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
```

2. Group Classification ¶

After having understood the best subdivision of scenarios into groups by taking the *six most "distant" clusters in the multidimensional space* (in Scenario-Grouping.ipynb), we are ready to train the model to make it classify **questions** into **groups**.

In [1]:
```
import spacy
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import Pipeline, make_pipeline, FeatureUnion
from sklearn.compose import ColumnTransformer
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import Normalizer
```

After having imported the needed libraries we load our train dataframe and the [**spaCy** model (https://spacy.io/models/en#en_core_web_lg)](https://spacy.io/models/en#en_core_web_lg) we will use. We use the *large model* because we will need vectors for *word embedding*.

In [2]:
```
nlp = spacy.load('en_core_web_lg')
train_df = pd.read_csv('dataset_intent_train.csv', sep=';')
```

We create a new column "group", that is going to be our label, by clustering scenarios.

In [ ]:
```
def grouping(df):
    groups = []
    for i in df['scenario']:
        if i in ['weather', 'cooking', 'transport', 'general', 'soc
ial',
                    'news', 'takeaway', 'qa']:
            groups.append('a')
        elif i in ['music', 'audio', 'play']:
            groups.append('b')
        elif i in ['recommendation', 'lists', 'datetime', 'calendar
']:
            groups.append('c')
        elif i == 'alarm':
            groups.append('d')
        elif i == 'iot':
            groups.append('e')
        elif i == 'email':
            groups.append('f')

    df['group'] = groups
    return df

grouping(train_df)
```

We then vectorize the questions, creating a *300 dimensions word embedding*.

In [4]:
```
train_df['vector'] = [nlp(text).vector for text in train_df.questio
n]
```

We define our **X** and **y**.

In [6]:
```
X = train_df[['question', 'vector']]
y = train_df['group']
```

We don't want our "vector" column to be a Series of length 300, but rather to add 300 new columns (**features**).

In [7]:
```
for i, row in X.iterrows():
    for j, vec in enumerate(X.loc[i, 'vector']):
        X.loc[i, f'Vec_{j+1}'] = vec
X = X.drop('vector', axis=1)
```

We define our:

> - Term Frequency - Inverse Document Frequency analyzer: proceding "hunder the hood" through a Bag-of-Words
> - Preprocessor: tfidf on question and normalizing the question-vector dimensions
> - Classifier: Linear Support Vector Classifier

In [8]:
```
tfidf = TfidfVectorizer(ngram_range=(1, 2))
preproc = ColumnTransformer([('tfidf', tfidf, 'question'),
                             ('scaler', Normalizer(), [i for i in X
.columns[1:]])])
lsvc = LinearSVC(C=1.7, loss='hinge', max_iter=10000, class_weight=
'balanced')
```

We now check our accuracy cross-validating via 10 different train_test_split

In [9]:
```
acc = []

for i in range(10):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.3)
    pipe = make_pipeline(preproc, lsvc).fit(X_train, y_train)
    pred = pipe.predict(X_test)
    acc.append(accuracy_score(y_test, pred))
print(np.array(acc).mean())
```

```
0.959550561797753
```

We fit our entire train dataframe to our Pipeline.

In [ ]:
```
pipe_t = make_pipeline(preproc, lsvc).fit(X, y)
```

We load the test dataframe and repeat the previous vectorization processes.

In [11]:
```
df_test = pd.read_csv('testset_notarget.csv').drop('Unnamed: 0', ax
is=1)
df_test['vector'] = [nlp(text).vector for text in df_test.question]
Xt = df_test[['question', 'vector']]

for i, row in Xt.iterrows():
    for j, vec in enumerate(Xt.loc[i, 'vector']):
        Xt.loc[i, f'Vec_{j+1}'] = vec
Xt = Xt.drop('vector', axis=1)
```

And, finally, we predict the test questions groups.

In [16]:
```
pred_t = pipe_t.predict(Xt)
```

In [24]:
```
df_out = pd.concat([df_test, pd.Series(pred_t)], axis=1).drop('vect
or', axis=1).rename({0: 'pred_group'}, axis=1)
df_out.head()
```

Out[24]:

	question	pred_group
0	delete item on list	c
1	what brand hair spray does donald trump use	a
2	play the song by michael jackson	b
3	what events are near me	c
4	can you reserve a ticket to grand rapids by train	a

We are now ready to proceed to the intent classifcation through BERT.

3. Intent Classification with BERT

Trying to predict the scenario (or the scenario group) mainly involves a semantic analysis of the question. This means that it is feasible to reach a high accuracy by only looking at the presence of some words, not considering at all the syntactic role of these words. To predict the intent, though, this approach is no longer optimal, since the range of words present in the questions is considerably restricted. This fact makes it paramount to draw additional information from the syntactic roles of the words present in a question.
Questions like "*Do I have any alarms set?*" and "*Remove all set alarms*" share a significant portion of their vocabulary, and knowing that both *set* and *alarms* are in the question no longer helps if we cannot determine if *set* refers to the *alarms* or if it is a verb acting on the *alarms* .

In other words, we need a **Contextual Model**, i.e. a model which takes into consideration the context of the sentence and, most importantly, the **syntactical relantionships** between the words. We chose BERT for this task, as it is one of the most advanced models for text classification, having undergone a contextual training fit of the whole Wikipedia and Books Corpus (>10,000 books of different genres).

The pre-trained BERT is already available in the PyTorch package as a *BertForSequenceClassification* model, all we need to do is fine-tune the last layer of the model in order for it to be able to predict an intent among our intended range.

The first step is importing the dataset.

In [ ]:
```
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

