Analysis of the Banzhaf Index Using National Council Voting Data

Dominic Steiner

Report for Mathematics in Politics and Law, ETH Zrich

Abstract. The Banzhaf voting power index provides a formula to calculate voting power. Voting power can be used to study fairness or political representation. However, by calculating the Banzhaf index of a voting body, we implicitly assume that all voter choose their vote at random. This report studies how these assumptions fail when applied to the Swiss National Council. To achieve this, we analyzed thousands of votes taken during the last eight years in the National Council.

1 Introduction

While researching or developing a voting body, it is often desirable to calculate how the power is distributed between its members. The distribution can be used to study fairness and political representation. One popular method of calculating voting power is the Banzhaf voting power index. This index assumes a fairly simple statistical model of a voting body. However, to fit complex political dynamics in a simple model, the index makes broad assumptions. This report tries to answer if the Banzhaf model fits reality. Therefore, we will study the Banzhaf index in the specific example of the Swiss National Council.

1.1 The Swiss National Council

The National Council [3] is one of the two Chambers in the Swiss legislature. The 200 seats in the council are allocated to the 26 cantons according to their respective populations. All members are elected every four years. The period in between is called a legislature. This report will use voting data from the 49th and 50th legislature. In the National Council members are organized in factions [5] by party. Usually, only a few minor parties with insufficient members to form their own faction join the faction of a politically similar bigger party. In this report, we will consider factions, not parties, because the voting data lists faction membership, not party membership. Nevertheless, we will still use the abbreviation of the main party in the faction as its name, to help with easier recognition.

	Faction seats in legislature	49th	50th
SVP	Faction of the Swiss People's Party (V)	56	68
SP	Faction of the Social Democratic Party (S)	46	43
FDP	Faction FDP.The Liberals (RL)	30	33
CVP	Faction of the Christian Democratic People's Party (C, CE)	31	30
GSP	Faction of the Green Party (G)	15	12
BDP	Faction of the Conservative Democratic Party (BD)	9	7
GLP	Faction of the Green Liberal Party (GL)	12	7

2 Voting Power

Let us dive deeper into the concept of voting power. First, we need to define a few terms. Imagine a voting body, with n members that vote on resolutions. Each member has a voting weight of w_i. A coalition is the set of members voting in favor of the resolution. The voting rule function $v()$ determines for a coalition if it can pass the resolution. A resolution passes if we have a coalition C with $v(C) = 1$. i.e., a body of three members $\{Alice, Bob, Charles\}$ with all equal voting weight, and a simple majority voting rule. In this case, $v(C)$ is one if C contains two or more members and zero otherwise.

Definition Winning Coalition: Considering a coalition C and a decision function $v()$, C is winning if and only if $v(C) = 1$. For the example above a valid winning coalition would be $C = \{Alice, Bob\}$.

Definition Pivotality: Considering a member m_i which is part of a winning coalition C. If the coalition $C \setminus m_i$ (C without m_i) is no longer winning then m_i is considered pivotal in respect to C. For our example both $Alice$ and Bob are considered pivotal because both $C \setminus \{Alice\} = \{Bob\}$ and $C \setminus \{BOB\} = \{Alice\}$ only have one member and so no longer contain a majority.

2.1 Banzhaf Voting Power Index

The fundamental idea of the Banzhaf index [6] is relatively simple. A pivotal member has all the power; he has the deciding vote. A non-pivotal member has no power at all. Concretely the index gives a pivotal member a power index of one and a non-pivotal member a power of zero. The problem is that pivotality depends on the coalition. So how can we calculate an overall power index which is independent of a specific coalition? The Banzhaf index solves this problem by assuming that all coalitions are equally likely. To calculate the voting power of party p in a body with n parties, we have to iterate over all possible winning coalitions and count how often p is pivotal. In the end, we normalize the results so that the total power sums up to one. The python code in appendix 8.2 is a simple implementation of the Banzhaf voting power index.

Example of Banzhaf Voting Power Index Calculation: Consider this new voting body with the members $\{Alice, Bob, Charles\}$, the voting weights $w_{Alice} = 3$, $w_{Bob} = 2$, $w_{Charles} = 1$ and the voting rule $v(C) = 1$ if the sum of the voting weight of all members in C is greater or equal to 4 otherwise $v(C) = 0$. Let $w_{sum}(C)$ denote the sum of all voting weight in C. First look at all possible coalitions:

$$C_0 : \quad w_{sum}(\{Alice\}) = 3$$
$$C_1 : \quad w_{sum}(\{Bob\}) = 2$$
$$C_2 : \quad w_{sum}(\{Charles\}) = 1$$

$$C_3: \quad w_{sum}(\{Alice, Bob\}) = 5$$

$$C_4: \quad w_{sum}(\{Alice, Charles\}) = 4$$

$$C_5: \quad w_{sum}(\{Bob, Charles\}) = 3$$

$$C_6: \quad w_{sum}(\{Alice, Bob, Charles\}) = 6$$

We have seven possible coalitions for this body but only C_3, C_4 and C_6 are winning. In C_3 Alice and Bob are pivotal, in C_4 Alice and Charles are pivotal but in C_6 only Alice is pivotal. When we add everything together, Alice was three times pivotal, Bob and Charles were once pivotal. Normalized we get:

$$power_{Alice} = \frac{3}{5} \quad power_{Bob} = \frac{1}{5} \quad power_{Charles} = \frac{1}{5}$$

2.2 Application on Swiss National Council

Later we will apply the Banzhaf index on the National Council. For this, the voting members in our model are the factions as in section 1.1. The voting weight w_f of each faction f is the number of members that are part of faction f. The voting rule function $v()$ is defined as follows: $v(C) = 1$ if the sum of the voting weight of all factions in C is greater than 100 otherwise $v(C) = 0$.

3 Voting Data

The Swiss Government publishes exact voting data for the National Council [4]. At the time of writing, the published data lasts from the 2011 winter session to the 2019 winter session. This period gives us the complete data for the 49th and the 50th legislature. Unfortunately, the data for each session comes in separate excel spreadsheet files. To analyze the data effectively, we wrote a python parser program (appendix 8.3) that reads all the files and prepares them for further analysis.

The voting data contains information on how every member voted for every vote taken in the council. Unlike our assumption for the Banzhaf index, the members not only vote *yes* and *no* but also abstain from the vote (*abstain*) or not be present for the vote at all (*novote*). We also have to change the voting rule. Since now members can abstain, it is enough to have more *yes* votes than *no* votes. In case of a draw, the *no* voters win the vote.

