Task 1

a)

Set a suitable threshold for each measure (look at 5–8 most central nodes):

- For the following a threshold of 8 most central nodes for visualization is used

Identify most central and influential nodes with different measures:

- node degree:

| Label | Degree |
|---|---|
| 1477 | 43 |
| 1443 | 43 |
| 1457 | 42 |
| 1502 | 42 |
| 1563 | 41 |
| 1452 | 41 |
| 1428 | 41 |
| 1458 | 40 |

- weighed degree:

| Label | Weighted Degree |
|---|---|
| 1437 | 221.000003 |
| 1563 | 216.800002 |
| 1457 | 186.800001 |
| 1458 | 183.799999 |
| 1452 | 171.800002 |
| 1477 | 165.2 |
| 1498 | 164.599999 |
| 1480 | 161.8 |

- closeness centrality:

| Id | Closeness Centrality |
|---|---|
| 1443 | 0.957447 |
| 1477 | 0.957447 |
| 1457 | 0.9375 |
| 1502 | 0.9375 |
| 1428 | 0.918367 |
| 1452 | 0.918367 |
| 1563 | 0.918367 |
| 1426 | 0.9 |

- betweenness centrality:

- | Id | Betweenness Centrality |
  | --- | --- |
  | 1443 | 10.267852 |
  | 1477 | 9.288583 |
  | 1502 | 8.774473 |
  | 1457 | 8.27553 |
  | 1563 | 8.193991 |
  | 1480 | 7.875956 |
  | 1522 | 7.730772 |
  | 1585 | 7.57164 |

What do these measures tell about nodes?

- node degree:
    - The degree of a node tells us the number of edges connected to the node.

- weighed degree:
    - The weighted node degree is the sum of the edge weights for edges incident to that node.

- closeness centrality:
    - **Closeness centrality**: $C_C(v) = \frac{1}{avg_{u \in V}\{Dist(v,u)\}}$
    - The closeness centrality of a node measures its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances to all other nodes. Therefore, the more central a node is, the closer it is to all other nodes.

- betweenness centrality:
    - **Betweenness centrality**: $C_B(v) = \frac{\sum_{u,w \in V, u \neq w} \frac{\#\{\text{shortest-paths}(u,w) \text{ through } v\}}{\#\{\text{shortest-paths}(u,w)\}}}{\binom{n}{2}}$
    - Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes. Betweenness centrality measures the extent to which a vertex lies on paths between other vertices. Vertices with high betweenness may have considerable influence within a network. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of the weights of the edges (for weighted graphs) is minimized. The betweenness centrality for each vertex is the number of these shortest paths that pass through the vertex.

b)

Definition of community measures:

- Modularity:
    - Modularity measures the strength of division of a network into clusters. Networks with high modularity have dense connections between the nodes

within the same cluster but sparse connections between nodes in different clusters. Modularity reflects the concentration of edges within modules compared with random distribution of links between all nodes regardless of modules.

- Girvan-Newman clustering:
    - Divisive hierarchical clustering based on edge betweenness. Number of shortest paths passing through the edge. The algorithm removes the “most valuable” edge, traditionally the edge with the highest betweenness centrality, at each step.

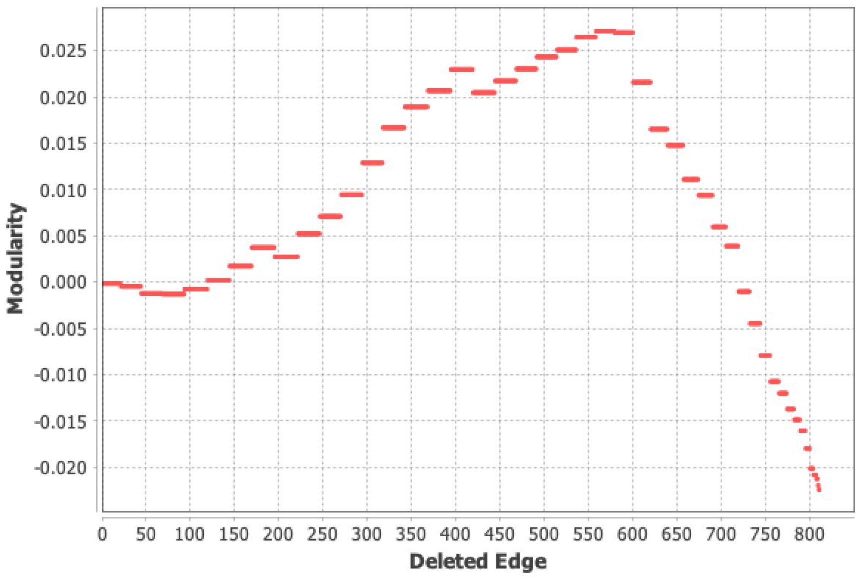
Identify communities:

- **Processed Graph Data**

  Nodes: 46

  Edges 809
- Girvan-Newman clustering:
    - **Communities**

      Number of communities: 24

      Maximum found modularity: 0.027242184
    - 

    - We can see that the Girvan-Newman algorithm is not that well suited for the given network. The network is very dense and therefore well connected. The algorithm needs to delete a lot of edges, here 600 out 800, to get the maximum found modularity of 0.027. That score is still very low. Furthermore, 24 communities are detecting while having only 46 nodes in the graph. That would mean around two nodes per cluster which is a bad community detection.

- Modularity:
    - For this algorithm we can fine tune the resolution by running the algorithm with different resolutions.

| Resolution | Number of communities | Modularity | Modularity with resolution |
|---|---|---|---|
| 0 | 46 | -0.026 | -0.026 |
| 0.2 | 17 | 0.207 | -0.010 |
| 0.4 | 10 | 0.309 | 0.056 |
| 0.6 | 7 | 0.341 | 0.144 |
| 0.8 | 5 | 0.371 | 0.251 |
| 1.0 | 4 | 0.379 | 0.379 |

    - For resolution = 0.8 we can see that modularity and modularity with resolution converged already very closely. We can assume that for modularity 4-5 clusters are reasonable results.

Compare results (similarities and differences):

- Done above.

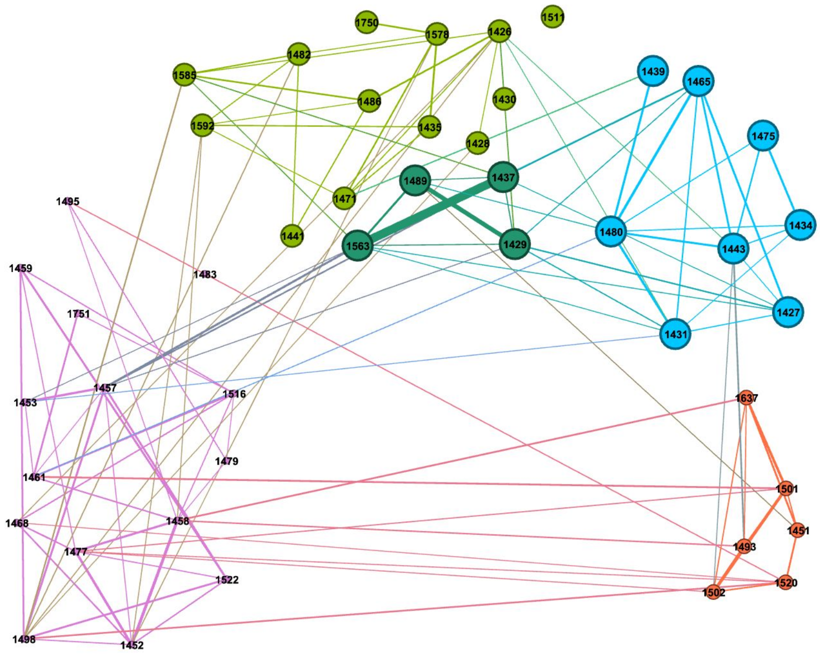
Does the background information (class and gender) explain communities?

