1 GPT-3 Related Code Listings

In this section, we include code listings relevant for generating data using GPT-3. The output of these listings can then be analyzed using the replication code in the online replication repository.

1.1 Study 1: Generating words describing the outparty

This code reads data from the original Pigeonholing Partisan's dataset, constructs a backstory for each participant, and queries GPT-3 for a response.

```
import time
import numpy as np
import openai
from tqdm import tqdm
import pickle
import re
import sys

#
# ============================================================================
#

openai.api_key = "PUT YOUR API KEY HERE"

def do_query( prompt, max_tokens=2 ):
    response = openai.Completion.create(
        engine="davinci",
        prompt=prompt,
        temperature=0.7,
        max_tokens=max_tokens,
        top_p=1,
        logprobs=100,
    )
    time.sleep(1.0) # to avoid rate limiters
    return response

#
# ============================================================================
#

def uniqvals( users, field ):
    vals = [ users[id][field] for id in users.keys() ]
    return list(set(vals))

fields_of_interest = {
    "Gender": { "Male":"male", "Female":"female", '':'' },
    "Hisp":{ "Hispanic":"Hispanic", "Not Hispanic":'', '':'' },
    "WHITE":{ "White":"white", "Non-white":"", '':'' },
    "Ideo": {
        '':'',
        'Liberal':'liberal',
        'Slightly conservative':'slightly conservative',
        'Conservative':'conservative',
        'Slightly liberal':'slightly liberal',
        "Moderate/Haven't thought about it":'moderate',
        'Extremely Liberal':'extremely liberal',
        'Extremely conservative':'extremely conservative',
```

```
        },
    "PID7": {
        '':'',
        'Ind':'am an independent',
        'Strong D':'am a strong Democrat',
        'Strong R':'am a strong Republican',
        'Lean D':'lean towards Democrats',
        'Lean R':'lean towards Rebublicans',
        'Weak D':'am a weak Democrat',
        'Weak R':'am a weak Republican',
        },

    "Inc": {
        '':'',
        'Less than $15K':'very poor',
        '$15K to $25K':'poor',
        '$25K to $50K':'poor',
        '$50K to $75K':'middle-class',
        '$75K to $100K':'middle-class',
        '$100K to $150K':'middle-class',
        '$150K to $200K':'upper-class',
        '$200K to $250K':'upper-class',
        '$250K to $500K':'upper-class',
        'Prefer not to answer':'',
        '-8':'',
    },
    }

def mapper( profile ):
    results = {}
    for k in profile.keys():
        if k in fields_of_interest:
            results[k] = fields_of_interest[k][profile[k]]

    if profile['Age'] != '':
        age = int( profile['Age'] )
        if age >= 18 and age < 25:
            results['Age'] = 'young'
        if age >= 25 and age < 40:
            results['Age'] = 'middle-aged'
        if age >= 40 and age < 60:
            results['Age'] = 'old'
        if age >= 60 and age < 100:
            results['Age'] = 'very old'
    else:
        results['Age'] = ''

    return results


lines = open( "../human_data/Pigeonholing Partisans.csv", "r" ).readlines()
dmap = {}
header_vals = lines[0].strip().split(",")
for line in lines[1:]:
    parts = line.strip().split(",")
    if len(parts) != len(header_vals):
        print("Error on line: " + line)
        continue
    dmap[ parts[0] ] = {}
    for hind,h in enumerate(header_vals):
        dmap[ parts[0] ][ header_vals[hind] ] = parts[hind]
```

```
party_list = ["Republican", "Democratic"]

results = {}
ids = dmap.keys()
for id in tqdm( ids ):
    results[id] = {}

    user_profile = mapper( dmap[id] )

    for party in party_list:

        print( f"---------------------------------------- {id} {party}" )

        prompt = ""

        if user_profile['Ideo'] != '':
            prompt += "Ideologically, I describe myself as " + user_profile['Ideo'] + ". "

        if user_profile['PID7'] != '':
            prompt += "Politically, I " + user_profile['PID7'] + ". "

        if user_profile['WHITE'] == 'White':
            prompt += "Racially, I am white. "

        if user_profile['Hisp'] == 'Hispanic':
            prompt += "Racially, I am Hispanic. "

        if user_profile['Gender'] != '':
            prompt += "I am " + user_profile['Gender'] + ". "

        if user_profile['Inc'] != '':
            prompt += "Financially, I am " + user_profile['Inc'] + ". "

        if user_profile['Age'] != '':
            prompt += "In terms of my age, I am " + user_profile['Age'] + ". "

        prompt += "When I am asked to write down four words that typically describe people"
        prompt += "who support the " + party + " Party, I respond with: 1."

        print( prompt )

        full_response = do_query( prompt, max_tokens=128 )
        text = full_response['choices'][0]['text']
        text = "1." + text  # since "1." is part of the prompt

        print(text)

        results[id][party] = text

pickle.dump( results, open( f"../gpt3_data/gpt3_synthetic_words.pkl", "wb" ) )
```

1.2 Study 1: Analyzing words generated by GPT-3

This code uses simple regular expressions to analyze the response from GPT-3.

```
import numpy as np
import pickle
import re

def extract_phrase( text, ind ):
```

```
    match = re.search( str(ind) + '.[ \t]+([A-Za-z /()-]+)([0-9]|$|\n|,|\.)', text )
    if match is None:
        return ""
    else:
        return match.group(1)

def extract_four( text ):
    # text = text.lower()
    text = re.sub( ' ', ' ', text )  # replace alternative space
    text = re.sub('[^A-Za-z0-9,. /\n\'-]', '', text)
    results = []
    for tmp in range(4):
        single_phrase = extract_phrase( text, tmp+1 )
        results.append( single_phrase )
    return results

party_list = ["Republican", "Democratic"]

results = pickle.load( open( f"../gpt3_data/gpt3_synthetic_words.pkl","rb") )

fout = open( "./gpt3_results_full.csv", "w" )

for id in results.keys():
    print( "*********************************************************************************")
    print( id )
    print( "*********************************************************************************")

    d_text = results[id][party_list[1]]
    d_results = extract_four( d_text )
    print( "================================================")
    print( d_text )
    print( "------------------------------------------------")
    print( d_results )

    r_text = results[id][party_list[0]]
    r_results = extract_four( r_text )
    print( "================================================")
    print( r_text )
    print( "------------------------------------------------")
    print( r_results )


    print()

    print( f"{id}" +
           "," + ",".join( [ '"'+x+'"' for x in d_results ] ) +
           "," + ",".join( [ '"'+x+'"' for x in r_results ] ),
           file=fout )
```

1.3 Study 2: Vote Prediction - Common Analytics

This code (named "common.py") contains common analytic routines:

```
import openai
import numpy as np
import time

openai.api_key = "PUT YOUR KEY HERE"

def lc( t ):
```

```
    return t.lower()

def uc( t ):
    return t.upper()

def mc( t ):
    tmp = t.lower()
    return tmp[0].upper() + t[1:]

def gen_variants( toks ):
    results = []

    variants = [ lc, uc, mc ]

    for t in toks:
        for v in variants:
            results.append( " " + v(t) )

    return results

def logsumexp( log_probs ):
    log_probs = log_probs - np.max(log_probs)
    log_probs = np.exp(log_probs)
    log_probs = log_probs / np.sum( log_probs )
    return log_probs

def extract_probs( lp ):
    lp_keys = list( lp.keys() )
    ps = [ lp[k] for k in lp_keys ]
    ps = logsumexp( np.asarray(ps) )
    vals = [ (lp_keys[ind], ps[ind]) for ind in range(len(lp_keys)) ]

    vals = sorted( vals, key=lambda x: x[1], reverse=True )

    result = {}
    for v in vals:
        result[ v[0] ] = v[1]

    return result

def do_query( prompt, max_tokens=2, engine="davinci" ):
    response = openai.Completion.create(
        engine=engine,
        prompt=prompt,
        temperature=0.7,
        max_tokens=max_tokens,
        top_p=1,
        logprobs=100,
    )

    token_responses = response['choices'][0]['logprobs']['top_logprobs']

    results = []
    for ind in range(len(token_responses)):
        results.append( extract_probs( token_responses[ind] ) )

    return results, response

def collapse_r( response, toks ):
    total_prob = 0.0
    for t in toks:
        if t in response:
```

```
            total_prob += response[t]
    return total_prob

def print_response( template_val, tok_sets, response ):
    #print( f"{template_val}" )

    print( tok_sets )

    tr = []
    for tok_set_key in tok_sets.keys():
        toks = tok_sets[tok_set_key]
        full_prob = collapse_r( response[0], toks )
        tr.append( full_prob )
        #print( f";{tok_set_key};{full_prob}", end="" )
        #print( "\t{:.2f}".format(full_prob), end="" )
    print("\t\t",end="")
    tr = np.asarray( tr )
    tr = tr / np.sum(tr)
    for ind, tok_set_key in enumerate( tok_sets.keys() ):
        print( f"\t{tok_set_key}\t{tr[ind]}", end="" )
        #print( "\t{:.2f}".format(tr[ind]), end="" )
    print("")

def parse_response( template_val, tok_sets, response ):
    tr = []
    for tok_set_key in tok_sets.keys():
        toks = tok_sets[tok_set_key]
        full_prob = collapse_r( response[0], toks )
        tr.append( full_prob )
    tr = np.asarray( tr )
    tr = tr / np.sum(tr)

    results = {}
    for ind, tok_set_key in enumerate( tok_sets.keys() ):
        results[ tok_set_key ] = tr[ind]
    return results


def run_prompts( prompts, tok_sets, engine="davinci" ):
    results = []
    for prompt in prompts:
        #print("-------------------------------------------------")
        #print( prompt )
        response, full_response = do_query( prompt, max_tokens = 2, engine=engine )
        #print( response )
        #print_response( prompt, tok_sets, response )
        simp_results = parse_response( prompt, tok_sets, response )
        #print( simp_results )
        time.sleep( 0.1 )
        results.append( (simp_results, response, full_response) )
    return results

def run_experiment( template, template_vals, tok_sets ):
    prompts = []
    for template_val in template_vals:
        grounded_prompt = template.replace( "XXX", template_val )
        prompts.append( grounded_prompt )
    return run_prompts( prompts, tok_sets )
```

