None
129.0      None
257.0      None
83.0       None
15782.0    None
15765.0    None
15778.0    None
15779.0    None
16072.0    None
39.0       None
79.0       None
15797.0    None
15762.0    None
15773.0    None
17.0       None
270.0      None
15781.0    None
167.0      None
366.0      None
119.0      None
15798.0    None
163.0      None
16428.0    None
15766.0    None
49.0       None
261.0      None
15791.0    None
265.0      None
3975.0     None
224.0      None
164.0      None
125.0      None
179.0      None
82.0       None
144.0      None
44.0       None
3082.0     None
15800.0    None
352.0      None
28.0       None
33.0       None
```

```
4956.0     None
152.0      None
182.0      None
173.0      None
15788.0    None
207.0      None
57.0       None
15785.0    None
102.0      None
15796.0    None
9603.0     None
15764.0    None
200.0      None
113.0      None
213.0      None
214.0      None
15789.0    None
656.0      None
14017.0    None
111.0      None
147.0      None
6737.0     None
205.0      None
15783.0    None
187.0      None
72.0       None
280.0      None
278.0      None
56.0       None
162.0      None
15761.0    None
199.0      None
24.0       None
16180.0    None
11583.0    None
124.0      None
249.0      None
271.0      None
194.0      None
77.0       None
16610.0    None
217.0      None
145.0      None
1504.0     None
15760.0    None
260.0      None
245.0      None
```

```
184.0      None
219.0      None
15774.0    None
50.0       None
112.0      None
93.0       None
15881.0    None
244.0      None
189.0      None
297.0      None
16473.0    None
16351.0    None
15790.0    None
15769.0    None
55.0       None
78.0       None
1146.0     None
70.0       None
141.0      None
15770.0    None
15795.0    None
378.0      None
73.0       None
15793.0    None
135.0      None
138.0      None
197.0      None
263.0      None
74.0       None
178.0      None
216.0      None
4770.0     None
220.0      None
15771.0    None
101.0      None
116.0      None
668.0      None
67.0       None
12.0       None
16176.0    None
68.0       None
99.0       None
246.0      None
16582.0    None
15777.0    None
273.0      None
15767.0    None
```

```
172.0      None
15768.0    None
69.0       None
4466.0     None
180.0      None
48.0       None
221.0      None
252.0      None
43.0       None
127.0      None
59.0       None
266.0      None
1454.0     None
15792.0    None
264.0      None
16073.0    None
7737.0     None
15786.0    None
655.0      None
16603.0    None
117.0      None
208.0      None
16092.0    None
262.0      None
100.0      None
130.0      None
16074.0    None
122.0      None
14372.0    None
667.0      None
34.0       None
1845.0     None
15784.0    None
132.0      None
176.0      None
139.0      None
13439.0    None
15780.0    None
15775.0    None
80.0       None
118.0      None
15794.0    None
206.0      None
84.0       None
225.0      None
15772.0    None
296.0      None
```

```
155.0      None
1619.0     None
191.0      None
dtype: object
```

---

[49]:
```
pd.set_option('display.max_rows', 200)
```

1.8 Discussion

In general we were able to use a lot of the tools we learned during the course. It gave us a way to actully comprehend a very large amount of information in an understandable way. For instance the ability to know what each person was talking about gave us insights into debates that had happened with in parliament like a discussion on facial recognition where a single politician had fought very hard against it. We were afraid that the fact that the dataset was in Danish would pose a major challenge. However `sentida` package was awesome is something that is great to know for natively Danish *computational social scientists in the making*.

On the other hand we probably took on too big a dataset with too wide a scope for the very limited timeframe we were working with. The notebook above is only a fraction of all the notebook that we have created. We created a lot of different networks with a variety of spectacular bad results, and we have waited for long periods on time when analyzing speeches. Some of the things that we didn't have the time to get into but really wanted to was:

- If a party is taken out as a subgraph - could we have found communities within the parties?
- How do all of these calculated statistics correlate with votes for each politician/party? (we actually began looking into this, but didn't have the time to figure out good models)
- Could we have linked the politicians in better ways? Perhaps combining multiple ways of linking people.
- Are there better suited community detection algorithms that we could have used?

These are just some of the questions we still feel like we haven't answered. In general we do however feel like we learned a lot from doing the project - even if we didn't find any major conspiracy within the Danish parliament.