ExposeYourParlament

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]:
```python
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  \
9965  13034      10            NaN
4420  19002       5            NaN
3493  14142       5            NaN

                                                navn  \
9965  DRRB Danske Reklame- og Relationsbureauers Bra…
4420                                 Gitte Willumsen
3493                             Inge Fischer Møller

                                             fornavn  efternavn  \
9965  DRRB Danske Reklame- og Relationsbureauers Bra…        NaN
4420                                           Gitte  Willumsen
3493                                    Inge Fischer     Møller

                                            biografi  periodeid  \
9965                                             NaN        NaN
4420  <member><url>/medlemmer/mf/g/gitte-willumsen</…        NaN
3493                                             NaN        NaN

             opdateringsdato            startdato             slutdato  \
9965  2014-09-22T15:58:01.197                  NaN                  NaN
4420    2021-01-19T11:22:30.9  2020-01-01T00:00:00  2020-06-25T00:00:00
3493  2020-06-19T10:34:14.433  1973-10-02T00:00:00                  NaN

              type
9965  Organisation
4420        Person
3493        Person
```

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

[3]:
```
          id  fraaktørid  tilaktørid            startdato slutdato  \
26850  86852       14897         662  2001-10-02T00:00:00      NaN
14062  51839         310          70  2012-10-03T00:00:00      NaN
```

```
31511  93356       15351       14998  1961-10-03T00:00:00      NaN

                  opdateringsdato  rolleid   rolle
26850   2015-04-29T10:59:02.22        15  medlem
14062  2014-09-22T15:55:51.463        15  medlem
31511   2015-05-19T10:49:11.75        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]:
```python
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]:
```python
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]:
```python
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).

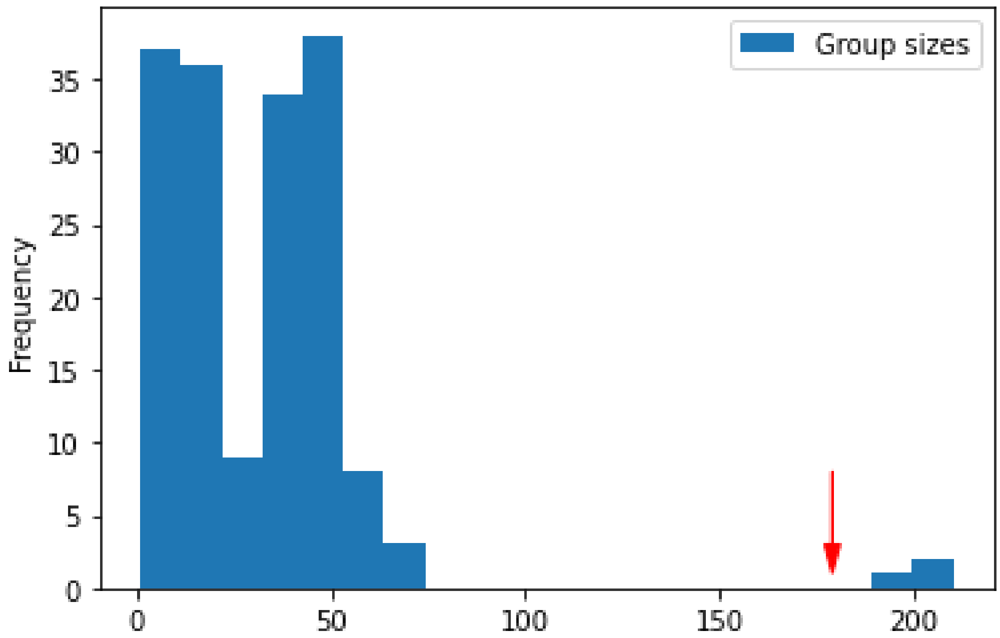
[7]:
```
# It seems to be that the `tilaktørid` always specifies the Actor id of the
↪group
# and the `fraaktørid` always specifies the person when linking a person to a
↪group that they are a member of.
relations = actor_relations[(actor_relations['tilaktørid'].isin(groupIds))
                            & (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
16022       [122, 15778, 15765, 224, 164, 200, 1504, 50, 1…
17660       [17140, 17543, 16176, 16374, 123, 49, 44, 199,…
16095                                           [18, 366]
```

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.