1.4 Study 2: Vote Prediction - Common Templates

This code (named "anes_common.py") contains common data structures for templatizing the ANES data:

```
fips_state_map = {
    1:"Alabama",
    2:"Alaska",
    4:"Arizona",
    5:"Arkansas",
    6:"California",
    8:"Colorado",
    9:"Connecticut",
    10:"Delaware",
    # 11:"DC",
    12:"Florida",
    13:"Georgia",
    15:"Hawaii",
    16:"Idaho",
    17:"Illinois",
    18:"Indiana",
    19:"Iowa",
    20:"Kansas",
    21:"Kentucky",
    22:"Louisiana",
    23:"Maine",
    24:"Maryland",
    25:"Massachusetts",
    26:"Michigan",
    27:"Minnesota",
    28:"Mississippi",
    29:"Missouri",
    30:"Montana",
    31:"Nebraska",
    32:"Nevada",
    33:"New Hampshire",
    34:"New Jersey",
    35:"New Mexico",
    36:"New York",
    37:"North Carolina",
    38:"North Dakota",
    39:"Ohio",
    40:"Oklahoma",
    41:"Oregon",
    42:"Pennsylvania",
    44:"Rhode Island",
    45:"South Carolina",
    46:"South Dakota",
    47:"Tennessee",
    48:"Texas",
    49:"Utah",
    50:"Vermont",
    51:"Virginia",
    53:"Washington",
    54:"West Virginia",
    55:"Wisconsin",
    56:"Wyoming",
    }
```

1.5 Study 2: Vote Prediction - ANES 2012 template

This code (named "anes2012.py") contains templates for templatizing the ANES 2012 dataset:

```
from common import *
from anes_common import *

SEP=','
OUTPUT_FN = "./full_results_2012.pkl"
OUTPUT_CSV = "./full_results_2012.csv"
ID_COL = "caseid"
VOTE_COL = "presvote2012_x"
K1 = "romney"
K2 = "obama"

tok_sets = {
    "romney": gen_variants( [ "romney", "mitt", "republican", "conservative" ] ),
    "obama": gen_variants( [ "obama", "barack", "democrat", "democratic", "liberal"] ),
    }

query = "In the 2012 presidential election, I voted for"

ANES_FN = './2012 ANES.csv'

fields_of_interest = {
    # race
    'dem_raceeth_x': {
        "template":"Racially, I am XXX.",
        "valmap":{ 1:'white', 2:'black', 3:'asian', 4:'native American', 5:'hispanic' },
        },

    # discuss_politics
    'discuss_disc': {
        "template":"XXX",
        "valmap":{1:'I like to discuss politics with my family and friends.',
                  2:'I never discuss politics with my family or friends.'},
        },

    # ideology
    'libcpre_self': {
        "template":"Ideologically, I am XXX.",
        "valmap":{
        1:"extremely liberal",
        2:"liberal",
        3:"slightly liberal",
        4:"moderate",
        5:"slightly conservative",
        6:"conservative",
        7:"extremely conservative"},
        },

    # party
    'pid_x': {
        "template":"Politically, I am XXX.",
        "valmap":{
        1:"a strong democrat",
        2:"a weak Democrat",
        3:"an independent who leans Democratic",
        4:"an independent",
        5:"an independent who leans Republican",
```

```
        6:"a weak Republican",
        7:"a strong Republican"},
        },

    # church_goer
    'relig_church': {
        "template":"I XXX.",
        "valmap":{ 1:"attend church", 2:"do not attend church"},
        },

    # age
    'dem_age_r_x': {
        "template":"I am XXX years old.",
        "valmap":{},
        },

    # gender
    'gender_respondent_x': {
        "template":"I am a XXX.",
        "valmap":{ 1:"man", 2:"woman"},
        },

    # political_interest
    'paprofile_interestpolit': {
        "template":"I am XXX interested in politics.",
        "valmap":{1:"very", 2:"somewhat", 3:"not very", 4:"not at all"},
        },

    # patriotism
    'patriot_flag': {
        "template":"It makes me feel XXX to see the American flag.",
        "valmap":{
        1:"extremely good",
        2:"moderately good",
        3:"a little good",
        4:"neither good nor bad",
        5:"a little bad",
        6:"moderately bad",
        7:"extremely bad"},
        },

    # this is sample address, which may be different than registration address...?
    'sample_stfips': {
        "template":"I am from XXX.",
        "valmap":fips_state_map,
        },

    }
```

1.6 Study 2: Vote Prediction - ANES 2016 template

This code (named "anes2016.py") contains templatizing information for the ANES 2016 dataset:

```
from common import *
from anes_common import *

OUTPUT_FN = "./full_results_2016.pkl"
OUTPUT_CSV = "./full_results_2016.csv"
SEP='|'
```

```
ID_COL = "V160001_orig"
VOTE_COL = "V162062x"
K1 = "trump"
K2 = "clinton"

tok_sets = {
    "trump": gen_variants( [ "trump", "donald", "republican", "conservative" ] ),  # the republican, mr trump
    "clinton": gen_variants( [ "clinton", "hillary", "rodham", "senator", "democrat", "democratic", "liberal"] ),
    }

query = "In the 2016 presidential election, I voted for"

ANES_FN = '../anes2016/anes_timeseries_2016_rawdata.txt'

fields_of_interest = {
    # race V161310x 1= white 2= black 3 = asian 5 = hispanic
    'V161310x': {
        "template":"Racially, I am XXX.",
        "valmap":{ 1:'white', 2:'black', 3:'asian', 4:'native American', 5:'hispanic' },
        },

    # discuss_politics V162174 1=yes discuss politics, 2=never discuss politics
    'V162174': {
        "template":"XXX",
        "valmap":{
        1:'I like to discuss politics with my family and friends.',
        2:'I never discuss politics with my family or friends.'},
        },

    # ideology V161126 1-7 = extremely liberal, ..., extremely conservative
    'V161126': {
        "template":"Ideologically, I am XXX.",
        "valmap":{
        1:"extremely liberal",
        2:"liberal",
        3:"slightly liberal",
        4:"moderate",
        5:"slightly conservative",
        6:"conservative",
        7:"extremely conservative"},
        },

    # party V161158x
    'V161158x': {
        "template":"Politically, I am XXX.",
        "valmap":{
        1:"a strong democrat",
        2:"a weak Democrat",
        3:"an independent who leans Democratic",
        4:"an independent",
        5:"an independent who leans Republican",
        6:"a weak Republican",
        7:"a strong Republican"},
        },

    # church_goer V161244
    'V161244': {
        "template":"I XXX.",
        "valmap":{ 1:"attend church", 2:"do not attend church"},
        },

    # age V161267
```

```
    'V161267': {
        "template":"I am XXX years old.",
        "valmap":{},
        },

    # gender  V161342 1=male  2=female
    'V161342': {
        "template":"I am a XXX.",
        "valmap":{ 1:"man", 2:"woman"},
        },

    # political_interest = if_else(V162256 > 0, V162256, NA_real_),
    'V162256': {
        "template":"I am XXX interested in politics.",
        "valmap":{1:"very", 2:"somewhat", 3:"not very", 4:"not at all"},
        },

    # patriotism = if_else(V162125x > 0, V162125x, NA_real_))
    'V162125x': {
        "template":"It makes me feel XXX to see the American flag.",
        "valmap":{
        1:"extremely good",
        2:"moderately good",
        3:"a little good",
        4:"neither good nor bad",
        5:"a little bad",
        6:"moderately bad",
        7:"extremely bad"},
        },

    # this is sample address, which may be different than registration address...?
    'V161010d': {
        "template":"I am from XXX.",
        "valmap":fips_state_map,
        },

    }
```