At the end of the parsing process, the parser returns the data object. The data object contains $members$, $factionList$ and $votes$. $members$ map member ids to individual national council members. $factionList$ maps each faction to a list of member IDs. $votes$ is a list of vote objects. Each vote contains the sets yes, no, $abstain$ and $novote$. These sets contain member IDs of the members that voted the particular way for this vote. Now we can use the data to draw graphs and make calculations. For example, we can plot a histogram fig.1 of $200 - |novote|$, which is the number of members present for a vote.

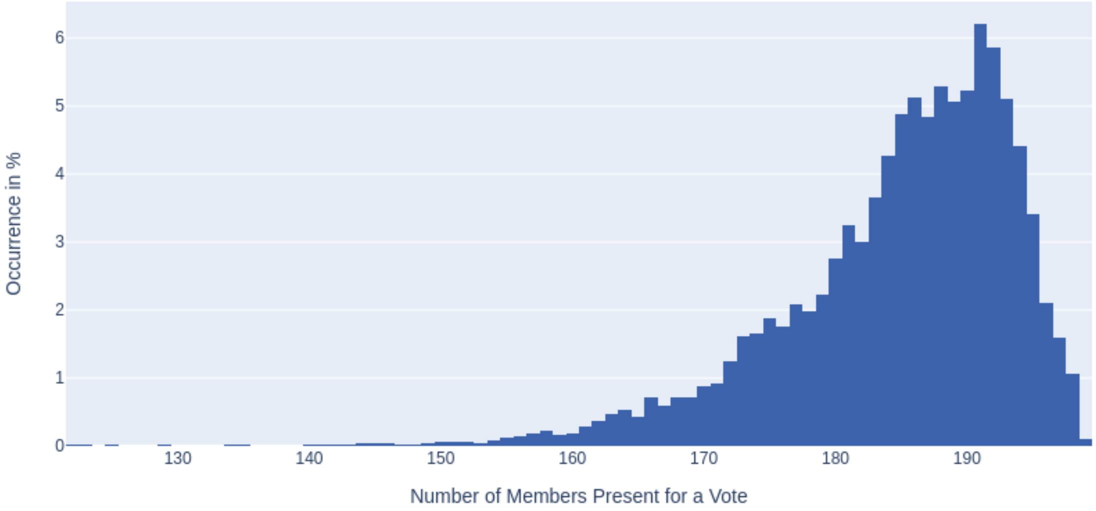

Fig. 1. Histogram of number of National Council members present for Votes

4 Equal Distribution of Winning Coalitions

One critical assumption the Banzhaf index takes is that all winning coalitions are equally likely, which means a party votes 'yes' or 'no' at random independent of all other parties. However, it is not clear how realistic this assumption is, especially in a political context where often the views of the different members are quite diverse. An example from the National Council: The factions SVP and SP have together a majority, but we still do not expect much collaboration between these parties as their political views are very different. With the voting data, we can now see how realistic this assumption is for the National Council. The objective is to count how often each winning coalition occurs. Our python program (appendix 8.9) iterates through all the votes. We define the vote of a faction as the majority vote of its members. All the faction which votes on the winning side of a vote, build its winning coalition. While we are going through the votes, we count how often each coalition occurs. The graph in fig. 2 shows the top 40 winning coalitions ordered by relative occurrence in percent. The graph is strong evidence against the random vote model. During the 50th legislature, there were 163 unique winning coalitions. However, the top 10 most occurring coalitions are responsible for more than 70% of all votes.

5 Empirical Voting Power Index

The graph fig. 2 is strong evidence that the assumption of equally distributed winning coalitions does not hold for the national council. One way of solving this problem is to use the

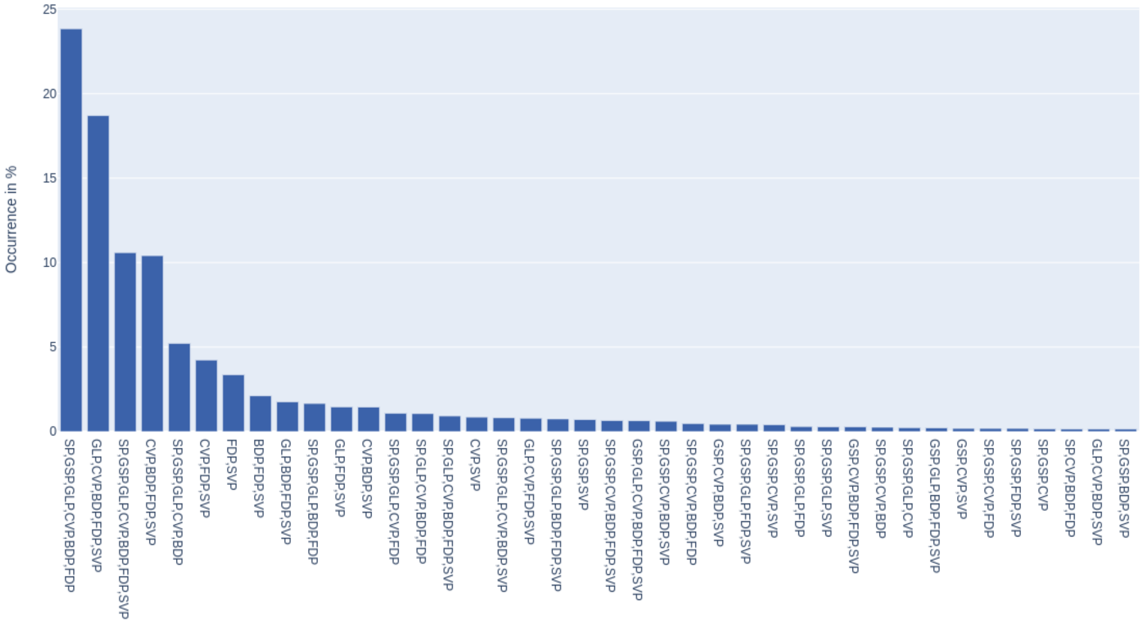

Fig. 2. Top 40 winning coalition ordered by relative occurrence in percent

winning coalitions in the voting data instead of iterating over all possible winning coalitions to calculate the index. However, before we can do this, we have to define pivotality in the empirical context.