- Yes, the background information explains communities. Since the community results for Girvan-Newmann are not really good (approx. 24 communities), I analyzed the background information for modularity. We have about 3-4 communities. So I picked out the nodes of each cluster separately and double-checked with metadata from "t1_schoolclass5meta.txt". And turned out that the background information explain communities. For example, in one cluster mainly female, and then also divided by class. However, I didn't figure out how to import the metadata into gephi. Because importing another txt/csv result in creating a new graph. Therefore, I needed to it separately manually on paper

c)

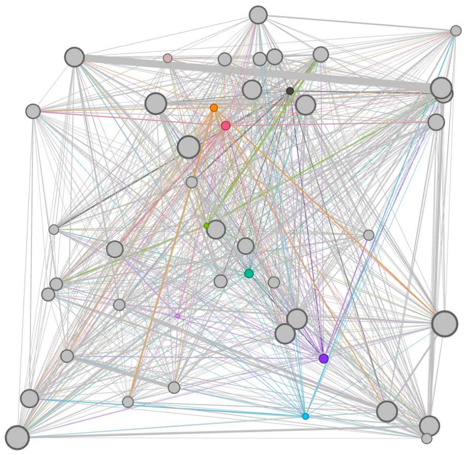
Present the communities visually

- Task:
    - Use distinct colours to show nodes, add labels, hide low weight edges to simplify the graph, move nodes so that communities become separated.
    - Take snapshots of both Modularity and Girvan-Newman results.
- Modularity:




- 
- each class has its own color
- nodes were dragged to separate from other cluster and closer to own cluster
- size of the node represents the modularity
- weights < 5.8 are filtered out

- Girvan-Newman Algorithm



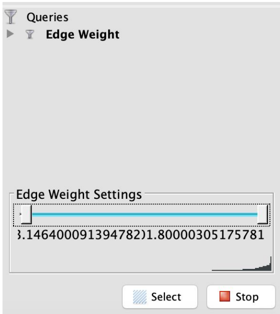
  - 
  - I could give every cluster a color. But that's unnecessary work because we can't conclude much out of it. Since anyways each cluster only contains 2 nodes. Maybe more interesting is to differentiate the size of nodes by clustering coefficient. However, also doesn't tell us much.

d)

Modularity function for community detection:

Hiding low weight edges with different thresholds and analyze links inside and between communities:

- Gephi has the nice option to filter by edge weights:




- By moving the bar slowly to the right and increasing the threshold you can observe the following. First of all, the intra edges - edges between different clusters - tend to disappear more frequent than the inter edges - edges within the cluster.

Which communities have strongest interconnections?

- The light green cluster is definitely the cluster with the weakest interconnections. These disappear the first.

What are bridge nodes that combine two communities (end points of strong links between communities)?

- Deletion of bridge nodes increases the graph's number of connected components. It is not contained in any cycle. For a connected graph, a bridge can uniquely determine a cut. Followed by blue, then purple, afterwards orange. See the colors in the screenshot above. The strongest links are of the dark green cluster with node IDs: 1429, 1437, 1489, 1563

Are these the same as central nodes? Or what is the role of central nodes in communities?

- Centrality identifies the most important vertices within a graph. Which node are the most influential nodes to other nodes in the graph. Deleting that node doesn't necessarily result in more clusters or disconnects the graph.

# Import

In [73]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import nltk
import random

from nltk.corpus import movie_reviews
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.corpus import wordnet as wn
from nltk.classify import SklearnClassifier
from nltk.corpus import wordnet as wn

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC

import networkx as nx
```

# Task 1

Task 1 has been done on an external software Gephi, and therefore I wrote the report on a word document. The report has been merged to the beginning of this pdf.

# Task 2

In [2]:

```
# Import data set
number_nodes = 62

df = pd.read_csv('t2_dolphins.txt', sep="\s+", header=None, names=["n1", "n2"]) #
print("data shape: ", df.shape)
df.head()
```

```
data shape:  (159, 2)
```

Out[2]:

| | n1 | n2 |
|---|---|---|
| 0 | 11 | 1 |
| 1 | 15 | 1 |
| 2 | 16 | 1 |
| 3 | 41 | 1 |
| 4 | 43 | 1 |

## Task 2 a)

**Questions** * In this task only undirected graphs are considered, since the professors confirmed that only undirected graphs need to be analyzed. * For undirected graphs, the notion of density of the subgraph is the average degree of the subgraph. * First thought: * Search node with highest degree * of all neighbors search node with highest degree and add it to the subgraphs * out of all current nodes in the subgraph find neighbor with highest degree and add it to the subgraph * However, the paper 'Greedy Approximation Algortihms for Finding Dense Components in a Graph' by Moses Charikar analyzes another algortihm that is more efficent and promises better results: * Given a number a set of vertices that build a graph, remove in every step the vertice with lowest degree * Calculate in each step the density and store for each step the density and vertices of used subgraph * Do that till all nodes are deleted * return subgraph with highest density

## Task 2 b)

In [41]:

```
def calculate_density(subgraph):
    """ Calculate density of subgraph according to formula given in task descriptio
n
    """
    number_edges = subgraph.shape[0]
    number_nodes = len(np.unique(subgraph))

    density = number_edges / number_nodes

    return density

def greedy_dense_subgraph(graph):
    """ Greedy algorithm to find a subgraph with good density
    """
    number_nodes = len(np.unique(graph))
    current_subgraph = graph
    subgraph_density_history = []  # keep track of subraphs and their density score
    density_score_history = []
    number_edges_history = []

    while (number_nodes > 0):

        # calculate density
        density_score = calculate_density(current_subgraph)
        subgraph_density_history.append((density_score, current_subgraph))
        density_score_history.append(density_score)
        number_edges_history.append(current_subgraph.shape[0])

        # delete node with lowest degree
        nodes, counts = np.unique(current_subgraph, return_counts=True)
        idx_lowest_degree = np.argmin(counts)
        node_lowest_degree = nodes[idx_lowest_degree]

        new_subgraph = []
        for edge in current_subgraph:  # loop through all edges
            if (node_lowest_degree not in edge):
                new_subgraph.append(edge)


        current_subgraph = np.array(new_subgraph)
        number_nodes = len(np.unique(current_subgraph))

    # identify subgraph with highest density
    densities = [density for (density, subgraph) in subgraph_density_history]
    #print("densities", densities)
    max_ids = np.argmax(densities)
    (max_density_score, subgraph) = subgraph_density_history[max_ids]


    return max_density_score, subgraph, density_score_history, number_edges_history
```

In [42]:

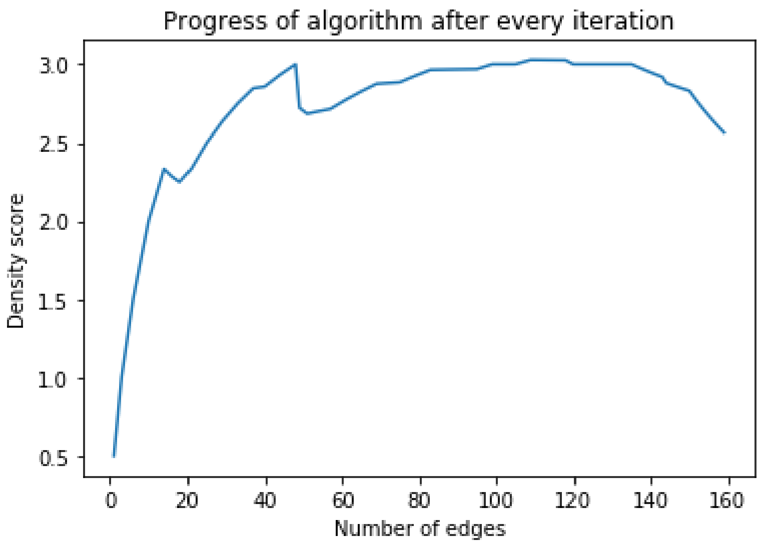
```
graph = df.to_numpy()

density_score, subgraph, density_score_history, number_edges_history = greedy_dense
_subgraph(graph)
print("Highest density score: ", density_score)
print("Number of edges: ", subgraph.shape[0])
print("Number of nodes: ", len(np.unique(subgraph)))
```

```
Highest density score:  3.0277777777777777
Number of edges:  109
Number of nodes:  36
```

In [48]:

```
plt.plot(number_edges_history, density_score_history)
plt.title("Progress of algorithm after every iteration")
plt.xlabel("Number of edges")
plt.ylabel("Density score")
plt.show()
#plt.plot(range(len(density_score)),density_score)
```




**Questions** * The given dolphins dataset is undirected and unweighted, and the implemented algorithm is adjusted to that

# Task 3

Since we only needed to do 4 out of 5 tasks, I decided to leave out Task 3.

# Task 4

In [74]:

```
# load datasets

df_train = pd.read_csv("t4_jester-800-10.csv")
df_test = pd.read_csv("t4_test-800-10.csv")

df_train.head()
```

Out[74]:

| | user_id | joke_5 | joke_7 | joke_8 | joke_13 | joke_15 | joke_16 | joke_17 | joke_18 | joke_19 | joke_20 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 5013 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 10016 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 2 | 21844 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 3403 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 4 | 23240 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

In [76]:

```
user_ids = df_train["user_id"].values
joke_names = df_train.columns.values
```

In [80]:

```
# Create bi-partite graph
B = nx.Graph()
# Add nodes with the node attribute "bipartite"
B.add_nodes_from(user_ids, bipartite=0)
B.add_nodes_from(joke_names, bipartite=1)
# Add edges only between nodes of opposite node sets

set_of_edges = []

for user in user_ids:
    row = df_train.loc[df_train['user_id'] == user]
    for joke in joke_names:
        like = int(row[joke])
        if (like == 1):  # add edge to the graph
            set_of_edges.append((user, joke))

B.add_edges_from(set_of_edges)
```

In [ ]:

```
# calculate simrank, everyone with everyone
#sim = nx.simrank_similarity(B)
```

In [ ]:

```
users_sorted = sorted(user_ids)

lol = [[sim[u][v] for v in users_sorted] for u in users_sorted]
sim_array = np.array(lol)
sim_array
```

Explanation:

Unfortunately, I couldn't finish this task because I underestimated the computation time of the simrank algorithm. Only a few steps are missing. So when the calculating of the SimRank finishes, we get a dictionary that contains for each node the similarity score to every other node. Therefore there are dictionaries within a dictionary. We get an n x n matrix. Then we search for the test nodes. For each test node get the highest value in each row which is not the same node. So we can set the diagonal to zero to make sure this doesn't happen. With np.argmax() we the index with the node of the highest similarity score. Then we check which jokes the most similar user liked that haven't been liked so far from the test user and recommend these jokes to the test user.

**Questions** * We use collaborative filtering, because we want to give recommendations based on similar users they have watched, what they liked and therefore what could be recommended to the other person. * Two users are similar if they watched and liked similar movies. Hence, it is very likely that a movie liked by User 1 but not yet watched by User 2, User 2 will also very likely will enjoy. * Therefore, the assumption we use: The user probably likes what other similar users have liked.

# Task 5

## Task 5 a)

**Without preprocessing**

Data collection

In [5]:

```
# get movie reviews data
documents = [(list(movie_reviews.words(fileid)), category)
              for category in movie_reviews.categories()
              for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
#print(documents[:1])
#documents = np.array(documents)
```

Data preprocessing

- Remove stopwords
- perform stemming

In [6]:

```
# calculate frequency of words
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_words)[:2000]  # first 2000 most frequent words; ordered b
y frequency
print(word_features[:5])  # first five most frequent words


def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# Feature extraction and training:
featuresets = [(document_features(d), c) for (d,c) in documents] # c: positive or n
egative, d: words
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
#classifier = SklearnClassifier(SVC(), sparse=False).train(train_set)

# Use of classifier:

print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(5)

print()
print()
```

```
['plot', ':', 'two', 'teen', 'couples']
0.79
Most Informative Features
       contains(stellan) = True              pos : neg    =      8.2 :
1.0
 contains(unimaginative) = True              neg : pos    =      7.8 :
1.0
    contains(schumacher) = True              neg : pos    =      7.5 :
1.0
     contains(atrocious) = True              neg : pos    =      6.7 :
1.0
        contains(turkey) = True              neg : pos    =      6.7 :
1.0
```

**With preprocessing**

In [7]:

```
# get movie reviews data
documents = [(list(movie_reviews.words(fileid)), category)
              for category in movie_reviews.categories()
              for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
#print(documents[:1])
#documents = np.array(documents)
```

Data preprocessing

- Remove stopwords
- perform stemming

In [8]:

```
# stemming
porter = PorterStemmer()
stemmed_documents = []
for (d,c) in documents:
    stemmed_documents.append(([porter.stem(j) for j in d], c))
print('After stemming:')
```

```
After stemming:
```

In [9]:

```
# stop words removal and punctation removal
stop_words = set(stopwords.words('english'))
# We include the punctation in the stop words set.
punctation = set("!”#$%&’()*+,-./:;<=>?@[\]^_`{|}~\"\'")
stop_words.update(punctation)
stop_words.add("...")