1.7 Study 2: Vote Prediction - ANES 2020 Template

This code (named "anes2020.py") contains templates for the ANES 2020 data:

```
from common import *
from anes_common import *

SEP=','
OUTPUT_FN = "./full_results_2020.pkl"
OUTPUT_CSV = "./full_results_2020.csv"
ID_COL = "V200001"
VOTE_COL = "V202110x" # 1-Biden, 2-Trump, 3-Jorgensen, 4-Hawkins
K1 = "trump"
K2 = "biden"

tok_sets = {
    "trump": gen_variants( [ "donald", "trump", "republican", "conservative" ] ),
    "biden": gen_variants( [ "joe", "joseph", "biden", "democrat", "democratic", "liberal"] ),
    }

query = "In the 2020 presidential election, Donald Trump is the Republican candidate, "
query += "and Joe Biden is the Democratic candidate, and I voted for"
```

```
ANES_FN = './2020 ANES.csv'
fields_of_interest = {
    # race  1= white 2= black 3 = asian 5 = hispanic
    'V201549x': {
        "template":"Racially, I am XXX.",
        "valmap":{ 1:'white', 2:'black', 3:'asian', 4:'native American', 5:'hispanic' },
        },

    # discuss_politics 1=yes discuss politics, 2=never discuss politics
    'V202022': {
        "template":"XXX",
        "valmap":{
        1:'I like to discuss politics with my family and friends.',
        2:'I never discuss politics with my family or friends.'},
        },

    # ideology 1-7 = extremely liberal, ..., extremely conservative
    'V201200': {
        "template":"Ideologically, I am XXX.",
        "valmap":{
        1:"extremely liberal",
        2:"liberal",
        3:"slightly liberal",
        4:"moderate",
        5:"slightly conservative",
        6:"conservative",
        7:"extremely conservative"},
        },

    # party
    'V201231x': {
        "template":"Politically, I am XXX.",
        "valmap":{
        1:"a strong democrat",
        2:"a weak Democrat",
        3:"an independent who leans Democratic",
        4:"an independent",
        5:"an independent who leans Republican",
        6:"a weak Republican",
        7:"a strong Republican"},
        },

    # church_goer
    'V201452': {
        "template":"I XXX.",
        "valmap":{ 1:"attend church", 2:"do not attend church"},
        },

    # age
    'V201507x': {
        "template":"I am XXX years old.",
        "valmap":{},
        },

    # gender 1=male  2=female
    'V201600': {
        "template":"I am a XXX.",
        "valmap":{ 1:"man", 2:"woman"},
        },

    # political_interest = if_else(V162256 > 0, V162256, NA_real_),
```

```
    'V202406': {
        "template":"I am XXX interested in politics.",
        "valmap":{1:"very", 2:"somewhat", 3:"not very", 4:"not at all"},
        },

    # this is sample address, which may be different than registration address...?
    'V201007d': {
        "template":"I am from XXX.",
        "valmap":fips_state_map,
        },

    }
```

1.8 Study 2: Vote Prediction - Main predictor

This code generates vote predictions from ANES data, as well as cost estimates:

```
import sys
import pandas as pd
import pickle
from tqdm import tqdm

# for cost analysis
from transformers import GPT2Tokenizer

if sys.argv[1] == '2012':
    from anes2012 import *
if sys.argv[1] == '2016':
    from anes2016 import *
if sys.argv[1] == '2020':
    from anes2020 import *

from common import *

foi_keys = fields_of_interest.keys()

#
# ============================================================================================
# ============================================================================================
#

def cost_approximation(prompt, engine="davinci", tokenizer=None):
    possible_engines = ["davinci", "curie", "babbage", "ada"]
    assert engine in possible_engines, f"{engine} is not a valid engine"

    if tokenizer==None:
        tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

    num_tokens = len(tokenizer(prompt)['input_ids'])

    if engine == "davinci":
        cost = (num_tokens / 1000) * 0.0600
    elif engine == "curie":
        cost = (num_tokens / 1000) * 0.0060
    elif engine == "babbage":
        cost = (num_tokens / 1000) * 0.0012
    else:
        cost = (num_tokens / 1000) * 0.0008

    return cost, num_tokens
```

```
def gen_backstory( pid, df ):
    person = df.iloc[pid]

    backstory = ""

    for k in foi_keys:
        anes_val = person[k]
        elem_template = fields_of_interest[k]['template']
        elem_map = fields_of_interest[k]['valmap']

        if len(elem_map) == 0:
            backstory += " " + elem_template.replace( 'XXX', str(anes_val) )

        elif anes_val in elem_map:
            backstory += " " + elem_template.replace( 'XXX', elem_map[anes_val] )

    if backstory[0] == ' ':
        backstory = backstory[1:]

    return backstory

#
# ============================================================================================
# ============================================================================================
#

anesdf = pd.read_csv( ANES_FN, sep=SEP, encoding='latin-1' )

costs = []
numtoks = []
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

full_results = []
for pid in tqdm( range(len(anesdf)) ):

    if "V200003" in anesdf.iloc[pid] and anesdf.iloc[pid]["V200003"]==2:
        print( f"SKIPPING {pid}..." )
        # we want to exclude cases marked as 2 on this variable;
        # those are the panel respondents (interviewed in 2016 and 2020)
        continue

    anes_id = anesdf.iloc[pid][ID_COL]

    prompt = gen_backstory( pid, anesdf )
    prompt += " " + query

    #print("--------------------------------------------------")
    #print( prompt )

    cost, numtok = cost_approximation( prompt, engine="davinci", tokenizer=tokenizer )
    costs.append( cost )
    numtoks.append( numtok )

    results = run_prompts( [prompt], tok_sets, engine="davinci" )
    #print(results[0][0])
    full_results.append( (anes_id, prompt, results) )

print( "Total cost: ", np.sum(np.array(costs)) )
print( "Averge numtok: ", np.mean(np.array(numtoks)) )

pickle.dump( full_results, open(OUTPUT_FN,"wb") )
```

1.9 Study 2: Ablation Analysis

This code tests the effect of dropping individual elements of back stories, and asking GPT-3 to make predictions. The steps in this process are run by the following script:

```
#!/bin/bash

cd create_prompts
./run_all.sh
cd ..

# THIS REQUIRES AN OPENAI KEY TO RUN
cd run_prompts
./run_all.sh
cd ..

cd extract_stats
./run_all.sh
cd ..
```

The result is the data file in "data/pa/plot_data.pkl". This data can then be processed by the Replication Code to generate the final plot.

1.9.1 Creating the prompts

Prompts are created with the following shell script:

```
#!/bin/bash

# generate the full backstory
python3 main.py '2016' none none

# generate the empty backstory
python3 main.py '2016' all all

# ignore one field
python3 main.py '2016' V161310x race
python3 main.py '2016' V162174 discuss_politics
python3 main.py '2016' V161126 ideology
python3 main.py '2016' V161158x party
python3 main.py '2016' V161244 church_goer
python3 main.py '2016' V161267 age
python3 main.py '2016' V161342 gender
python3 main.py '2016' V162256 political_interest
python3 main.py '2016' V162125x patriotism
python3 main.py '2016' V161010d state

# ignore party and ideo fields
python3 main.py '2016' party_ideo party_ideo

# ignore all BUT one field
python3 main.py '2016' V161310x race_only
python3 main.py '2016' V162174 discuss_politics_only
python3 main.py '2016' V161126 ideology_only
python3 main.py '2016' V161158x party_only
python3 main.py '2016' V161244 church_goer_only
python3 main.py '2016' V161267 age_only
python3 main.py '2016' V161342 gender_only
```

```
python3 main.py '2016' V162256 political_interest_only
python3 main.py '2016' V162125x patriotism_only
python3 main.py '2016' V161010d state_only
```