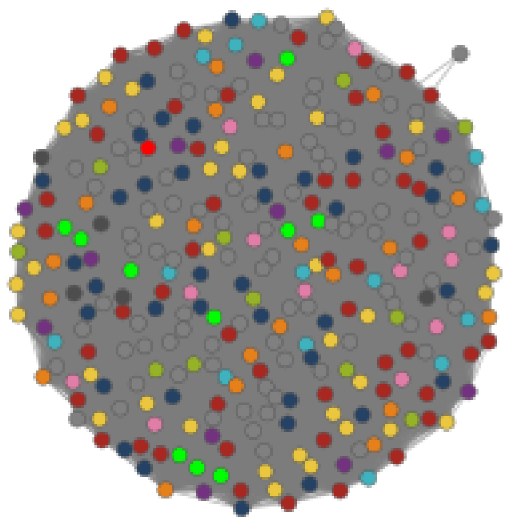
[14]:
```
G_medium = nx.Graph()
G_medium.add_weighted_edges_from(weighted_edges)
```

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')
```

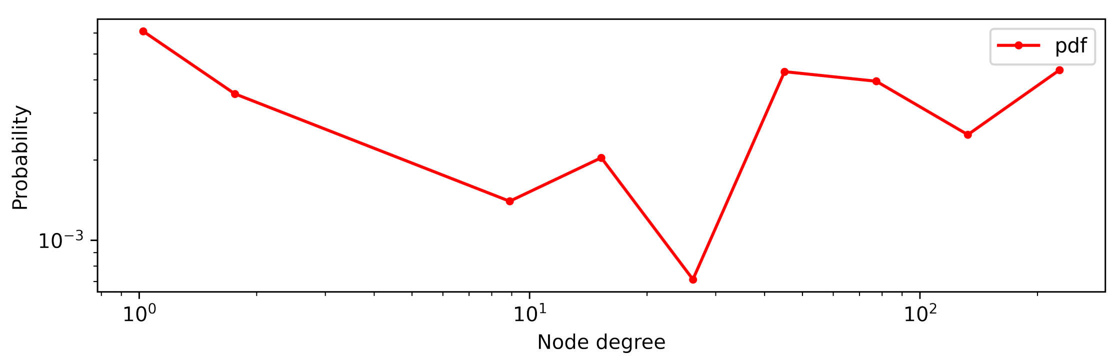
We can then visualize the network.

[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).

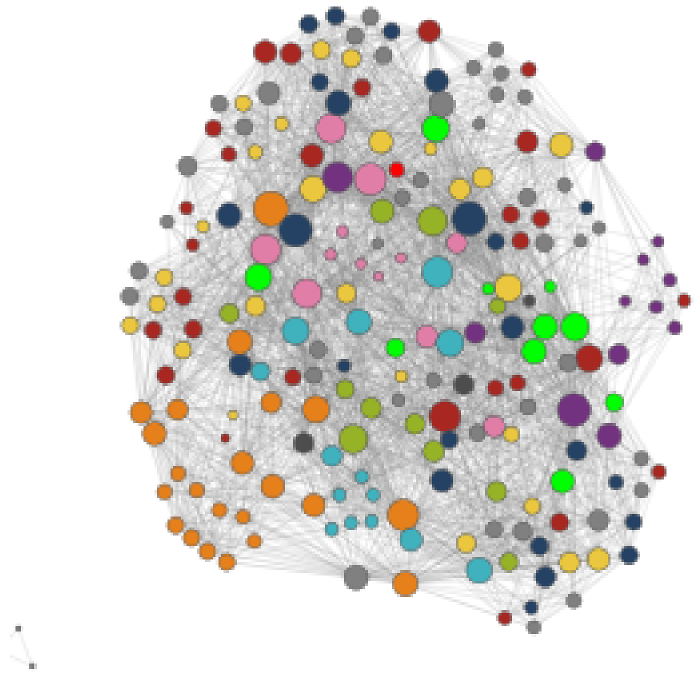
[16]:
```
from custom_plots import degree_distribution_histogram