# install
!pip install pytorch-pretrained-bert pytorch-nlp

# BERT imports
import torch
from torch.utils.data import TensorDataset, DataLoader, RandomSampl
er, SequentialSampler
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from pytorch_pretrained_bert import BertTokenizer, BertConfig
from pytorch_pretrained_bert import BertAdam, BertForSequenceClassi
fication
from tqdm import tqdm, trange
```

```
import pandas as pd
import io
import numpy as np
import matplotlib.pyplot as plt
% matplotlib inline

# specify GPU device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
")
n_gpu = torch.cuda.device_count()
torch.cuda.get_device_name(0)
import pandas as pd
df_complete = pd.read_csv("/content/dataset_intent_train.csv", sep=
";")

def group(groups: list):
    for group in groups:
        to_group = group.split("/")
        for scen in to_group:
            df_complete["scenario"][df_complete["scenario"] == scen
] = group
    return sorted(list(set(df_complete["scenario"])))

scenarios = group(['alarm',
                   'email',
                   'iot',
                   'music/audio/play',
                   'recommendation/lists/datetime/calendar',
                   'weather/cooking/transport/general/social/news/t
akeaway/qa'])

df = df_complete[df_complete["scenario"]=='alarm']  ## predicting h
ere for ALARM
```

Now we modify our questions to meet BERT's requirements for input text. Each sentence must begin with a "[CLS]" token and must end with a "[SEP]" token. We can then tokenize the sentences with the builtin *BertTokenizer*, which will tokenize the words in a way that BERT can understand, and for which has already undergone extensive fitting. We also need to encode the intent labels.

In [ ]:
```
sentences = ["[CLS] " + question + " [SEP]" for question in df["que
stion"]]

# Tokenize with BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_l
ower_case=True)
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]

MAX_LEN = 128
# Pad our input tokens
input_ids = pad_sequences([tokenizer.convert_tokens_to_ids(txt) for
txt in tokenized_texts],
                          maxlen=MAX_LEN, dtype="long", truncating=
"post", padding="post")
# Use the BERT tokenizer to convert the tokens to their index numbe
rs in the BERT vocabulary
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_
texts]
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long",
truncating="post", padding="post")

intent_labels = {intent: i for (i, intent) in enumerate(list(set(df
["intent"])))}
labels = np.asarray(df["intent"].apply(lambda intent: intent_labels
[intent]))
```

Now it is time to create all the tensor datasets we will need for fitting the model.

In [ ]:
```
attention_masks = []
# Create a mask of 1s for each token followed by 0s for padding
for seq in input_ids:
  seq_mask = [float(i>0) for i in seq]
  attention_masks.append(seq_mask)


train_inputs, validation_inputs, train_labels, validation_labels = 
train_test_split(input_ids, labels,
                                                            random_
state=2018, test_size=0.1)
train_masks, validation_masks, _, _ = train_test_split(attention_ma
sks, input_ids,
                                                  random_state=2018, tes
t_size=0.1)

# Convert all of our data into torch tensors, the required datatype 
for our model
train_inputs = torch.tensor(train_inputs)
validation_inputs = torch.tensor(validation_inputs)
train_labels = torch.tensor(train_labels)
validation_labels = torch.tensor(validation_labels)
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)

# Select a batch size for training.
batch_size = 32

# Create an iterator of our data with torch DataLoader
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, ba
tch_size=batch_size)
validation_data = TensorDataset(validation_inputs, validation_masks
, validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validat
ion_sampler, batch_size=batch_size)

model = BertForSequenceClassification.from_pretrained("bert-base-un
cased", num_labels=len(intent_labels))
```

Now we are ready to fit and evaluate the model.

In [ ]:
```
torch.cuda.empty_cache()
model.cuda()
# BERT fine-tuning parameters
param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'gamma', 'beta']
optimizer_grouped_parameters = [
    {'params': [p for n, p in param_optimizer if not any(nd in n fo
```

```
r nd in no_decay)],
     'weight_decay_rate': 0.01},
    {'params': [p for n, p in param_optimizer if any(nd in n for nd
in no_decay)],
     'weight_decay_rate': 0.0}
]

optimizer = BertAdam(optimizer_grouped_parameters,
                     lr=2e-5,
                     warmup=.1)