Which calls the following script, named "main.py":

```
import sys
import pandas as pd
import pickle
from tqdm import tqdm
import os

def lc( t ):
    return t.lower()

def uc( t ):
    return t.upper()

def mc( t ):
    tmp = t.lower()
    return tmp[0].upper() + t[1:]

def gen_variants( toks ):
    results = []

    variants = [ lc, uc, mc ]

    for t in toks:
        for v in variants:
            results.append( " " + v(t) )

    return results

OUTPUT_FN = "./full_results_2016.pkl"
OUTPUT_CSV = "./full_results_2016.csv"
SEP='|'
ID_COL = "V160001_orig"
VOTE_COL = "V162062x"
VOTE_MAP = {
    1: "clinton",
    2: "trump",
}
K1 = "trump"
K2 = "clinton"

query = "In the 2016 presidential election, I voted for"

ANES_FN = '2016 ANES.csv'

tok_sets = {
    "trump": gen_variants( [ "trump", "donald", "republican", "conservative" ] ),  # the republican, mr trump
    "clinton": gen_variants( [ "clinton", "hillary", "rodham", "senator", "democrat", "democratic", "liberal"] ),
    }

fips_state_map = {
    1:"Alabama",
    2:"Alaska",
    4:"Arizona",
    5:"Arkansas",
    6:"California",
    8:"Colorado",
```

```
    9:"Connecticut",
    10:"Delaware",
    # 11:"DC",
    12:"Florida",
    13:"Georgia",
    15:"Hawaii",
    16:"Idaho",
    17:"Illinois",
    18:"Indiana",
    19:"Iowa",
    20:"Kansas",
    21:"Kentucky",
    22:"Louisiana",
    23:"Maine",
    24:"Maryland",
    25:"Massachusetts",
    26:"Michigan",
    27:"Minnesota",
    28:"Mississippi",
    29:"Missouri",
    30:"Montana",
    31:"Nebraska",
    32:"Nevada",
    33:"New Hampshire",
    34:"New Jersey",
    35:"New Mexico",
    36:"New York",
    37:"North Carolina",
    38:"North Dakota",
    39:"Ohio",
    40:"Oklahoma",
    41:"Oregon",
    42:"Pennsylvania",
    44:"Rhode Island",
    45:"South Carolina",
    46:"South Dakota",
    47:"Tennessee",
    48:"Texas",
    49:"Utah",
    50:"Vermont",
    51:"Virginia",
    53:"Washington",
    54:"West Virginia",
    55:"Wisconsin",
    56:"Wyoming",
    }

fields_of_interest = {
    # race V161310x 1= white 2= black 3 = asian 5 = hispanic
    'V161310x': {
        "template":"Racially, I am XXX.",
        "valmap":{ 1:'white', 2:'black', 3:'asian', 4:'native American', 5:'hispanic' },
        "colname": "race",
        },

    # discuss_politics V162174 1=yes discuss politics, 2=never discuss politics
    'V162174': {
        "template":"XXX",
        "valmap":{1:'I like to discuss politics with my family and friends.',
        2:'I never discuss politics with my family or friends.'},
        "colname": "discuss_politics",
        },
```

```
    # ideology V161126 1-7 = extremely liberal, liberal, slightly liberal,
    # moderate, slightly conservative, conservative, extremely conservative
    'V161126': {
        "template":"Ideologically, I am XXX.",
        "valmap":{1:"extremely liberal",2:"liberal",
        3:"slightly liberal",4:"moderate",
        5:"slightly conservative",6:"conservative",
        7:"extremely conservative"},
        "colname": "ideology",
        },

    # party V161158x
    'V161158x': {
        "template":"Politically, I am XXX.",
        "valmap":{1:"a strong democrat", 2:"a weak Democrat",
        3:"an independent who leans Democratic", 4:"an independent",
        5:"an independent who leans Republican", 6:"a weak Republican",
        7:"a strong Republican"},
        "colname": "party",
        },

    # church_goer V161244
    'V161244': {
        "template":"I XXX.",
        "valmap":{ 1:"attend church", 2:"do not attend church"},
        "colname": "church_goer",
        },

    # age V161267
    'V161267': {
        "template":"I am XXX years old.",
        "valmap":{},
        "colname": "age",
        },

    # education V161270 <= 9, "High School or Less", >= 10 & V161270 <= 12,
    # "Some College / AA", 13, "Bachelor's", >=14 & V161270 <= 16, "Graduate Degree"
#    '': {
#        "template":"",
#        "valmap":{},
#        },

    # gender  V161342 1=male  2=female
    'V161342': {
        "template":"I am a XXX.",
        "valmap":{ 1:"man", 2:"woman"},
        "colname": "gender",
        },

#    # income  V161361x
#    '': {
#        "template":"",
#        "valmap":{},
#        },

    # vote_2016 = if_else(V162034 == 2, 0, if_else(as.numeric(V162062x) > 0, as.numeric(V162062x), NA_real_)),
#    '': {
#        "template":"",
#        "valmap":{},
#        },
```

```
    # political_interest = if_else(V162256 > 0, V162256, NA_real_),
    'V162256': {
        "template":"I am XXX interested in politics.",
        "valmap":{1:"very", 2:"somewhat", 3:"not very", 4:"not at all"},
        "colname": "political_interest",
        },

    # patriotism = if_else(V162125x > 0, V162125x, NA_real_))
    'V162125x': {
        "template":"It makes me feel XXX to see the American flag.",
        "valmap":{1:"extremely good", 2:"moderately good",
        3:"a little good", 4:"neither good nor bad",
        5:"a little bad", 6:"moderately bad", 7:"extremely bad"},
        "colname": "patriotism",
        },

    # this is sample address, which may be different than registration address...?
    'V161010d': {
        "template":"I am from XXX.",
        "valmap":fips_state_map,
        "colname": "state",
        },

    }


ignore_field = sys.argv[2]
ignore_field_hr = sys.argv[3] # human readable

#
# ================================================================================================
# ================================================================================================
#

foi_keys = fields_of_interest.keys()

def gen_backstory( pid, df ):
    person = df.iloc[pid]

    backstory = ""

    for k in foi_keys:

        # ignore NO fields - full backstory
        if ignore_field_hr == 'none':
            pass

        # ignore ALL fields - empty backstory
        elif ignore_field_hr == 'all':
            continue

        # ignore party and ideo
        elif ignore_field_hr == 'party_ideo':
            if k == 'V161126' or k == 'V161158x':
                continue

        # ignore all BUT one field
        elif ignore_field_hr.endswith("_only"):
            if k == ignore_field:
                pass
            else:
                continue
```

```
        # ignore just one field
        elif k == ignore_field:
            continue

        anes_val = person[k]
        elem_template = fields_of_interest[k]['template']
        elem_map = fields_of_interest[k]['valmap']

        if len(elem_map) == 0:
            backstory += " " + elem_template.replace( 'XXX', str(anes_val) )

        elif anes_val in elem_map:
            backstory += " " + elem_template.replace( 'XXX', elem_map[anes_val] )

    if len(backstory)>0 and backstory[0] == ' ':
        backstory = backstory[1:]

    return backstory

#
# ============================================================================================
# ============================================================================================
#

anesdf = pd.read_csv( ANES_FN, sep=SEP, encoding='latin-1' )

foi_keys = list( fields_of_interest.keys() )
# for each field of interest, there is valmap and colname
for k in foi_keys:
    # for each key, make a new column
    colname, valmap = fields_of_interest[k]['colname'], fields_of_interest[k]['valmap']
    anesdf[colname] = anesdf[k].map( valmap )


# add new columns: prompt token_sets, ground_truth, and template_name
# template name is "first_person_backstory"
anesdf['template_name'] = "first_person_backstory"

# prompt is empty to start
anesdf['prompt'] = ""

# token sets is tok_sets
anesdf['token_sets'] = [tok_sets] * len(anesdf)

# ground truth is same column as VOTE_COL
anesdf['ground_truth'] = anesdf[VOTE_COL].map(VOTE_MAP)


full_results = []
prompts = []
for pid in tqdm( range(len(anesdf)) ):

    if "V200003" in anesdf.iloc[pid] and anesdf.iloc[pid]["V200003"]==2:
        print( f"SKIPPING {pid}..." )
        # we want to exclude cases marked as 2 on this variable; those are the
        # panel respondents (interviewed in 2016 and 2020)
        prompts.append( "" )
        continue

    anes_id = anesdf.iloc[pid][ID_COL]
```

```
    prompt = gen_backstory( pid, anesdf )
    prompt += " " + query
    prompts.append( prompt )

anesdf['prompt'] = prompts
# drop anesdf where prompt is empty
anesdf = anesdf[anesdf['prompt'] != ""]

# check if ../data/anes{year} exists, if not, create it
if not os.path.exists( f"../data/anes{sys.argv[1]}_if_{ignore_field_hr}" ):
    os.makedirs( f"../data/anes{sys.argv[1]}_if_{ignore_field_hr}" )

# save to dir as ds.pkl
anesdf.to_pickle(f"../data/anes{sys.argv[1]}_if_{ignore_field_hr}/ds.pkl")
```