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



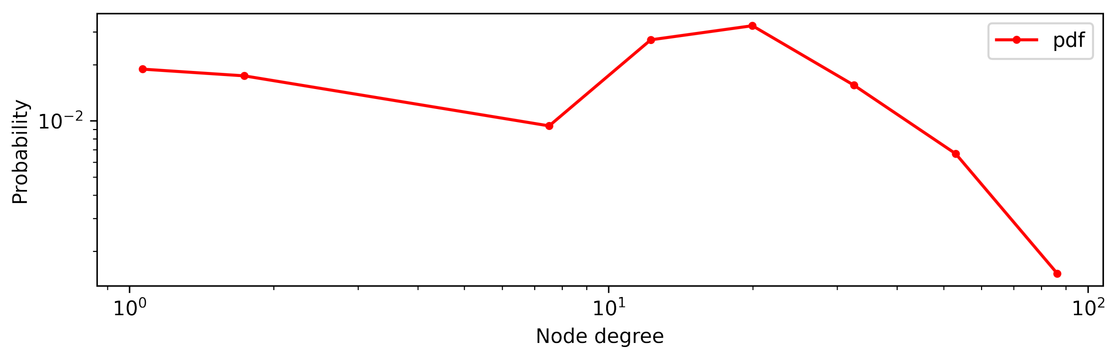
This might be due to several different reasons, but one of them is that a group of 50 people is not certain to know each other. To check if this might be the case, we repeat the plotting proces with all the big groups cutted off.

[17]:
```
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)
```



```
[18]: 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.

[19]:
```
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.

[20]:
```
import re
from bs4 import BeautifulSoup
import os
```

```python
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.

```python
[21]: 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.

[22]:
```
len(unparsed), len(parsed)
```

[22]:
```
(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.

[23]:
```
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.

[24]:
```
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.

[25]:
```
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))
```

[26]:
```
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.

[27]:
```
import networkx as nx
G = nx.Graph()
G.add_weighted_edges_from(links)
```

[28]:
```
len(G.nodes()), len(G.edges)
```

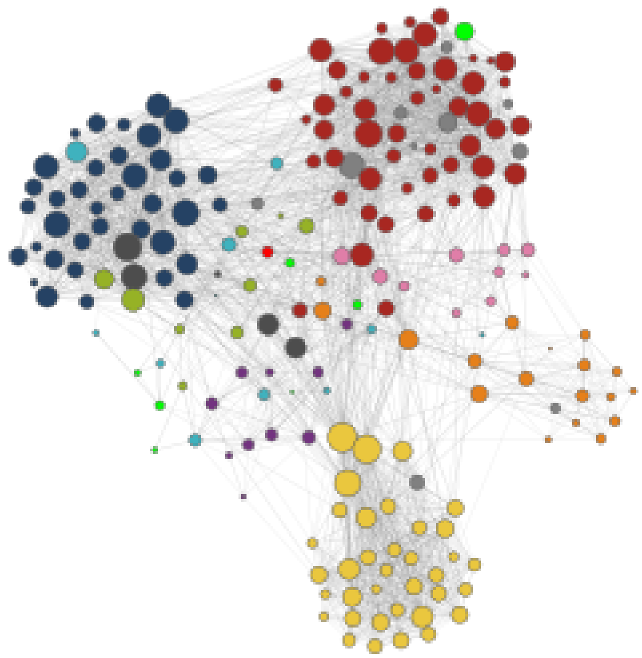
[28]: (207, 3044)

We then add colors based on their party.

[29]:
```
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.

```
[30]: 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).

```
[31]: 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]
```

```
[31]: [('Søren Espersen', 0.01437467892796017),
 ('Kristian Thulesen Dahl', 0.013340298619523301),
 ('Nick Hækkerup', 0.011760569885407557),
```

```
('Peter Skaarup', 0.011639902858985954),
 ('Lars Løkke Rasmussen', 0.011507319706256014),
 ('Mette Frederiksen', 0.010874786971999926),
 ('Lars Christian Lilleholt', 0.010734129042382108),
 ('Mogens Jensen', 0.010489218138530458),
 ('Inger Støjberg', 0.010060846529803151),
 ('Mogens Lykketoft', 0.009948853824119612)]
```

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.