#print('Stop words that will get removed:')
#print(stop_words)
#print()

filtered_documents = []
for (d,c) in stemmed_documents:
    filtered_documents.append(([word for word in d if word not in stop_words], c))
```

In [10]:

```
# calculate frequency of words
all_words_documents = []
for (d,c) in filtered_documents:
    for word in d:
        all_words_documents.append(word)

all_words = nltk.FreqDist(w.lower() for w in all_words_documents)
word_features = list(all_words)[:2000]  # first 2000 most frequent words; ordered b
y frequency
print(word_features[:5])  # first five most frequent words


def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# Feature extraction and training:
featuresets = [(document_features(d), c) for (d,c) in filtered_documents] # c: posi
tive or negative, d: words
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
#classifier = SklearnClassifier(SVC(), sparse=False).train(train_set)

# Use of classifier:

print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(5)
```

```
['robin', 'william', 'comed', 'genu', 'one']
0.8
Most Informative Features
      contains(outstand) = True              pos : neg    =     13.4 :
1.0
          contains(plod) = True              neg : pos    =     13.1 :
1.0
      contains(furnitur) = True              neg : pos    =      7.0 :
1.0
       contains(sputter) = True              neg : pos    =      7.0 :
1.0
  contains(breakthrough) = True              pos : neg    =      7.0 :
1.0
```

**Questions** Results without preprocessing: * Naive Bayes Classifier: 0.77 * Support Vector Classifier: 0.77 Results with preprocessing (stemming and removal of stopwords): * Naive Bayes Classifier: 0.78 * Support Vector Classifier: 0.8 Results and Interpretation: * The results get slightly better * Only important features were put into considertion * Deletion of stopwards and punctuation

## Task 5 b)

### Replace words by hypernym

In [11]:

```
# get movie reviews data
documents = [(list(movie_reviews.words(fileid)), category)
              for category in movie_reviews.categories()
              for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
#print(documents[:1])
#documents = np.array(documents)
```

Data preprocessing

- Remove stopwords

In [12]:

```
# stop words removal and punctation removal
stop_words = set(stopwords.words('english'))
# We include the punctation in the stop words set.
punctation = set("!”#$%&’()*+,-./:;<=>?@[\]^_`{|}~\"\'")
stop_words.update(punctation)
stop_words.add("...")

filtered_documents = []
for (d,c) in documents:
    filtered_documents.append(([word for word in d if word not in stop_words], c))
```

In [13]:

```
# Replace words with hypernym
hypernym_documents = []
for (d,c) in filtered_documents:
    hypernym_array = []
    for word in d:

        synsets = wn.synsets(word)  # get all sysnsets
        if (synsets == []):
            name = word
        else:
            first_synset = synsets[0]  # choose first synset
            hypernyms = first_synset.hypernyms()  # get all hypernyms of first syns
et
            if (hypernyms == []):
                name = word
            else:
                hypernym = hypernyms[0]  # choose first hyernym
                lemmas = hypernym.lemmas()  # get all lemmas
                lemma = lemmas[0]  # choose first lemma

                name = lemma.name()  # get name of lemma


        hypernym_array.append(name)

    hypernym_documents.append( (hypernym_array, c) )
```

Feature engineering

In [14]:

```
# calculate frequency of words
all_words_documents = []
for (d,c) in hypernym_documents:
    for word in d:
        all_words_documents.append(word)

all_words = nltk.FreqDist(w.lower() for w in all_words_documents)
word_features = list(all_words)[:2000]  # first 2000 most frequent words; ordered b
y frequency
print(word_features[:5])  # first five most frequent words


def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# Feature extraction and training:
featuresets = [(document_features(d), c) for (d,c) in hypernym_documents] # c: posi
tive or negative, d: words
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
#classifier = SklearnClassifier(SVC(), sparse=False).train(train_set)