1.9.2 Running GPT-3 on the prompts

Prompts are run through GPT-3 with the following shell script:

```
#!/bin/bash

MODELS="gpt3-davinci"

DATASETS="anes2016_if_none anes2016_if_age anes2016_if_church_goer anes2016_if_discuss_politics \
anes2016_if_gender anes2016_if_ideology anes2016_if_political_interest anes2016_if_party \
anes2016_if_patriotism anes2016_if_race anes2016_if_state anes2016_if_party_ideo \
anes2016_if_age_only anes2016_if_church_goer_only anes2016_if_discuss_politics_only \
anes2016_if_gender_only anes2016_if_ideology_only anes2016_if_political_interest_only \
anes2016_if_party_only anes2016_if_patriotism_only anes2016_if_race_only anes2016_if_state_only \
anes2016_if_back_only"

export OPENAI_API_KEY="XXX YOUR KEY HERE"

for MODEL in $MODELS; do
    for DATASET in $DATASETS; do
        python3 pipeline_single.py ../data/${DATASET}/ds.pkl ${MODEL}
    done
done
```

which calls the following scripts, starting with "pipeline_single.py":

```
from datetime import date
from experiment import Experiment
from postprocessor import Postprocessor
from datetime import date
from pdb import set_trace as breakpoint
import os

def run_experiment(ds_path, model_name, n=500):

    # Pass data through model
    print("Passing data through model...")

    in_fname = ds_path
    date_str = date.today().strftime("%Y-%m-%d")

    # get directory where ds_path is located
    replace_str = f"_exp_results_{model_name.replace('/', '-')}_{date_str}.pkl"

    # replace .pkl with replace_str
```

```
    out_fname = in_fname.replace(".pkl", replace_str)

    Experiment(
        model_name=model_name,
        in_fname=in_fname,
        out_fname=out_fname,
    )

    model_name = model_name.replace('/', '-')

    # Postprocessing
    print("Postprocessing...")
    date_str = date.today().strftime("%d-%m-%Y")
    processed_in = out_fname

    # replace .pkl with _processed.pkl
    processed_out = processed_in.replace('.pkl', '_processed.pkl')

    Postprocessor(results_fname=processed_in,
                  save_fname=processed_out,
                  matching_strategy="startswith")

if __name__ == "__main__":
    import sys
    ds_path = sys.argv[1]
    model_name = sys.argv[2]

    run_experiment(ds_path=ds_path,
                   model_name=model_name)
```

which depends on the following files: Experiment.py:

```
from datetime import date
from lmsampler import LMSampler

import pandas as pd
import tqdm
import warnings

class Experiment:

    def __init__(self,
                 model_name,
                 ds_name=None,
                 in_fname=None,
                 out_fname=None,
                 n_probs=100):

        # Get in_fname and out_fname
        if ds_name is not None:
            # Use ds_name if it is available,
            # regardless of in_fname/out_fname
            # availability
            in_fname = f"data/{ds_name}/ds.pkl"
            date_str = date.today().strftime("%d-%m-%Y")
            # replace '/' with '-'
            out_fname = (f"data/{ds_name}/exp_results_"
                         f"{model_name.replace('/', '-')}_{date_str}.pkl")
        else:
            if (in_fname is None) or (out_fname is None):
                msg = ("Please either specify ds_name "
                       "OR (in_fname AND out_fname)")
```

```
                raise RuntimeError(msg)
            else:
                # Use the in_fname and out_fname
                # provided
                msg = ("Specifying in_fname and out_fname "
                       "instead of ds_name is not generally "
                       "recommended.")
                warnings.warn(msg)
                pass

        self._model_name = model_name
        self._out_fname = out_fname
        self._n_probs = n_probs

        # Open the data for this experiment
        self._ds_df = self._open_ds_df(in_fname)

        # Create model
        self._model = LMSampler(model_name)

        # Run the experiment and save the resulting data
        self._run()

    def _open_ds_df(self, ds_fname):
        return pd.read_pickle(ds_fname)

    def _run(self):
        resps = []

        for _, row in tqdm.tqdm(self._ds_df.iterrows(),
                                total=self._ds_df.shape[0]):
            try:
                resp = self._model.send_prompt(row["prompt"],
                                               n_probs=self._n_probs)
                resps.append(resp)
            except Exception as e:
                print(e)
                resps.append(None)

        # Make new df and save it
        self._ds_df["resp"] = resps
        self._ds_df["model"] = [self._model_name] * len(resps)
        self._ds_df.to_pickle(self._out_fname)
```

lm_gpt3.py:

```
from lmsampler_baseclass import LMSamplerBaseClass
import openai
import time


class LM_GPT3(LMSamplerBaseClass):
    def __init__(self, model_name):
        '''
        Supported models: 'ada', 'babbage', 'curie', 'davinci', 'gpt3-ada', gpt3-babbage', gpt3-curie', gpt3-davinci'
        '''
        super().__init__(model_name)
        if 'gpt3' in model_name:
            # engine is all text after 'gpt3-'
            self.engine = model_name.split('-')[1]
        else:
            self.engine = self.model_name
```

```
        # make sure engine is a valid model
        if self.engine not in ["ada", "babbage", "curie", "davinci"]:
            raise ValueError("Invalid model name. Must be one of: 'ada', 'babbage', 'curie', 'davinci'")
        # make sure API key is set
        if openai.api_key is None:
            raise ValueError("OpenAI API key must be set")


    def send_prompt(self, prompt, n_probs=100):
        start_time = time.time()
        response = openai.Completion.create(
            engine=self.engine,
            prompt=prompt,
            max_tokens=1,
            logprobs=n_probs,
        )
        logprobs = response['choices'][0]['logprobs']['top_logprobs'][0]
        # sort dictionary by values
        sorted_logprobs = dict(sorted(logprobs.items(), key=lambda x: x[1], reverse=True))
        # TODO - automate this?
        now = time.time()
        min_time = .1
        if now - start_time < min_time:
            wait_time = min_time - (now - start_time)
            wait_time = max(wait_time, 0)
            time.sleep(wait_time)
        return sorted_logprobs

    def sample_several(self, prompt, temperature=0, n_tokens=10):
        response = openai.Completion.create(
            engine=self.engine,
            prompt=prompt,
            max_tokens=n_tokens,
            temperature=temperature,
        )
        return response['choices'][0]['text']

if __name__ == '__main__':
    # test LM_GPT2
    lm = LM_GPT3("gpt3-ada")
    # probs = lm.send_prompt("What is the capital of France?\nThe capital of France is")
    text = lm.sample_several(prompt="What is the capital of France?\nThe capital of France is", temperature=0, n_tokens=50)
    print(text)
    pass
```

lm_utils.py:

```
def get_device_map(gpus, n_layers):
    '''
    Given a list of gpus, make a dictionary map from GPU to layer, covering all layers evenly.
    params:
        gpus (list): list of gpus
        n_layers (int): number of layers
    return:
        device_map (dict): dictionary mapping from GPU to layer
    '''
    layers_per_gpu = n_layers // len(gpus)
    remainder = n_layers % len(gpus)
    cutoffs = [layers_per_gpu * i + min(i, remainder) for i in range(len(gpus))]
    cutoffs.append(n_layers)

    device_map = {
```

```
        gpu: [i for i in range(cutoffs[gpu_num], cutoffs[gpu_num + 1])] for gpu_num, gpu in enumerate(gpus)
    }
    return device_map

if __name__ == '__main__':
    test_gpus = [0, 1, 3, 4, 5, 6, 7]
    n_layers = 48
    device_map = get_device_map(test_gpus, n_layers)
    print(device_map)
```

lmsampler_baseclass.py:

```
from abc import ABCMeta, abstractmethod

class LMSamplerBaseClass(metaclass=ABCMeta):
    def __init__(self, model_name):
        self.model_name = model_name

    @abstractmethod
    def send_prompt(self, prompt, n_probs):
        '''
        Sends the given prompt to a LM.
        Arguments:
            prompt (str) a prompt to be sent to LM
            n_probs (int) number of desired output probalities.
        Return:
            dict (str:int) a dictionary of log probabilities of length n_probs
        '''
        pass
```

lmsampler.py:

```
from lmsampler_baseclass import LMSamplerBaseClass
from lm_gpt3 import LM_GPT3
#from lm_gpt2 import LM_GPT2
#from lm_gptj import LM_GPTJ
#from lm_gptneo import LM_GPTNEO
#from lm_bert import LM_BERT

class LMSampler(LMSamplerBaseClass):
    '''
    Class to wrap all other LMSampler classes. This way, we can instantiate just by passing
    a model name, and it will initialize the corresponding class.
    '''
    def __init__(self, model_name):
        self.model_name = model_name
        super().__init__(model_name)
        '''
        Supported models:
            - GPT-3: 'gpt3-ada', 'gpt3-babbage', 'gpt3-curie', 'gpt3-davinci', 'ada', 'babbage', 'curie', 'davinci'
            - GPT-2: 'gpt2', 'gpt2-medium', 'gpt2-large', 'gpt2-xl', 'distilgpt2'
            - GPT-J: 'EleutherAI/gpt-j-6B'
            - GPT-Neo: 'EleutherAI/gpt-neo-2.7B', 'EleutherAI/gpt-neo-1.3B', 'EleutherAI/gpt-neo-125M'
            - BERT: 'bert-base-uncased', 'bert-base-cased'
        '''
        if model_name in ['gpt3-ada', 'gpt3-babbage', 'gpt3-curie', 'gpt3-davinci', 'ada', 'babbage', 'curie', 'davinci']:
            self.model = LM_GPT3(model_name)
        elif model_name in ['gpt2', 'gpt2-medium', 'gpt2-large', 'gpt2-xl', 'distilgpt2']:
            self.model = LM_GPT2(model_name)
        elif model_name in ['EleutherAI/gpt-j-6B']:
            self.model = LM_GPTJ(model_name)
```

```
        elif model_name in ['EleutherAI/gpt-neo-2.7B', 'EleutherAI/gpt-neo-1.3B', 'EleutherAI/gpt-neo-125M']:
            self.model = LM_GPTNEO(model_name)
        elif model_name in ['bert-base-uncased', 'bert-base-cased']:
            self.model = LM_BERT(model_name)
        else:
            raise ValueError(f'Model {model_name} not supported.')

    def send_prompt(self, prompt, n_probs=100):
        return self.model.send_prompt(prompt, n_probs)

    def sample_several(self, prompt, temperature=0, n_tokens=10):
        return self.model.sample_several(prompt, temperature, n_tokens)

if __name__ == '__main__':
    # model_name = 'gpt3-ada'
    model_name = 'gpt3-ada'
    sampler = LMSampler(model_name)
    print(sampler.sample_several('The capital of France is', temperature=0, n_tokens=10))
    #print(sampler.send_prompt('The best city in Spain is', 5))
    #print(sampler.send_prompt('In 2016, I voted for', 5))
```