Definition Pivotality in Empirical Model: A faction f is pivotal if it can change the outcome of the vote. Before we could use the voting weight of the party to decide if it is pivotal now, the voting weight can change depending on how many members are present or abstain. Also, the members of a faction are not required to vote on the party line. Given a vote v we can calculate the sets yes_f , no_f, $abstain_f$ and $novote_f$ for the faction f. yes_f is the set of council members voting 'yes' in vote v and are members of faction f, similar for the other sets. Intuitively a faction is pivotal if all faction members on the winning side switch their vote and the outcome changes.

$$\begin{aligned} f \textbf{ is pivotal if}\text{: } & \text{case vote decision is yes: } |yes| - |yes_f| \leq |no| + |yes_f| \\ & \text{case vote decision is no: } |yes| + |no_f| > |no| - |no_f| \end{aligned}$$

With the empirical definition of pivotal, we can iterate through all votes and count how often each faction is pivotal. Like the Banzhaf index, we normalize the results.

5.1 Results

Now we can compare the results of the Banzhaf index with the empirical version (graph fig. 3). Most of the parties are pivotal more often than the Banzhaf index predicts. Only the SVP is fewer times pivotal than expected in the Banzhaf index. Overall the empirical voting power is substantially different from the Banzhaf index. Interestingly the smallest factions have the biggest diffreces. Somehow these small factions manage to use their limited power more efficiently then the Banzhaf index expects.

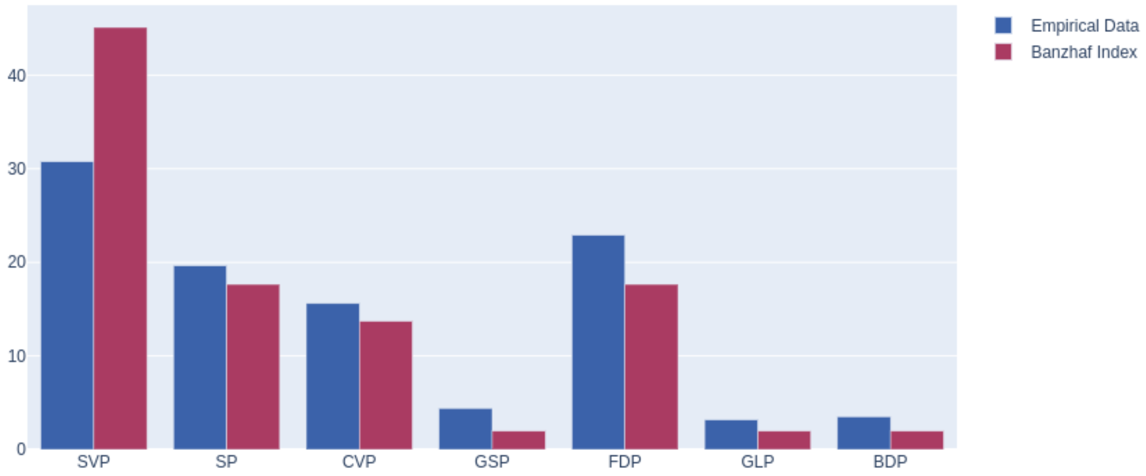

Fig. 3. Banzhaf index and empirical voting power for 50th legislature

6 Simulated Voting Data

Previously we developed an empirical voting power index similar to the Banzhaf index. In this section, we want to test how this index compares if we use simulated data that follows the assumption made in the Banzhaf index. We use the same factions with the same number of seats as before. However, instead of using real historical data, we will simulate random voting. The simulator.py (appendix 8.4) file contains code that generates the same data structure as the parser, but this time the factions vote completely at random ’yes’ or ’no’. This simulated data now follows the assumption that all winning coalitions are equally likely. We can visualize this by redrawing the same graph as in fig. 2 but now with our simulated data (fig. 4). The simulated data set used in this graph consists of 5000 votes with the faction sizes of the 50th legislature.

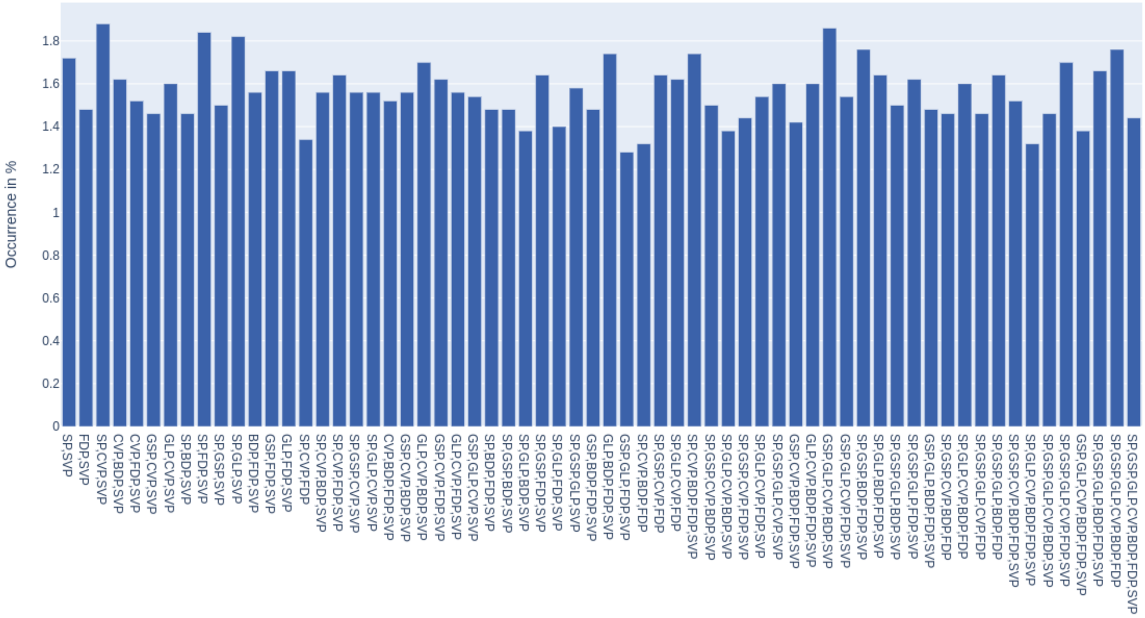

Fig. 4. Relative occurrence of winning coalition Wining coalition in percent

Finally, we can combine the empirical index with real-world data, the empirical index with simulation data, and the Banzhaf index all together in a graph (fig. 5).

6.1 Result

Graph fig. 4 shows that all winning coalitions occur about the same number of times in the simulated data. From this, we expect that all coalition have that same weight as in the Banzhaf index. And indeed the result from the empirical index with simulated data is nearly identical to the Banzhaf index. This result is strong evidence that variation between the empirical index with real-world data and the Banzhaf index is indeed due to a different distribution of the winning coalitions.