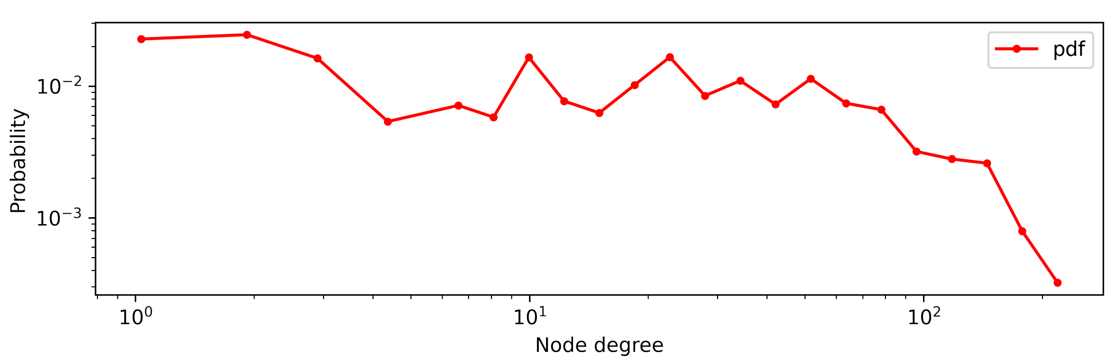
Using Page-Rank wasn't really neccesarry as we get a very similar result by just finding the degree of each node.

[32]:
```
dgs = list(G.degree(weight='weight'))
dgs.sort(key=lambda v: v[1], reverse=True)
dgs[:10]
```

[32]:
```
[('Søren Espersen', 219),
 ('Kristian Thulesen Dahl', 202),
 ('Lars Løkke Rasmussen', 200),
 ('Nick Hækkerup', 186),
 ('Lars Christian Lilleholt', 177),
 ('Peter Skaarup', 177),
 ('Mette Frederiksen', 173),
 ('Kristian Jensen', 172),
 ('Inger Støjberg', 167),
 ('Mogens Jensen', 158)]
```

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

[33]:
```
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

[34]:
```
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.

[35]:
```
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.

[36]:
```
import networkx.algorithms.community.quality as nxquality

nxquality.modularity(G, louvain_partitions)
```

[36]: 0.5522506921469433

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

[37]:
```
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)
```

[37]: 0.45104138388552817

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

[38]:
```
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]
```

[38]:

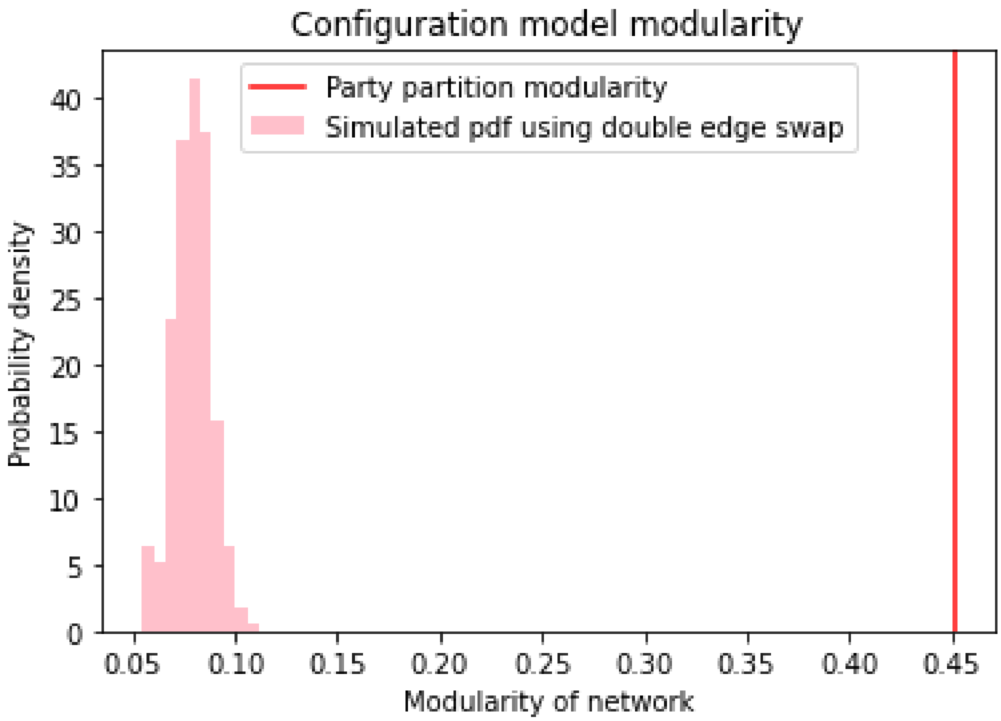
	S	DF	V	EL	LA	SF	RV	KF	ALT	UFG	JF	IA
0	50.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
1	0.0	37.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
3	5.0	0.0	2.0	19.0	6.0	10.0	10.0	6.0	4.0	3.0	0.0	1.0
4	1.0	0.0	35.0	0.0	1.0	0.0	0.0	2.0	0.0	2.0	1.0	0.0
5	0.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0	2.0	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.

[39]:

```
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.

[40]:
```
# For export to the website
Ddf[s.sort_values(ascending=False).index].columns.tolist(), Ddf[s.
 ↪sort_values(ascending=False).index].values.tolist()
```

[40]:
```
(['S', 'DF', 'V', 'EL', 'LA', 'SF', 'RV', 'KF', 'ALT', 'UFG', 'JF', 'IA'],
 [[50.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 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],
  [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0],
  [5.0, 0.0, 2.0, 19.0, 6.0, 10.0, 10.0, 6.0, 4.0, 3.0, 0.0, 1.0],
  [1.0, 0.0, 35.0, 0.0, 1.0, 0.0, 0.0, 2.0, 0.0, 2.0, 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]])
```

[41]:
```
# 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.

[43]:
```
# To install the sentida module
# `pip install sentida`
```

[44]:
```
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.

[45]:
```
votingString = "2015"
speechString = "20151"
```

[46]:
```
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())
```

[47]:
```
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.

[48]:
```
def _sentiment_analysis_helper(text, output):
    return Sentida().sentida(text, output=str(output), normal = True, speed =␣
↪"normal")
```

[49]:
```
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:

Analysis not run in notebook, but can if the last boolean of the following function is switched to True. It will take a while, be ware!

[51]:
```
[party_sent_avg, party_sent] = SentAnalysis(party_docs, "party", "partySent",␣
↪True, False)
```

Average sentiment of each party

[52]:
```
dict(sorted(party_sent_avg.items(), key=lambda item: item[1], reverse=True))
```

[52]:
```
{'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

[53]:
```
dict(sorted(party_sent.items(), key=lambda item: item[1], reverse=True))
```

[53]:
```
{'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.

[57]:
```
[person_sent_avg, person_sent] = SentAnalysis(person_docs, "person",␣
↪"personSent", False, False)
```

Average sentiment of persons in the danish parliment

[58]:
```
sorted(person_sent_avg.items(), key=lambda item: item[1], reverse=True)[:5]
```

[58]:
```
[('Jan Erik Messmann', 0.8252734671700788),
 ('Sisse Marie Welling', 0.6770443349753699),
 ('Daniel Toft Jakobsen', 0.5550600436810947),
 ('Roger Matthisen', 0.5362402039356772),
 ('Lars Christian Lilleholt', 0.5246535010154891)]
```

Total sentiment of persons in the danish parliment

[60]:
```
sorted(person_sent.items(), key=lambda item: item[1], reverse=True)[:5]
```

[60]:
```
[('Pia Kjærsgaard', 2446.505552960304),
 ('Finn Sørensen', 1883.8315982844051),
 ('Lisbeth Bech Poulsen', 1771.844211748423),
 ('Christian Juhl', 1741.7137698197866),
 ('Martin Lidegaard', 1637.8114593187854)]
```