postprocessor.py:

```
import pandas as pd
import numpy as np
from tqdm import tqdm
import os

def exponentiate(d):
    '''
    Exponentiates a dictionary's probabilities.
    '''
    return {k: np.exp(v) for k, v in d.items()}

def normalize(d):
    '''
    Normalizes a dictionary to sum to one.
    '''
    return {k: v/sum(d.values()) for k, v in d.items()}

def collapse_lower_strip(d):
    '''
    Collapses a dictionary's probabilities after doing lower case and strip, combining where needed.
    '''
    new_d = {}
    for k, v in d.items():
        new_k = k.lower().strip()
        # check if empty
        if new_k:
            # if already present, add to value
            if new_k in new_d:
                new_d[new_k] += v
            # if not present, add to dictionary
            else:
                new_d[new_k] = v
    return new_d

def collapse_token_sets(d, token_sets, matching_strategy='startswith'):
    '''
    Collapses a dictionary's probabilities combining into token sets.
    Args:
        d (dict): Dictionary of probabilities.
```

```
        token_sets (dict:str->list, list): Dictionary of token sets, where keys are categories and
            values are lists of tokens. If token_sets is a list, it is assumed that the keys are
            the lists of tokens.
        matching_strategy (str): Strategy for matching tokens. Can be 'startswith' or 'exact'.
    Returns:
        new_d (dict): Dictionary of probabilities after collapsing.
    '''
    # if token_sets is a list, convert to dictionary
    if isinstance(token_sets, list):
        token_sets = {t: [t] for t in token_sets}
    # make sure all items in dictionary are lists
    for k, v in token_sets.items():
        if not isinstance(v, list):
            token_sets[k] = [v]
    # create new dictionary
    new_d = {cat: 1e-10 for cat in token_sets.keys()}
    # iterate over tokens and probs in d
    for token, prob in d.items():
        # iterate over token sets
        for category, tokens in token_sets.items():
            # if token is in the token set, add to new dictionary
            match = False
            for t in tokens:
                if matching_strategy == 'startswith':
                    if t.lower().strip().startswith(token):
                        match = True
                elif matching_strategy == 'exact':
                    if token == t:
                        match = True
            if match:
                new_d[category] += prob
    return new_d

def prob_function(row):
    '''
    Collapses a row of probabilities.
    Args:
        row (pandas.Series): Series of probabilities.
    Returns:
        d (dict: str->float): Dictionary of probabilities after collapsing.
    '''
    d = row['resp']
    # logprobs to probs
    d = exponentiate(d)
    # lower strip collapse
    d = collapse_lower_strip(d)
    # if 'token_sets' in row, collapse token sets
    if 'token_sets' in row:
        # make sure 'token_sets' isn't None or an empty list
        if row['token_sets']:
            # check if matching strategy exists
            if 'matching_strategy' in row:
                d = collapse_token_sets(d, row['token_sets'], row['matching_strategy'])
            else:
                d = collapse_token_sets(d, row['token_sets'])
    return d

def coverage(d):
    '''
    Returns the sum of the values of d.
    '''
    return sum(d.values())
```

```
class Postprocessor:

    def __init__(self, results_fname, save_fname, matching_strategy=None):
        '''
        Instantiates a Postprocessor object.
        matching_strategy (str): Strategy for matching tokens. Can be 'startswith', 'exact', or None.
        '''

        # Read in dataframe specified by results_fname
        self.df = pd.read_pickle(results_fname)

        self.df['ground_truth'] = self.df['ground_truth'].astype(str)

        # get number of instances where 'resp' is missing
        num_missing = self.df.loc[self.df.resp.isnull()].shape[0]
        # print Dropping {} instances with missing responses
        print(f'Dropping {num_missing} instances with missing responses from', results_fname)
        # drop na where 'resp' is missing
        self.df = self.df.dropna(subset=['resp'])

        if matching_strategy:
            if matching_strategy not in ['startswith', 'exact']:
                msg = f"{matching_strategy} is not a valid matching strategy"
                raise RuntimeError(msg)
            self.df['matching_strategy'] = matching_strategy
        self.calculate_probs()
        self.calculate_coverage()
        self.normalize_probs()

        # calculate mutual information
        self.df = self.calculate_mutual_information(self.df)

        # calculate accuracy
        self.df = self.calculate_accuracy(self.df)

        # calculate correct weight
        self.df = self.calculate_correct_weight(self.df)

        # save df
        self.df.to_pickle(save_fname)

    def prob_dict_to_arr(self, d):
        '''
        Converts a probability dictionary into an array of probabilities.
        Args:
            d (dict: str->float): dictionary of probabilities for each category/token.
        Returns:
            arr (np.array): array of probabilities.
        '''
        arr = np.array([d[k] for k in d])
        return arr

    def agg_prob_dicts(self, dicts):
        '''
        Given a list of probability dictionaries, aggregate them.
        '''
        n = len(dicts)
        agg_dict = {}
        for d in dicts:
            for k, v in d.items():
```

```
            if k not in agg_dict:
                agg_dict[k] = v / n
            else:
                agg_dict[k] += v / n
    return agg_dict

def get_marginal_distribution(self, df, groupby='template_name'):
    '''
    Calculates the marginal distribution over categories.
    '''
    marginal_df = df.groupby(by=groupby)['probs'].agg(self.agg_prob_dicts)
    # series to df
    marginal_df = pd.DataFrame(marginal_df)
    return marginal_df

def calculate_correct_weight(self, df):
    '''
    Calculates the correct_weight. Adds a column called 'correct_weight' to df.
    df (pandas.DataFrame): dataframe with columns 'template_name', and 'ground_truth'

    Returns modified df.
    '''
    df = df.copy()

    # Our function for calculating weight on ground truth
    get_correct_weight = lambda row: row['probs'].get(row['ground_truth'], 0)

    # Calculate conditional entropy for each row
    df['correct_weight'] = df.apply(get_correct_weight, axis=1)

    return df

def entropy(self, arr):
    '''
    Given an array of probabilities, calculate the entropy.
    '''
    return -sum(arr * np.log(arr))

def calculate_conditional_entropy(self, df):
    '''
    Calculates the conditional entropy, up to a constant. Adds a column called 'conditional_entropy' to df.
    df (pandas.DataFrame): dataframe with columns 'template_name', and 'ground_truth'

    Returns modified df.
    '''
    df = df.copy()

    entropy_lambda = lambda row: self.entropy(self.prob_dict_to_arr(row['probs']))

    # Calculate entropy for each row
    df['conditional_entropy'] = df.apply(entropy_lambda, axis=1)

    return df

def calculate_mutual_information(self, df, groupby='template_name'):
    '''
    Calculate the mutual information between the template and the output distribution.
    '''
    # H(Y) - H(Y|X) method
    # first, calculate conditional entropy
    df = self.calculate_conditional_entropy(df)
    # get marginal distributions
```

```
        marginal_df = self.get_marginal_distribution(df, groupby)
        # get entropy
        entropy_lambda = lambda row: self.entropy(self.prob_dict_to_arr(row['probs']))
        marginal_df['entropy'] = marginal_df.apply(entropy_lambda, axis=1)
        # function to apply per row
        def mutual_inf(row):
            index = row[groupby]
            mutual_info = marginal_df.loc[index]['entropy'] - row['conditional_entropy']
            return mutual_info

        # apply function to each row
        df['mutual_inf'] = df.apply(mutual_inf, axis=1)

        return df

    def calculate_probs(self):
        '''
        Population the 'probs' column in the dataframe.
        '''
        self.df['probs'] = self.df.apply(prob_function, axis=1)

    def calculate_coverage(self):
        coverage_lambda = lambda row: coverage(row['probs'])
        self.df['coverage'] = self.df.apply(coverage_lambda, axis=1)

    def normalize_probs(self):
        '''
        Normalize the 'probs' column to sum to 1.
        '''
        self.df['probs'] = self.df['probs'].apply(normalize)

    def calculate_accuracy(self, df):
        '''
        Calculates the accuracy of the model. Adds a column called 'accuracy' to df.
        df (pandas.DataFrame): dataframe with columns 'template_name', and 'ground_truth'

        Returns modified df.
        '''
        df = df.copy()

        # if row['ground_truth'] starts with argmax(row['probs']) stripped and lowercase, then it's correct
        def accuracy_lambda(row):
            # guess is argmax of row['probs'] dict
            guess = max(row['probs'], key=row['probs'].get)
            # lower and strip
            guess = guess.lower().strip()
            if row['ground_truth'].lower().strip().startswith(guess):
                return 1
            else:
                return 0
        df['accuracy'] = df.apply(accuracy_lambda, axis=1)

        return df

def get_files_to_process():
    '''
    Step down into the data subdirectory, and get all files in all subdirectories that have
    'exp_results' in them, end with 'pkl', and don't include 'processed'.
    '''
    files_to_process = []
    for root, dirs, files in os.walk('data'):
        for file in files:
```

```
            if 'exp_results' in file and file.endswith('pkl') and 'processed' not in file:
                # if processed file already exists, don't process it again
                if file.replace('.pkl', '_processed.pkl') not in files:
                    files_to_process.append(os.path.join(root, file))
    return files_to_process

def process(files):
    for input_fname in tqdm(files):
        try:
            # save_fname is same name, but replace .pkl with _processed.pkl
            save_fname = input_fname.replace('.pkl', '_processed.pkl')
            # process
            Postprocessor(input_fname, save_fname)
        except Exception as e:
            print('Error processing {}'.format(input_fname))
            print(e)

def process_all():
    files_to_process = get_files_to_process()
    file_string = '\n'.join(files_to_process)
    print(f'Processing: {file_string}')
    process(files_to_process)

if __name__ == '__main__':
    import argparse
    import os
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', type=str, help='file with results to use')
    # get dataset arg
    args = parser.parse_args()
    input_fname = args.input

    if input_fname == 'all':
        process_all()
    else:
        # save_fname is same name, but replace .pkl with _processed.pkl
        save_fname = input_fname.replace('.pkl', '_processed.pkl')

        # process
        Postprocessor(input_fname, save_fname)
```