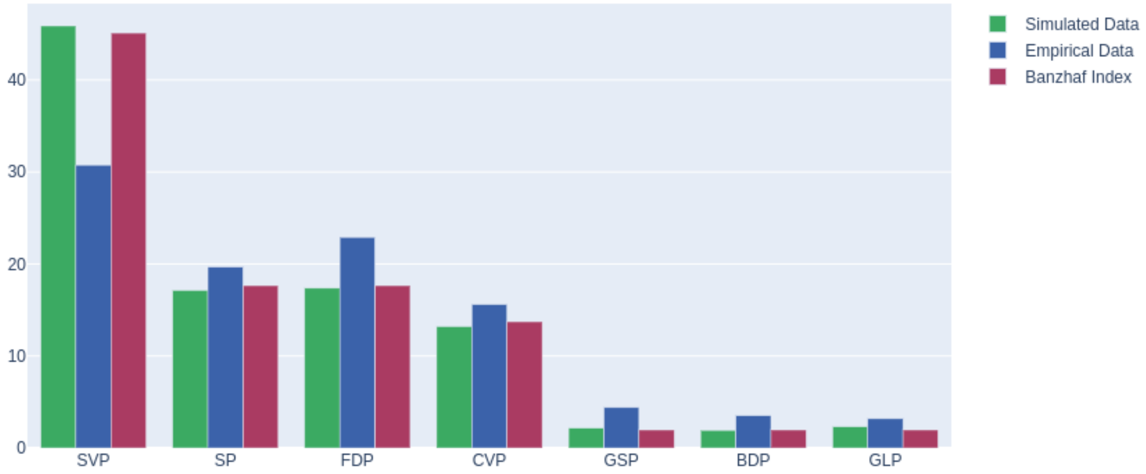

Fig. 5. Empirical index with simulated data, empirical index real voting data and Banzhaf index for 50th legislature

7 Conclusion

In the previous sections, we have shown that when applied to simulated data that follows the assumptions, the empirical index returns the same results as the Banzhaf index. If the national council would follow the Banzhaf assumption, it would also return the same results. All this is evidence that the assumption of random voting does not hold for the national council. We expect that ideology and general party behavior has a substantial effect on how often a party is pivotal. However, our analysis does not account for potential deals between the parties. Nor does it make a distinction between types of votes. The data list all votes taken in the full National Council. However, intuitively a vote about a small amendment is less important than the final vote over a bill. Often these less important votes have fewer voting members. Maybe votes should be weighted by the number of members voting. Also, one can not forget that the National Council does not stand alone. The Council of States probably also has an impact on voting decisions in the National Council. Unfortunately, the Council of States does not publish its voting data.

8 Appendix

8.1 Python Source Code

The voting data came in excel spreadsheets, but to parse theme with python, they had to be converted into .csv text files. Also, the text files had to be converted into UTF-8 encoding so that python can correctly handle the spacial characters. The converted files and all python source code can be downloaded on GitHub [7][1][2].

8.2 PBI.py

```python
import itertools as itt
#calculates Penrose-Banzhaf index (PBI) for parties with >=t votes for a winning coalition
#parties beeing dict form party key to number of seats
#t is the min number of votes needed for a winning coalition
def votingPower(parties, t = 100, v=False):
    power = dict()
    countPivotal = 0

    #initialize power to zero
    for p in parties:
        power[p] = 0

    #iterate over all subsets
    for i in range(1, len(parties) + 1):
        for partition in itt.combinations(parties.keys(), i):
            s = 0
            #sum all the votes in the partition
            for x in partition:
                s += parties[x]

            if v:
                print(partition, s)

            if s >= t:
                #the partition in winning
                for p in partition:
                    #check for all parties in the partion if they are pivotal
                    if s - parties[p] < t:
                        #party in pivotal add 1 to the power
                        countPivotal += 1
                        power[p] += 1

    if v:
        print(countPivotal)

    if countPivotal > 0:
        #divide the power of each party by the number of subsets without this party
        for p in power:
            power[p] /= countPivotal

    return power
```