# Function to calculate the accuracy of our predictions vs labels
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)

# Store our loss and accuracy for plotting
train_loss_set = []
# Number of training epochs
epochs = 4

# BERT training loop
for _ in trange(epochs, desc="Epoch"):

  ## TRAINING

  # Set our model to training mode
  model.train()
  # Tracking variables
  tr_loss = 0
  nb_tr_examples, nb_tr_steps = 0, 0
  # Train the data for one epoch
  for step, batch in enumerate(train_dataloader):
    # Add batch to GPU
    batch = tuple(t.to(device) for t in batch)
    # Unpack the inputs from our dataloader
    b_input_ids, b_input_mask, b_labels = batch
    # Clear out the gradients (by default they accumulate)
    optimizer.zero_grad()
    # Forward pass
    loss = model(b_input_ids, token_type_ids=None, attention_mask=b
_input_mask, labels=b_labels)
    train_loss_set.append(loss.item())
    # Backward pass
    loss.backward()
    # Update parameters and take a step using the computed gradient
    optimizer.step()
    # Update tracking variables
    tr_loss += loss.item()
    nb_tr_examples += b_input_ids.size(0)
    nb_tr_steps += 1
  print("Train loss: {}".format(tr_loss/nb_tr_steps))
```

```
  ## VALIDATION

  # Put model in evaluation mode
  model.eval()
  # Tracking variables
  eval_loss, eval_accuracy = 0, 0
  nb_eval_steps, nb_eval_examples = 0, 0
  # Evaluate data for one epoch
  for batch in validation_dataloader:
    # Add batch to GPU
    batch = tuple(t.to(device) for t in batch)
    # Unpack the inputs from our dataloader
    b_input_ids, b_input_mask, b_labels = batch
    # Telling the model not to compute or store gradients, saving m
emory and speeding up validation
    with torch.no_grad():
      # Forward pass, calculate logit predictions
      logits = model(b_input_ids, token_type_ids=None, attention_ma
sk=b_input_mask)
    # Move logits and labels to CPU
    logits = logits.detach().cpu().numpy()
    label_ids = b_labels.to('cpu').numpy()
    tmp_eval_accuracy = flat_accuracy(logits, label_ids)
    eval_accuracy += tmp_eval_accuracy
    nb_eval_steps += 1
  print("Validation Accuracy: {}".format(eval_accuracy/nb_eval_step
s))
```

The last piece, a little helper class with a simple `predict` method that will give us the final predictions.

In [ ]:
```
class BertForIntent:
  def __init__(self, model):
    self.model = model

  def predict(self, questions):
    # Create sentence and label lists
    # Tokenize all of the sentences and map the tokens to thier wor
d IDs.

    # For every sentence...
    sentences = ["[CLS] " + question + " [SEP]" for question in que
stions]

    # Tokenize with BERT tokenizer
    tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
do_lower_case=True)
    tokenized_texts = [tokenizer.tokenize(sent) for sent in sentenc
es]
    MAX_LEN = 128
    # Pad our input tokens
    input_ids = pad_sequences([tokenizer.convert_tokens_to_ids(txt)
for txt in tokenized_texts],
```

```
                                      maxlen=MAX_LEN, dtype="long", truncat
ing="post", padding="post")
    # Use the BERT tokenizer to convert the tokens to their index n
umbers in the BERT vocabulary
    input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokeni
zed_texts]
    input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="lon
g", truncating="post", padding="post")

    # Create attention masks
    attention_masks = []

    # Create a mask of 1s for each token followed by 0s for padding
    for seq in input_ids:
      seq_mask = [float(i>0) for i in seq]
      attention_masks.append(seq_mask)

    # Convert to tensors.
    prediction_inputs = torch.tensor(input_ids)
    prediction_masks = torch.tensor(attention_masks)

    # Set the batch size.
    batch_size = 1

    # Create the DataLoader.
    prediction_data = TensorDataset(prediction_inputs, prediction_m
asks)
    prediction_sampler = SequentialSampler(prediction_data)
    prediction_dataloader = DataLoader(prediction_data, sampler=pre
diction_sampler, batch_size=batch_size)


    self.model.cuda()

    self.model.eval()

    # Tracking variables
    predictions = []

    # Predict
    for batch in prediction_dataloader:
      # Add batch to GPU
      batch = tuple(t.to(device) for t in batch)

      # Unpack the inputs from our dataloader
      b_input_ids, b_input_mask = batch

      # Telling the model not to compute or store gradients, saving
memory and
      # speeding up prediction
      with torch.no_grad():
          # Forward pass, calculate logit predictions
          outputs = self.model(b_input_ids, token_type_ids=None,
                              attention_mask=b_input_mask)
```

```
        logits = outputs[0]

        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()

        # Store predictions and true labels
        predictions.append(np.argmax(logits))
    return predictions
```

The code above is for the *alarm* group, we only need to repeat it for all the other groups.