1.9.3 Post-processing the data files

Finally, the resulting data files are post-processed to extract the relevant statistics to be used for plotting.

This is the main file, called "extract.py":

```
import pandas as pd
import numpy as np
from analysis import get_sorted_templates, compare_per_template
from tqdm import tqdm
import os
from pdb import set_trace as breakpoint

datasets = [
    'anes2016_if_none',
    'anes2016_if_age',
    'anes2016_if_church_goer',
    'anes2016_if_discuss_politics',
    'anes2016_if_gender',
```

```
    'anes2016_if_ideology',
    'anes2016_if_party',
    'anes2016_if_patriotism',
    'anes2016_if_political_interest',
    'anes2016_if_race',
    'anes2016_if_state',

    'anes2016_if_party_ideo',  # drop two

    'anes2016_if_party_only',
    'anes2016_if_state_only',
    'anes2016_if_age_only',
    'anes2016_if_church_goer_only',
    'anes2016_if_discuss_politics_only',
    'anes2016_if_gender_only',
    'anes2016_if_ideology_only',
    'anes2016_if_patriotism_only',
    'anes2016_if_political_interest_only',
    'anes2016_if_race_only',

    'anes2016_if_back_only',  # no backstory at all

    'anes2016_if_five_only_only',  # only the five most useful (?)
]

#models = ['j1-jumbo', 'gpt3-davinci', 'gpt3-curie', 'j1-large', 'gpt3-babbage', 'gpt3-ada',
#           'gpt-j', 'gpt-neo-2.7B', 'gpt-neo-1.3B', 'gpt-neo-125M', 'gpt2-xl', 'gpt2-large',
#           'gpt2-medium']
models = ['gpt3-davinci']

model_map = {
    'j1-jumbo': 'Jurassic: 178B',
    'gpt3-davinci': 'GPT-3: 175B',
    'gpt3-curie': 'GPT-3: 13B',
    'j1-large': 'Jurassic: 7.5B',
    'gpt3-babbage': 'GPT-3: 6.7B',
    'gpt3-ada': 'GPT-3: 2.7B',
    'gpt-j': 'GPT-J: 6B',
    'gpt-neo-2.7B': 'GPT-Neo: 2.7B',
    'gpt-neo-1.3B': 'GPT-Neo: 1.3B',
    'gpt-neo-125M': 'GPT-Neo: 125M',
    'gpt2-xl': 'GPT-2: 1.5B',
    'gpt2-large': 'GPT-2: 774M',
    'gpt2-medium': 'GPT-2: 355M',
    'gpt2': 'GPT-2: 124M',
}

# B is billion, M is million
param_counts = {
    'j1-jumbo': 178e9,
    'gpt3-davinci': 175e9,
    'gpt3-curie': 13e9,
    'j1-large': 7.5e9,
    'gpt3-babbage': 6.7e9,
    'gpt3-ada': 2.7e9,
    'gpt-j': 6e9,
    'gpt-neo-2.7B': 2.7e9,
    'gpt-neo-1.3B': 1.3e9,
    'gpt-neo-125M': 125e6,
    'gpt2-xl': 1.5e9,
    'gpt2-large': 774e6,
    'gpt2-medium': 355e6,
```

```
    'gpt2': 124e6,
}

# to model type
model_type = {
    'j1-jumbo': 'Jurassic',
    'gpt3-davinci': 'GPT-3',
    'gpt3-curie': 'GPT-3',
    'j1-large': 'Jurassic',
    'gpt3-babbage': 'GPT-3',
    'gpt3-ada': 'GPT-3',
    'gpt-j': 'GPT-J',
    'gpt-neo-2.7B': 'GPT-Neo',
    'gpt-neo-1.3B': 'GPT-Neo',
    'gpt-neo-125M': 'GPT-Neo',
    'gpt2-xl': 'GPT-2',
    'gpt2-large': 'GPT-2',
    'gpt2-medium': 'GPT-2',
    'gpt2': 'GPT-2',
}

dataset_map = {
    'anes2016_if_none': 'ANES 2016 - None',
    'anes2016_if_age': 'ANES 2016 - Age',
    'anes2016_if_church_goer': 'ANES 2016 - Church Goer',
    'anes2016_if_discuss_politics': 'ANES 2016 - Discusses politics',
    'anes2016_if_gender': 'ANES 2016 - Gender',
    'anes2016_if_ideology': 'ANES 2016 - Ideology',
    'anes2016_if_party': 'ANES 2016 - Party',
    'anes2016_if_patriotism': 'ANES 2016 - Patriotism',
    'anes2016_if_political_interest': 'ANES 2016 - Political interest',
    'anes2016_if_race': 'ANES 2016 - Race',
    'anes2016_if_state': 'ANES 2016 - State',

    'anes2016_if_party_ideo': 'ANES 2016 - Party+Ideo',

    'anes2016_if_party_only': 'ANES 2016 + Party ONLY',
    'anes2016_if_state_only': 'ANES 2016 + State ONLY',
    'anes2016_if_age_only': 'ANES 2016 + Age ONLY',

    'anes2016_if_church_goer_only': 'ANES 2016 + Church Goer ONLY',
    'anes2016_if_discuss_politics_only': 'ANES 2016 + Discusses Politics ONLY',
    'anes2016_if_gender_only': 'ANES 2016 + Gender ONLY',
    'anes2016_if_ideology_only': 'ANES 2016 + Ideology ONLY',
    'anes2016_if_patriotism_only': 'ANES 2016 + Patriotism ONLY',
    'anes2016_if_political_interest_only': 'ANES 2016 + Political Interest ONLY',
    'anes2016_if_race_only': 'ANES 2016 + Race ONLY',

    'anes2016_if_back_only': 'ANES 2016 + NO BACKSTORY',

    'anes2016_if_five_only_only': 'ANES 2016 + Five only',
}

def check_files_present():
    '''
    Check if all files are present.
    '''
    present = True
    for dataset in datasets:
        for model in models:
            file_name = get_file(dataset, model)
            if file_name is None:
```

```
                breakpoint()
                present = False
                print(f'No file found for {model} on {dataset}')
    if present:
        print('All files present')
    else:
        raise Exception('Some files missing')

def get_file(dataset, model):
    path = f'../data/{dataset}'
    # get all filenames in path
    files = os.listdir(path)
    try:
        # get the file with the model name in it AND '_processed.pkl' in it
        files = [f for f in files if model in f and '_processed.pkl' in f]
        # if model is gpt2, filter so 'gpt2_' is only thing included
        if model == 'gpt2':
            breakpoint()
            files = [f for f in files if 'gpt2_' in f]
        file_path = files[0]
        # if length of files is more than 1, print the files and raise a warning
        if len(files) > 1:
            print(f'Multiple files found for {model} on {dataset}')
            print(files)
            print()
    except:
        # if no file found, return None and raise warning
        print(f'No file found for {model} on {dataset}')
        return None
    return os.path.join(path, file_path)

def prep_scatter():
    '''
    For each dataset and model, get the file and read in the df. Then, aggregate by 'template_name'
    and take the mean of 'accuracy' and 'mutual_inf' columns.
    '''
    print('Prepping data file for plots')
    loop = tqdm(total=len(datasets) * len(models))
    # make empty df with points. Columns are models, rows are datasets
    dataset_dicts = []
    for dataset in datasets:
        model_dicts = []
        for model in models:
            file_name = get_file(dataset, model)
            print(file_name)
            exp_df = pd.read_pickle(file_name)
#             if dataset == 'anes2016':
                # keep only where ground_truth == 'clinton' or 'trump'
            exp_df = exp_df[(exp_df['ground_truth'] == 'clinton') | (exp_df['ground_truth'] == 'trump')]
#             elif dataset == 'anes2020':
#                 # keep only where ground_truth == 'biden' or 'trump'
#                 exp_df = exp_df[(exp_df['ground_truth'] == 'biden') | (exp_df['ground_truth'] == 'trump')]
#             elif dataset == 'anes2012':
#                 # keep only where ground_truth == 'obama' or 'romney'
#                 exp_df = exp_df[(exp_df['ground_truth'] == 'obama') | (exp_df['ground_truth'] == 'romney')]
            # aggregate by 'template_name' and take the mean of 'accuracy' and 'mutual_inf' columns
            exp_df = exp_df.groupby('template_name').agg({
              'accuracy': np.mean,
              'correct_weight': np.mean,
              'mutual_inf': np.mean,
              'version': 'count'})
            # change version to count
```

```
            exp_df.rename(columns={'version': 'count'}, inplace=True)
            # add param_count column
            exp_df['param_count'] = param_counts[model]
            # add model_type column
            exp_df['model_type'] = model_type[model]
            # make 'template_name' (index) a column
            # exp_df.reset_index(inplace=True)
            # add to df
            model_dicts.append(exp_df)
            # increment loop
            loop.update(1)
        # add
        dataset_dicts.append(model_dicts)
    dataset_names = [dataset_map[d] for d in datasets]
    model_names = [model_map[m] for m in models]
    # make df with datasets as rows and models as columns
    df = pd.DataFrame(dataset_dicts, index=dataset_names, columns=model_names)
    # save to ../data/plot_data.pkl
    df.to_pickle('../data/pa/plot_data.pkl')
    print('Saved to ../data/pa/plot_data.pkl')

if __name__ == '__main__':
    check_files_present()
    prep_scatter()
```