8.3 parser.py

```python
import csv
import classes

class VotingData:
    votes = None
    members = None
    factionList = None

    def __init__(self):
        self.votes = []
        self.members = dict()
        self.factionList = dict()


#loads all the session form sessions.csv
def loadSessions(file = 'sessions.csv', csv_data_dir = './csv_data'):
    #load session data form sessions.csv
    sessions = [] #list of all the sessions
    with open(file) as sessions_csv:
        sessionno = 0
        csv_reader = csv.reader(sessions_csv, delimiter=',')
        for l in csv_reader:
            sessions.append(classes.Session(sessionno, l[1], l[2], csv_data_dir + '/' + l[3]))
            sessionno += 1
    return sessions

#load the size of each faction form factions.csv
#the avoids the problem that the faction member list contain all members that were part of the faction in some session
#so the length of the member list can be higher then the actual member count of a faction.
def loadFactionsSize(file = 'factions.csv', period='50'):
    factions_count = dict()
    with open(file) as factions_csv:
        csv_reader = csv.reader(factions_csv, delimiter=',')
        l = next(csv_reader)
        periodIndex = None
        for i,item in enumerate(l):
            if item == period:
                periodIndex = i
                break
        if periodIndex == None:
            #could not fined period
            return None

        l = next(csv_reader)
        while True:
            try:
                factions_count[l[0]] = int(l[periodIndex])
                l = next(csv_reader)
            except StopIteration:
                break

        t = 200 - sum(factions_count.values())
        if t > 0:
            factions_count[’None’] = t


        return factions_count


#list of members for each faction
def genFactions(members):
    factions = dict()
    for mem in members:
        f = members[mem].faction
        if f in factions:
            factions[f].append(members[mem].bioid)
        else:
            factions[f] = [members[mem].bioid]
    return factions

#counts members that were part of each fation a some point in parsed data
def genFactionSize(factions):
    factions_size = dict()
    for f in factions:
        factions_size[f] = len(factions[f])

    return factions_size

#parses information about the members
def parseMems(r, session, members):
    #members = dict()
    posIdMap = []
    line = next(r)
    start = 0
    end = 0
    for i, item in enumerate(line):
        if item == ’CouncillorId’:
            start = i+1
            break

    line = next(r)

    #enumerate ID
    for i, item in enumerate(line[start:]):
        if item == ’’:
            end = i + start
            break

        posIdMap.append(item)
        if not item in members:
            members[item] = classes.Member(item)
        (members[item].ismem).append(session.idno)

    ’’’
    #enumerate bioid
    line = next(r)
    for i,item in enumerate(line[start:end]):
        members[posIdMap[i]].bioid = item
    '''

    #enumerate name
    line = next(r)
    for i,item in enumerate(line[start:end]):
        #print(i), print(item)
        members[posIdMap[i]].name = item

    #skip line (this line only shows members are part of national council)
    next(r)
    #enumerate faction membership
    line = next(r)
    for i,item in enumerate(line[start:end]):
        members[posIdMap[i]].faction = item

    #enumerate canton
    line = next(r)
    for i,item in enumerate(line[start:end]):
        members[posIdMap[i]].canton = item

    #enumerate birthdata
    line = next(r)
    for i,item in enumerate(line[start:end]):
        members[posIdMap[i]].birthdate = item

    return

#parses information about the votes
def parseVotes(r, s, votes):
    l = next(r)
    voters = []
    #votes = []
    voteno = 0
    start = 0
    while l[0] != 'VoteDate':
        l = next(r)
        start += 1

    #voters contains all the voters in a session
    endVotes = 0
    for i, item in enumerate(l[12:]):
        if item != 'Decision':
            voters.append(item)
        else:
            endVotes = i + 12
            break
    #print(len(voters))

    g = set()
    l = next(r)


    #need to handle:
    vote_none = '' #member was not yet or is no longer a national coucil member
    vote_yes = 'Ja'
    vote_no = 'Nein'
    vote_novote = ['Entschuldigt', 'Hat nicht teilgenommen']
    vote_abstain = 'Enthaltung'
    vote_pres = 'Der Pr sident stimmt nicht'
    decision_set = {'yes', 'no'}


    while True:
        try:
            #store vote meta data
            v = classes.Vote(voteno)
            v.date = l[0]
            v.affairId = l[4]
            v.title = l[5]
            v.session = s
            v.sessionId = s.idno
            #store how each member voted by adding the members bioid to the yes/no/abstain/noset set
            for i, item in enumerate(l[12:endVotes]):
                if item == vote_none:
                    pass
                elif item == vote_yes:
                    v.yes.add(voters[i])
                elif item == vote_no:
                    v.no.add(voters[i])
                elif item == vote_abstain:
                    v.abstain.add(voters[i])
                elif item in vote_novote:
                    v.novote.add(voters[i])
                elif item == vote_pres:
                    pass
                else:
                    print('err vote: ' + item)

            #store decision data as stored in file and compare with calculated decision value
            v.decision = l[endVotes].lower()

            if v.decision != v.getDecision() or (not v.decision in decision_set):
                v.decision = 'err: \"' + v.decision + '\"'
            votes.append(v)
            voteno += 1
            l = next(r)
        except StopIteration:
            break


#parses all the session data files
#to parse can limit the session to parse
def parseFiles(sessions, toParse = {}):
    data = VotingData()

    for s in sessions:
        if s.idno in toParse or len(toParse) == 0:
            with open(s.path) as csv_file:
                csv_reader = csv.reader(csv_file, delimiter=',')
                print('-', end='')
                #print(str(key) + '. reading ... ' + csv_file.name)
                #parseFile(csv_reader)
                parseMems(csv_reader, s, data.members)
                parseVotes(csv_reader, s, data.votes)
                csv_file.close


    data.factionList = genFactions(data.members)

    return data
```

8.4 simulator.py

```
import random
import parser
import classes

#simulate voting data were faction vote randomly
#data has the same structure as the data returned form parser.py

#generates faction member lists with number of members according to faction_size
def genFactions(faction_size):
    t = 0
    simFaction = dict()

    for name, count in faction_size.items():
        simFaction[name] = list(range(t, t+count))
        t += count

    return simFaction

#generates a member id to member maping that can be use to determine the faction coresponding to a member id
def genMembers(factionList):
    members = dict()

    for f in factionList:
        for mem in factionList[f]:
            members[mem] = classes.Member(mem)
            members[mem].faction = f
    return members


#generates vote array containing n votes where each faction votes randomly yes/no
def genVotes(simFaction, n, unity=1):
    simVotes = []

    for i in range(n):
        vote = classes.Vote(i)

        for faction in simFaction:
            withFaction = []
            againstFaction = []

            for mem in simFaction[faction]:
                if random.random() < unity:
                    withFaction.append(mem)
                else:
                    againstFaction.append(mem)

            if random.randint(0,1) == 0:
                #faction votes yes
                vote.yes.update(withFaction)
                vote.no.update(againstFaction)
            else:
                #fation votes no
                vote.no.update(withFaction)
                vote.yes.update(againstFaction)

        vote.decision = vote.getDecision()
        simVotes.append(vote)

    return simVotes

#generates voting data with n votes
def genData(faction_size, n, unity = 1):
    simData = parser.VotingData()

    simData.factionList = genFactions(faction_size)
    simData.votes = genVotes(simData.factionList, n, unity=unity)
    simData.members = genMembers(simData.factionList)

    return simData
```