# Use of classifier:

print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(5)
```

```
['steve', 'martin', 'act', 'increase', 'leisure']
0.76
Most Informative Features
     contains(ludicrous) = True              neg : pos    =     15.2 :
1.0
   contains(outstanding) = True              pos : neg    =     11.1 :
1.0
    contains(incoherent) = True              neg : pos    =      9.1 :
1.0
        contains(feeble) = True              neg : pos    =      7.7 :
1.0
      contains(believer) = True              pos : neg    =      6.9 :
1.0
```

**Questions** Results: * Naive Bayes Classifier: 0.81 * Support Vector Classifier: 0.8 Results and Interpretation: * For task b) no stemming, because wordnet synset and hypernyms work worse on stemmed words. For instance, wordnet works well on wn.synsets('happiest') or wn.synsets('happier'), but not on stemmed word of happy which is wn.synsets('happi'). * The results improved again slightly in the case of Naive Bayes Classifier to 0.81 * makes absolutely sense since we replace words with similar meaning to the same word

### Add hypernyms of words

In [15]:

```
# get movie reviews data
documents = [(list(movie_reviews.words(fileid)), category)
              for category in movie_reviews.categories()
              for fileid in movie_reviews.fileids(category)]
random.shuffle(documents)
#print(documents[:1])
#documents = np.array(documents)
```

Data preprocessing

- Remove stopwords

In [16]:

```
# stop words removal and punctation removal
stop_words = set(stopwords.words('english'))
# We include the punctation in the stop words set.
punctation = set("!”#$%&’()*+,-./:;<=>?@[\]^_`{|}~\"\'")
stop_words.update(punctation)
stop_words.add("...")

filtered_documents = []
for (d,c) in documents:
    filtered_documents.append(([word for word in d if word not in stop_words], c))
```

In [17]:

```
# Replace words with hypernym
hypernym_documents = []
for (d,c) in filtered_documents:
    hypernym_array = []
    for word in d:
        hypernym_array.append(word)

        synsets = wn.synsets(word)  # get all sysnsets
        if (synsets == []):
            continue
        else:
            first_synset = synsets[0]  # choose first synset
            hypernyms = first_synset.hypernyms()  # get all hypernyms of first syns
et
            if (hypernyms == []):
                continue
            else:
                hypernym = hypernyms[0]  # choose first hyernym
                lemmas = hypernym.lemmas()  # get all lemmas
                lemma = lemmas[0]  # choose first lemma
                name = lemma.name()  # get name of lemma

                hypernym_array.append(name)

    hypernym_documents.append( (hypernym_array, c) )
```

Feature engineering

In [18]:

```
# calculate frequency of words
all_words_documents = []
for (d,c) in hypernym_documents:
    for word in d:
        all_words_documents.append(word)

all_words = nltk.FreqDist(w.lower() for w in all_words_documents)
word_features = list(all_words)[:2000]  # first 2000 most frequent words; ordered b
y frequency
print(word_features[:5])  # first five most frequent words


def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# Feature extraction and training:
featuresets = [(document_features(d), c) for (d,c) in hypernym_documents] # c: posi
tive or negative, d: words
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
#classifier = SklearnClassifier(SVC(), sparse=False).train(train_set)

# Use of classifier:

print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(5)
```

```
['tempe', 'mills', 'cinema', 'medium', 'az']
0.72
Most Informative Features
      contains(bothered) = True              neg : pos    =      9.7 :
1.0
     contains(strongest) = True              pos : neg    =      9.0 :
1.0
       contains(winslet) = True              pos : neg    =      7.7 :
1.0
    contains(cronenberg) = True              pos : neg    =      7.0 :
1.0
          contains(gump) = True              pos : neg    =      7.0 :
1.0
```

**Questions** Results: * Naive Bayes Classifier: 0.73 * Support Vector Classifier: 0.76 Results and Interpretation: * For task b) no stemming, because wordnet synset and hypernyms work worse on stemmed words. For instance, wordnet works well on wn.synsets('happiest') or wn.synsets('happier'), but not on stemmed word of happy which is wn.synsets('happi'). * The results improved got worse for both classifiers while the scores dropped to 0.73 and 0.76. * That makes kind of sense, because not all words have hypernyms. So imagine for instance not important words have hypernyms while important words don't have hypernyms. So we add the hypernyms of not important words. Consequently, not important words are more frequent and have more weight. That results in a worse classification score. Therefore, it is better to replace the words by its hypernym.

---

In [ ]:

```

```