1. Scenario Grouping ¶

Clearly, we needed a more scientific way to group the scenarios. We started by retrieving some information about the model's misclassifications.

In [ ]:
```
import pandas as pd
import numpy as np
import networkx as nx
from graph import *
from SimAnn import SimAnnProbl, simann
df = pd.read_csv("LSVC.csv")
df = df.drop("index", axis = 1)
nrows = df.shape[0]

accuracies = {}

##### ACCURACIES FOR EACH SCENARIO
for context in set(df["label"]):
    correct = len(df[df["correct"]==True][df["label"]==context])
    total = len(df[df["label"]==context])
    accuracies[context] = correct/total


#### CREATES A DICTIONARY WHERE EACH SCENARIO IS ASSOCIATED WITH A
TUPLE CONTAINING
#### THE WRONG SCENARIO THAT THE MODEL PREDICTED AND THE PERCENTAGE
OF QUESTIONS OF THAT SCENARIO
#### WRONGLY PREDICTED WITH SAID WRONG SCENARIO (ONLY DOES IT IF TH
IS PERCENTAGE IS > 2%)
common_misclassifications = {}
for context in set(df["label"]):
    errors = df[df["label"]==context][df["correct"]==False]
    common_misclassifications[context] = list(filter(
        lambda x: x[1]>0.02,sorted([(sc,
        np.sum(errors["prediction"]==sc)/len(df[df["label"]==contex
t]))
        for sc in set(df["label"])], key = lambda x: x[1], reverse
= True)))

def explain():
    print("Common Misclassifications")
    print()
    for context in set(df["label"]):
        print(f"--- CONTEXT: {context} ---")
        for c in common_misclassifications[context]:
            print(f"{c[0]}: {c[1]}")
        print()
```

This information is certaintly useful, but now we need an algorithm capbable of using this information to construct an optimal grouping for the scenarios. Too many groups will make it hard on the scenario classifier, while too few groups will make it hard for the intent classifier. We started by plotting the misclassifications on a graph. The nodes represent the scenarios, an edge from scenario A to scenario B means that scenario A often gets misclassified as scenario B. The edge weight (not shown here) is the percentage of questions of scenario A wrongly classified as belonging to scenario B.

In [ ]:
```
graph = nx.DiGraph()
for scenario in set(df["label"]):
    for miscls in common_misclassifications[scenario]:
        graph.add_edge(scenario, miscls[0], weight = miscls[1])

def draw():
    nx.draw_networkx(graph)
```

Even though the graph does not seem to convey much additional information, we managed to find a way to use this representation to obtain a better grouping.

Start with the above described graph, our objective is to obtain a second graph by contracting the nodes in such a way as to reflect the outcome of grouping the scenarios on the classification process. We have observed that, when two scenarios A and B are grouped together, the percentage of misclassifications from AB to C decrease, but the percentage of misclassifications from C to AB increases. We have defined an empirical rule for the edges ingoing and outgoing from a contracted node as:

$$w(AB, C) = min(w(A, C), w(B, C))$$
$$w(C, AB) = w(A, C) + w(B, C)$$

We want to perform a series of node contraction in order to minimize a cost function. We have defined the cost function to be:

$$L = \frac{1}{n-k} \sum_{e \in E} w_e$$

Where n is the number of nodes (groups), k is a penalization for an excessive amount of groups, and w_e is the weight of edge e. Basically, we want to minimize the weights in the graph, but at the same time taking into account also the number of groups.

As for the minimization method, we used Monte Carlo Markov Chain optimization through the Simulated Annealing algorithm, where set the kernel to be a random contraction in the graph.

In [ ]:
```
from collections import defaultdict
from SimAnn import SimAnnProbl, simann
import random

class Graph:
    def __init__(self):
        self.vertices = []
        self.edges = defaultdict(dict)

    def add_edge(self, v1: str, v2: str, w: float):
        if v1 not in self.vertices:
            self.vertices.append(v1)
        if v2 not in self.vertices:
            self.vertices.append(v2)
        self.edges[v1][v2] = w

    def merge(self, v1: str, v2: str):
        print(f"merging {v1} and {v2}")
        # remove edges between v1 and v2
        self.edges[v1].pop(v2, 0)
        self.edges[v2].pop(v1, 0)

        # create new vertex
        self.vertices.remove(v1)
        self.vertices.remove(v2)
        new = v1+"/"+v2
        self.vertices.append(new)

        # outgoing edges: w(ab, v) = min(w(a, v), w(b, v))
        for v in self.vertices:
            wav = self.edges[v1].pop(v, float("inf"))
            wbv = self.edges[v2].pop(v, float("inf"))
            if wav != float("inf") and wbv != float("inf"):
                self.edges[new][v] = min(wav, wbv)

        # ingoing edges: w(v, ab) = w(v, a) + w(v, b)
        for v in self.vertices:
            wva = self.edges[v].pop(v1, 0)
            wvb = self.edges[v].pop(v2, 0)
            if wva + wvb != 0:
                self.edges[v][new] = wva + wvb

    def total_cost(self):
        c = 0
        for v1 in self.vertices:
            for v2 in self.vertices:
                c += self.edges[v1].get(v2, 0)
        return c/(len(self.vertices)-2) #cost function with k=2


class MisclassificationProblem(SimAnnProbl):
    def __init__(self, graph: Graph):
        self.graph = graph
    def cost(self):
        return self.graph.total_cost()
    def propose_move(self):
        while True:
            v1 = random.choice(self.graph.vertices)
            if len(self.graph.edges[v1])>0:
                v2 = random.choice(list(self.graph.edges[v1].keys()
))
                return (v1, v2)
    def accept_move(self, move):
        v1, v2 = move
```

```
            self.graph.merge(v1, v2)
    def copy(self):
        g = Graph()
        g.edges = self.graph.edges.copy()
        g.vertices = self.graph.vertices.copy()
        return MisclassificationProblem(g)
    def compute_delta_cost(self, move):
        a, b = move
        n = len(self.graph.vertices)-2
        delta = 0
        delta += self.graph.edges[a].get(b, 0)
        delta += self.graph.edges[b].get(a, 0)
        for v in self.graph.vertices:
            if v != a and v!= b:
                wav = self.graph.edges[a].get(v, float("inf"))
                wbv = self.graph.edges[b].get(v, float("inf"))
                m = max(wav, wbv)
                if m != float("inf"):
                    delta += m
        c = self.graph.total_cost()
        return (c-delta)/(n-1)
```

Now we are ready to use this optimization method to obtain the optimal grouping.

In [ ]:
```
g = Graph()
for scenario in set(df["label"]):
    for miscls in common_misclassifications[scenario]:
        g.add_edge(scenario, miscls[0], miscls[1])

probl = MisclassificationProblem(g)

def run_simann():
    simann(probl, beta0=2.0, beta1=100.0, anneal_steps=10, mcmc_ste
ps=10**3)


# g.vertices contains the optimal grouping
```