8.5 classes.py

```
import itertools as itt
import plotly.graph_objects as go

#This file contain the diffrent class definition
#These datastructures are use by the parser.py to load the date in memory

#for raming faction to party abbreviations
faction_names = {'V': 'SVP', 'S': 'SP', 'RL': 'FDP', 'C': 'CVP', 'CE': 'CVP', 'G':
     'GSP', 'BD': 'BDP', 'GL': 'GLP', '-': 'None'}

party_names = {'SVP', 'SP', 'FDP', 'CVP', 'GSP', 'BDP', 'GLP', 'None'}
#orders party by member count
party_order = {'SP': 1, 'GSP': 2, 'GLP': 3, 'CVP': 4, 'BDP': 5, 'FDP': 6, 'SVP':
    7, 'None': 8}




class KeyList:
    key = None
    value = None
    isSorted = False

    def __init__(self):
        self.key = []
```

```python
        self.value = []

    def append(self, key, item):
        self.isSorted = False
        self.key.append(key)
        self.value.append(item)

    def extend(self, keys, items):
        assert(len(keys) == len(items))
        self.isSorted = False
        self.key.extend(keys)
        self.value.extend(items)

    def sort(self, key= lambda x: x):
        temp = sorted(zip(self.value, self.key), key= lambda x: key(x[0]))
        self.key = [k for v, k in temp]
        self.value = [v for v, k in temp]
        self.isSorted = True

    def normalize(self, percent= False):
        t = sum(self.value)

        if percent:
            t *= 100

        for i, _ in enumerate(self.value):
            self.value[i] /= t

    def keyMap(self, f):
        self.key = list(map(f, self.value))

    def valueMap(self, f):
        self.value = list(map(f, self.value))

    def bar(self):
        fig = go.Figure([go.Bar(x=self.key, y=self.value)])
        fig.show()

    def __sizeof__(self):
        return len(self.key)

    def __str__(self):
        return str(list(zip(self.key, self.value)))

    def __iter__(self):
        return iter(zip(self.key, self.value))

#stores meta date of a session
#a session corresponds to one *.csv file
class Session:
    idno = 0
    year = ’0000’
    no = ’0’
    path = ’’

    def __init__(self, idno, y, n, path):
        self.idno = idno
        self.year = y
        self.no = n
        self.path = path

    def __str__(self):
        return str(self.idno) + ' ' + self.path

    def __eq__(self, y):
        if isinstance(y, Session):
            return self.idno == y.idno
        else:
            return False

#stores data for a inividual national council member
class Member:
    idno = 0
    bioid = '0'
    name = 'none'
    faction = 'none'
    canton = 'none'
    birthdate = '0000-00-00'
    ismem = None

    def __init__(self, i):
        self.bioid = i
        self.ismem = []

    def __str__(self):
        f = self.faction
        if len(f) == 1:
            f += ' '
        return '#' + str(self.bioid) + ' ' + f + ' ' + self.canton + ' ' + self.name + ' c' + str(len(self.ismem))

    def __eq__(self, y):
        if isinstance(y, Member):
            return self.bioid == y.bioid
        else:
            return False

#stores data of a inivdual vote
class Vote:
    idno = 0
    date = '0000-00-00 00:00:00'
    affairId = '000.00'
    title = 'none'
    decision = 'undef'

    #set of member bioids
    yes = None #members voted yes
    no = None #members voted no
    abstain = None #members voted abstain
    novote = None #members not present for vote

    sessionId = 'none'
    session = None #ref to the session this vote was taken

    def __init__(self, i):
        self.idno = i
        self.yes = set()
        self.no = set()
        self.abstain = set()
        self.novote = set()

    #returns the decison based on the yes/no set size
    def getDecision(self):
        if len(self.yes) > len(self.no):
            return 'yes'
        if len(self.no) >= len(self.yes):
            return 'no'
        return 'undef'

    # -tests if party is pivotal for this vote
    # -party is set/list of members bioids
    # -party can contain members that were not national council members
    #  at the time of the vote
    def isPivotal(self, party, abstain=True, novote=False, draw=False):
        p_yes = 0
        p_no = 0
        p_abstain = 0
        p_novote = 0

        for mem in party:
            if mem in self.yes:
                p_yes += 1
            elif mem in self.no:
                p_no += 1
            elif mem in self.abstain:
                p_abstain += 1
            elif mem in self.novote:
                p_novote += 1

        if not abstain:
            #do not consider members voted abstained
            p_abstain = 0
        if not novote:
            #not consider not present members
            p_novote = 0


        #decision was no
        if self.decision == 'no':
            new_yes = len(self.yes) + p_no + p_abstain + p_novote
            new_no = len(self.no) - p_no
            if new_yes > new_no:
                #can change vote pivotal
                return True
            elif new_no > new_yes:
                #can not change vote not pivotal
                return False
            else:
                #draw by default False
                return ’draw’

        #decision was yes
        if self.decision == ’yes’:
            new_yes = len(self.yes) - p_yes
            new_no = len(self.no) + p_yes + p_abstain + p_novote
            if new_no > new_yes:
                return True
            elif new_yes > new_no:
                return False
            else:
                return ’draw’

    # -messure for how unifed the party was in this vote
    # -returns larges fraction of the party voted the same
    # -party is list/set of members bioids
    # -members not part of the national council at the time of the vote
    #  are not counted in any case
    def unity(self, party, abstain=True, novote=False):
        p_yes = 0
        p_no = 0
        p_abstain = 0
        p_novote = 0

        for mem in party:
            if mem in self.yes:
                p_yes += 1
            elif mem in self.no:
                p_no += 1
            elif mem in self.abstain:
                p_abstain += 1
            elif mem in self.novote:
                p_novote += 1

        if not abstain:
            #not consider abstain voters
            p_abstain = 0

        if not novote:
            #not consider abstain voters
            p_novote = 0

        p_max = max({p_yes, p_no, p_abstain, p_novote})
        p_sum = sum({p_yes, p_no, p_abstain, p_novote})

        if p_sum == 0:
            return 1

        return p_max / p_sum

    #enables print for Vote class
    def __str__(self):
        out = str(self.sessionId) + ’# ’
        out += str(self.idno)
        out += ’ ’ + self.decision
        out += ' yes:' + str(len(self.yes))
        out += ' no:' + str(len(self.no))
        out += ' abstain:' + str(len(self.abstain))
        out += ' novote:' + str(len(self.novote))
        return out

# class to store voter profile of a vote for a party
class VoteProfile:
    yes = 0
    no = 0
    abstain = 0
    novote = 0

    def __init__(self):
        self.yes = 0
        self.no = 0
        self.abstain = 0
        self.novote = 0

    def total(self):
        return self.yes + self.no + self.abstain + self.novote

    def partyVote(self, abstain=True, novote=False):
        out = 'None'
        t = -1

        if self.yes > t:
            out = 'yes'
            t = self.yes

        if self.no > t:
            out = 'no'
            t = self.no
        elif self.no == t:
            out = 'draw'

        if abstain and self.abstain > t:
            out = 'abstain'
            t = self.abstain
        elif abstain and self.abstain == t:
            out = 'draw'

        if novote and self.novote > t:
            out = 'novote'
            t = self.novote
        elif novote and self.novote == t:
            out = 'draw'

        return out

    def __str__(self):
        out = 'party vote: ' + self.partyVote()
        out += ' yes: ' + str(self.yes)
        out += ' no:' + str(self.no)
        out += ' abstain: ' + str(self.abstain)
        out += ' novote: ' + str(self.novote)
        out += ' total: ' + str(self.total())
        return out
```