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

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

$$A : Number\ of\ words\ in\ text \quad (1)$$

$$B : Number\ of\ periods\ (all\ ending\ characters\ are\ converted\ to\ periods) \quad (2)$$

$$C : Number\ of\ long\ words\ (more\ than\ 6\ letters) \quad (3)$$

Function for LIX is defined as a function

[61]:
```
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:

[62]:
```
party_LIX = dict([(party, LIX(doc)) for (party, doc) in party_docs.items()])
```

[63]:
```
dict(sorted(party_LIX.items(), key=lambda item: item[1], reverse=True))
```

[63]:
```
{'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

[64]:
```
person_LIX = dict([(person, LIX(doc)) for (person, doc) in person_docs.items()])
```

[65]:
```
sorted(person_LIX.items(), key=lambda item: item[1], reverse=True)[:5]
```

[65]:
```
[('Hans Christian Thoning', 55.225820962663065),
 ('Carsten Kudsk', 53.49299719887955),
 ('Kirsten Brosbøl', 50.04360183841315),
 ('Steen Holm Iversen', 49.19016453079425),
 ('Jakob Sølvhøj', 48.378575575413365)]
```

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.

[66]:
```
from tf_idf import TfIdf
from nltk.corpus import stopwords
```

[67]:
```
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

[68]:
```
Party_TF_IDF_ALL = SpeechTFIDF(party_docs, "Party")
```

[69]:
```
Party_TF_IDF_ALL
```

[69]:
```
   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

[70]:
```python
Person_TF_IDF_ALL = SpeechTFIDF(person_docs, "Person")
```

[71]:
```python
Person_TF_IDF_ALL.sample(5)
```

[71]:
```
                      Person  \
133           Mikkel Dencker
157       Peter Kofod Poulsen
90   Kenneth Kristensen Berth
76                Jonas Dahl
62            Jan E. Jørgensen

                                                 Words
133  {'tak': 1.199466075581251e-05, 'formand': 8.18…
157  {'tak': 6.068199041129711e-05, 'formand': 0.00…
90   {'må': 0.00014273820803926081, 'sige': 0.00012…
76   {'tak': 2.0027663423754422e-05, 'debatten': 0…
62   {'bare': 0.00011554448865290144, 'høre': 2.675…
```

1.7.5 Member information

Connecting name and id for each Person in data

[72]:
```python
actors = pd.read_csv("data/ft/Aktør.csv", index_col = False)
```

[73]:
```python
ActorNameDict = dict(zip(actors["navn"], actors["id"]))
```

[74]:
```python
import re
def getTag(bio,tag):
    if bio:
```

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

[75]:
```
actor_types = pd.read_csv('data/ft/Aktørtype.csv').set_index('id', drop=False)
```

[76]:
```
actors = actors.fillna('')
actors['parti'] = actors['biografi'].apply(getTag, args=('partyShortname',))
```

[77]:
```
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

[79]:
```
#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)
```

[80]:
```
PartyData
```

[80]:
```
         totalSent   avgSent        lix  \
ALT    7184.792473  0.342537  40.253608
DF     9816.984189  0.244083  37.484111
EL    11673.655360  0.217732  38.870138
IA     1081.779810  0.462341  43.005482
JF      275.704860  0.331436  38.510285
KF     3637.448697  0.296384  37.824342
LA     5573.353110  0.261870  39.577172
RV     4968.206200  0.313630  40.520201
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

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

Saving Data

[81]:
```
PartyData.to_csv("data/AppData/" + speechString + "/PartyData"+ speechString +".
↪csv")
```

1.7.7 People

Actor name to id dict

[83]:
```
#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]

#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", "totalSent", "avgSent", "lix",␣
↪"tfIdf"]]
```

Saving data

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

1.7.8 Export to json

```
[87]: import json
```

Parties

```
[88]: 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)
```

```
[88]: 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

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

```
[90]: 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)
```

[90]:
```
id
15757.0    None
15758.0    None
18.0       None
16374.0    None
148.0      None
           …
15772.0    None
296.0      None
155.0      None
1619.0     None
191.0      None
Length: 196, dtype: object
```

---

[91]:
```
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.