8.6 plot.py

```
import plotly.graph_objects as go
import plotly.express as px
import mathHelper as mh

def hist(data):
    fig = go.Figure(data=[go.Histogram(x=data, histnorm='percent')])
    fig.show()

#colors = ['rgb(0, 154, 46)', 'rgb(255, 0, 0)', 'rgb(6, 60, 255)','rgb(255, 135, 0)', 'rgb(42, 232, 2)', 'rgb(255, 220, 0)', 'rgb(190, 239, 0)']
faction_color = ['#009A2E', '#FF0000', '#063CFF', '#FF8700', '#2AE802', '#FFDC00', '#BEEF00']

def factionBar(x, y):
    fig = go.Figure([go.Bar(x=x, y=y, marker_color=faction_color)])
    fig.show()

# prints bar chart form dict d
# key function applied to the dict keys
# value function applied to the dict items
# if relativ is true the average is substracted form all values
# if sort is true values are sorted by value
# sortkey function can modify sort key
def bar_dict(d, key= lambda x: x, value= lambda x: x, relativ= False, sort=False, sortKey= lambda x: x, xlable='', ylable='', titel=''):
    x = []
    y = []

    for k in d:
        x.append(key(k))
        y.append(value(d[k]))

    if relativ:
        mean = mh.mean(y)
        y = list(map(lambda x: x - mean, y))

    if sort:
        temp = sorted(zip(x, y), key= lambda x: sortKey(x[1]))
        x = [k for k, _ in temp]
        y = [v for _, v in temp]

    fig = go.Figure([go.Bar(x=x, y=y)])
    fig.update_layout(title=titel,
                      xaxis_title=xlable,
                      yaxis_title=ylable)
    fig.show()
```

8.7 mathHelper.py

```
#lists all the item no in both sets
def setDiff(x, y):
    return (x - y) | (y - x)

def setPrint(x):
    for i in x:
        print(i)

def dictPrint(x):
    for k in x:
        print(x[k])

def norm(xs, percet=True):
    s = sum(xs)
    if percet:
        s /= 100

    for i, _ in enumerate(xs):
        xs[i] /= s
    return xs

def mean(l):
    return sum(l) / len(l)
```

8.8 generalStatistics.py

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:


import itertools as itt
import classes as vp
import parser
import simulator as sim
import plotly.graph_objects as go
import plot
import mathHelper as mh


# In[3]:


leg_49 = set(range(20))
leg_50 = set(range(20,39))

#load voting data in memory
sessions = parser.loadSessions()
data = parser.parseFiles(sessions, toParse=leg_50)
data_49 = parser.parseFiles(sessions, toParse=leg_49)
factions_size_49 = parser.loadFactionsSize(period='49')
factions_size_50 = parser.loadFactionsSize(period='50')


# In[18]:


```

```python
def showVoteDist(votes):
    #histogramm of how many members votes yes, no, abstain per vote
    present = []
    absent = []
    for v in votes:
        s = len(v.yes) + len(v.no) + len(v.abstain)
        present.append(s)
        absent.append(len(v.novote))

    fig = go.Figure(data=[go.Histogram(x=present, histnorm='percent', marker_color='rgb(59,98,170)')])
    fig.update_layout(#title='Partisipation of National Council Members in 50th legislature',
                   xaxis_title='Number of Members Present for a Vote',
                   yaxis_title='Occurrence in %', height=500, width=900)
    fig.show()

    fig = go.Figure(data=[go.Histogram(x=absent, histnorm='percent')])
    fig.update_layout(#title='Partisipation of National Council Members in 50th legislature',
                   xaxis_title='Number of Members Absent for a Vote',
                   yaxis_title='Occurrence in %', height=500, width=900)
    fig.show()
showVoteDist(data.votes)


# In[6]:


def memberPartisipation(data):
    partisipation = dict()

    def update(key, i, d=partisipation):

        key = vp.faction_names[data.members[key].faction]

        if not key in d:
            d[key] = [0,0,0,0]
        else:
            d[key][i] += 1

    def presents(x):
        return sum(x[0:3]) / sum(x)


    for v in data.votes:
        for mem in v.yes:
            update(mem, 0)
        for mem in v.no:
            update(mem, 1)
        for mem in v.abstain:
            update(mem, 2)
        for mem in v.novote:
            update(mem, 3)

    plot.bar_dict(partisipation, value=lambda x: presents(x) * 100, relativ= True, sort= True)
memberPartisipation(data)


# In[7]:


# creates bar chart for party unity in votes
def factionUnity(data):
    factionUnity = dict()
    for f in data.factionList:
        factionUnity[vp.faction_names[f]] = []

    for v in data.votes:
        for f in data.factionList:
            factionUnity[vp.faction_names[f]].append(v.unity(data.factionList[f], abstain=True))
    plot.bar_dict(factionUnity, value= lambda x: sum(x) / len(x) * 100, relativ= False, sort=True)
factionUnity(data)


# In[ ]:
```

8.9 winningCoalitions.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import itertools as itt
import classes as vp
import PBI
import parser
import simulator as sim
import plotly.graph_objects as go
import plot
import mathHelper as mh


# In[2]:


leg_49 = set(range(20))
leg_50 = set(range(20,39))

#load voting data in memory
sessions = parser.loadSessions()
data_50 = parser.parseFiles(sessions, toParse=leg_50)
data_49 = parser.parseFiles(sessions, toParse=leg_49)
factions_size_49 = parser.loadFactionsSize(period=’49’)
factions_size_50 = parser.loadFactionsSize(period=’50’)
```

```python

#creating simulated voting data
simData_49 = sim.genData(factions_size_49, 5000)
simData_50 = sim.genData(factions_size_50, 5000)


# In[3]:


# find the wining coelition for all the votes
# needs member dict
def allWinningC(data, ids=False):
    coelitions = dict()

    for i in range(1, len(vp.party_names) + 1):
        for p in itt.combinations(vp.party_names, i):
            key = list(p)
            key.sort(key=lambda x: vp.party_order[x])
            if ids:
                coelitions[', '.join(key)] = []
            else:
                coelitions[','.join(key)] = 0

    for i, v in enumerate(data.votes):
        try:
            #change to lists vot ids instead
            if ids:
                coelitions[', '.join(winningC(data, v))].append(i)
            else:
                coelitions[','.join(winningC(data, v))] += 1
        except KeyError:
            print(v)
            print(i)

    return coelitions


#returns winning coelition for a vote
def winningC(data, vote):
    profiles = dict()
    for f in vp.party_names:
        profiles[f] = vp.VoteProfile()

    for mem in vote.yes:
        f = data.members[mem].faction
        profiles[vp.faction_names.get(f,f)].yes += 1

    for mem in vote.no:
        f = data.members[mem].faction
        profiles[vp.faction_names.get(f,f)].no += 1

    for mem in vote.abstain:
        f = data.members[mem].faction
        profiles[vp.faction_names.get(f,f)].abstain += 1

    for mem in vote.novote:
        f = data.members[mem].faction
        profiles[vp.faction_names.get(f,f)].novote += 1

    out = list()

    for p in profiles:
        if profiles[p].partyVote(abstain=False) == vote.decision:
            out.append(p)

    out.sort(key=lambda x: vp.party_order[x])
    return out




# In[20]:


#displays a histogram of all winning coelitions
def coelitions(data, simpleName = False, normalize=False, cut=0, ele=0, order=True):
    l = list(filter(lambda x: x[1] > cut, allWinningC(data).items()))
    if order:
        l.sort(key=lambda x: x[1], reverse=True)

    x = []
    y = []

    for i in l:
        if simpleName and (i[0] in coelition_names):
            x.append(coelition_names[i[0]])
        else:
            x.append(i[0])
        y.append(i[1])

    if normalize:
        #normalize data
        mh.norm(y, percet=True)

    print(sum(y))
    print(len(y))

    #plot.hist(y)

    if ele != 0:
        x = x[:ele]
        y = y[:ele]

    #print(y[:40])
    print(sum(y[:ele]))



    fig = go.Figure([go.Bar(x=x, y=y, marker_color='rgb(59,98,170)')])
    fig.update_layout(#title='Average High and Low Temperatures in New York',
                   #xaxis_title='winning Coalitions',
                   yaxis_title='Occurrence in %',
                    width=1200, height=700)
    fig.show()

#simpler names for coelitions
coelition_names = {'SP,GSP,GLP,CVP,BDP,FDP': 'against SVP',
                  'GLP,CVP,BDP,FDP,SVP': 'against left',
                  'SP,GSP,GLP,CVP,BDP,FDP,SVP': 'unanimous',
                  'CVP,BDP,FDP,SVP': 'center right',
                  'SP,GSP,GLP,CVP,BDP': 'center left',
                  'CVP, FDP, SVP': 'right and CVP',
                  'BDP, FDP, SVP': 'right and BDP',
                  'SP, GSP, GLP, BDP, FDP': 'center left'}


# In[21]:


coelitions(data_50, normalize=True, cut=1, ele=40, simpleName=False)


# In[22]:


coelitions(simData_50, normalize=True, cut=1, ele=64, simpleName=False, order=False)


# In[13]:


coelitions(data_49, normalize=True, cut=1, ele=40, simpleName=False)


# In[17]:


coelitions(simData_49, normalize=True, cut=1, ele=64, simpleName=False, order=False)


# In[ ]:
```

8.10 voting_power.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import csv
import os
import itertools as itt
import classes as vp
import PBI
```

```python
import parser
import simulator as sim
import plotly.graph_objects as go
import plot
import mathHelper as mh


# In[2]:


#impirical voting power
#count how often a party in pivotal
#uses votes
def impVotingPower1(data):
    pivotal = dict()
    for f in data.factionList:
        count = 0
        for v in data.votes:
            if v.isPivotal(data.factionList[f]):
                count += 1
        pivotal[vp.faction_names.get(f,f)] = count

    n = sum(pivotal.values())

    for p in pivotal:
        pivotal[p] /= n

    return pivotal


# In[3]:


def profie(vote):
    profiles = dict()
    for f in vp.party_names:
        profiles[f] = vp.VoteProfile()

    for mem in vote.yes:
        profiles[vp.faction_names[members[mem].faction]].yes += 1

    for mem in vote.no:
        profiles[vp.faction_names[members[mem].faction]].no += 1

    for mem in vote.abstain:
        profiles[vp.faction_names[members[mem].faction]].abstain += 1

    for mem in vote.novote:
        profiles[vp.faction_names[members[mem].faction]].novote += 1

    for p in profiles:
        print(p + ' ' + str(profiles[p]))


# In[4]:

```

```

leg_49 = set(range(20))
leg_50 = set(range(20,39))

#load voting data in memory
sessions = parser.loadSessions()
data_50 = parser.parseFiles(sessions, toParse=leg_50)
data_49 = parser.parseFiles(sessions, toParse=leg_49)
factions_size_49 = parser.loadFactionsSize(period=’49’)
factions_size_50 = parser.loadFactionsSize(period=’50’)

#creating simulated voting data
simData_49 = sim.genData(factions_size_49, 5000)
simData_50 = sim.genData(factions_size_50, 5000)


# In[26]:


def groupedBar_Dict(dicts, names, log=False):
    data = []
    color = [’rgb(59,170,98)’,’rgb(59,98,170)’,’rgb(170,59,98)’][1:]

    for k, d in enumerate(dicts):
        l = []
        for i in d:
            l.append(d[i] * 100)

        #for i,item in enumerate(l):
         #   l[i] /= sum(l)
        #norm(l)
        data.append(go.Bar(name=names[k], x=list(d.keys()), y=l, marker_color=color[k]))

    fig = go.Figure(data=data)
    # ChagroupedBar_Dict([impVotingPower1(simData.votes, simData.factionList), impVotingPower1(data.votes, data.factionList), PBI.votingPower(factions_size_49)], [’sim’,’imp’,’50’])nge the bar mode
    if log:
        fig.update_layout(barmode=’group’, yaxis_type=’log’)
    else:
        fig.update_layout(barmode=’group’, width=900, height=500)
    fig.show()



# In[27]:


groupedBar_Dict([impVotingPower1(data_50), PBI.votingPower(factions_size_50)], [’Empirical Data’, ’Banzhaf Index’])


# In[23]:


```

```
groupedBar_Dict([impVotingPower1(simData_50), impVotingPower1(data_50), PBI.votingPower(factions_size_50)], ['Simulated Data', 'Empirical Data', 'Banzhaf Index'])


# In[13]:


groupedBar_Dict([impVotingPower1(simData_49), impVotingPower1(data_49), PBI.votingPower(factions_size_49)], ['simulated data','empirical data','Banzhaf Index'])


# In[ ]:
```

References

1. plotly documentation, https://plot.ly/python/
2. Wie wir bei srf (neu) parteien einfrben, https://medium.com/srf-schweizer-radio-und-fernsehen/wie-wir-bei-srf-parteien-einf\%C3\%A4rben-9f010f80cf62
3. über das parlament (January 28 , 2020), https://www.parlament.ch/de/\%c3\%bcber-das-parlament-home
4. Abstimmung datenbank (November 30, 2019), https://www.parlament.ch/de/ratsbetrieb/abstimmungen/abstimmung-nr-xls
5. Die fraktionen (November 30, 2019), https://www.parlament.ch/de/organe/fraktionen
6. "Banzhaf, J.F.: Weighted voting doesn't work: A mathematical analysis. Rutgers Law Review **19**, 317–343 (1963)
7. Steiner, D.: Python code for report https://github.com/partun/mpl_voting_power