which depends on "analysis.py":

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from sklearn.linear_model import LinearRegression, LogisticRegression

# sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# TODO - do correlation analysis between templates and ground truth. Add in functions for this?
def compare_per_template(df):
    group = df.groupby(by='template_name')

    output_df = group[['accuracy', 'mutual_inf']].agg(np.mean)

    corr = output_df.corr().iloc[0,1]

    x, y = output_df.mutual_inf.values, output_df.accuracy.values

    plt.scatter(
        x=x,
        y=y,
        alpha=0.7,
        s=50,
        edgecolors='none',
    )

    # fit linear regression
    lr = LinearRegression()
    a, b = lr.fit(x.reshape(-1,1), y).coef_[0], lr.intercept_
    # plot line
    x_linspace = np.linspace(x.min(), x.max(), 100)
    plt.plot(x_linspace, a*x_linspace + b, 'C1', alpha=0.7)

    plt.title(f'Grouped by Template, Corr Coeff: {corr:.3f}')
```

```
    plt.xlabel(r'Mutual Information: $I(Y, f_{\theta}(X))$')
    plt.ylabel('Accuracy')

    return corr

def compare_per_response(df, y_jitter=.05):
    corr = df[['accuracy', 'mutual_inf']].corr().iloc[0,1]

    x, y = df.mutual_inf.values, df.accuracy.values

    plt.scatter(
        x=x,
        y=y + np.random.normal(0, y_jitter, len(y)),
        alpha=0.2,
        s=20,
        edgecolors='none',
    )

    # fit logistic regression
    lr = LogisticRegression()
    a, b = lr.fit(x.reshape(-1,1), y).coef_[0], lr.intercept_
    # plot sigmoid regression
    x_linspace = np.linspace(x.min(), x.max(), 100)
    plt.plot(x_linspace, sigmoid(a*x_linspace + b), 'C1', alpha=0.7)

    plt.title(f'Entropy Difference vs. Response Accuracy, Corr Coeff: {corr:.3f}')
    plt.xlabel(r'Entropy Difference: $H(Y) - H(Y|f_{\theta}(x_i))$')
    plt.ylabel('Accuracy')

    return corr


def compare_per_response_weight(df):
    corr = df[['correct_weight', 'mutual_inf']].corr().iloc[0,1]

    x, y = df.mutual_inf.values, df.correct_weight.values

    # unique_template_names = list(set(df.template_name.values))
    # template_name_map = dict(zip(unique_template_names,
    #                              range(len(unique_template_names))))

    plt.scatter(
        x=x,
        y=y,
        alpha=0.2,
        s=20,
        edgecolors='none',
        # c=[template_name_map[v] for v in df.template_name.values]
    )

    # fit linear regression
    lr = LinearRegression()
    a, b = lr.fit(x.reshape(-1,1), y).coef_[0], lr.intercept_
    # plot line
    x_linspace = np.linspace(x.min(), x.max(), 100)
    plt.plot(x_linspace, a*x_linspace + b, 'C1', alpha=0.7)

    plt.title(f'Weight of Correct Response, Corr Coeff: {corr:.3f}')
    plt.xlabel(r'Entropy Difference: $H(Y) - H(Y|f_{\theta}(x_i))$')
    plt.ylabel('Weight on Correct')
```

```
    return corr

def compare_per_idx(df):
    group = df.groupby(by='raw_idx')

    output_df = group[['accuracy', 'mutual_inf']].agg(np.mean)

    corr = output_df.corr().iloc[0,1]

    x, y = output_df.mutual_inf.values, output_df.accuracy.values

    plt.scatter(
        x=x,
        y=y,
        alpha=0.7,
        s=50,
        edgecolors='none',
    )

    # fit linear regression
    lr = LinearRegression()
    a, b = lr.fit(x.reshape(-1,1), y).coef_[0], lr.intercept_
    # plot line
    x_linspace = np.linspace(x.min(), x.max(), 100)
    plt.plot(x_linspace, a*x_linspace + b, 'C1', alpha=0.7)


    plt.title(f'Grouped by Instance, Corr Coeff: {corr:.3f}')
    plt.xlabel(r'Mean Entropy Difference: $\mathbb{E}_{\theta}[H(Y) - H(Y|f_{\theta}(x_i))]$')
    plt.ylabel('Accuracy')

    return corr

def plot_comparisons(df, show=True, save=False, filename=None):
    """
    Calculates four different comparisons and shows or saves the results based
    on user input.
    """
    corrs = {}
    # # make figure big
    # plt.figure(figsize=(14,6))

    # plt.subplot(121)
    # corrs['per_template'] = compare_per_template(df)

    # plt.subplot(122)
    # corrs['per_response_weight'] = compare_per_response_weight(df)

    plt.figure(figsize=(14,8))
    plt.subplot(221)
    corrs['per_template'] = compare_per_template(df)

    plt.subplot(222)
    corrs['per_response'] = compare_per_response(df)

    plt.subplot(223)
    corrs['per_response_weight'] = compare_per_response_weight(df)

    plt.subplot(224)
    corrs['per_id'] = compare_per_idx(df)

    # make suptitle with dataset
```

```
    plt.suptitle(f'{df.dataset.unique()[0].upper()} - {df.model.unique()[0].upper()}')

    plt.tight_layout()


    if save:
        if filename is None:
            raise ValueError('filename needs to be specified if save is True')
        plt.savefig(filename)
    if show:
        plt.show()
    else:
        plt.cla()

    return corrs

def get_sorted_templates(df):
    group = df.groupby(by='template_name')

    # agg accuracy and conditional entropy by mean, and prompt by first
    output_df = group.agg({
        'accuracy': 'mean',
        'mutual_inf': 'mean',
        'coverage': 'mean',
        'prompt': 'first',
    })

    # sort by conditional entropy
    output_df = output_df.sort_values(by='mutual_inf', ascending=True)
    return output_df


if __name__ == '__main__':
    import argparse
    import os
    parser = argparse.ArgumentParser()
    parser.add_argument('--results', type=str, help='file with results to use')
    # get dataset arg
    args = parser.parse_args()
    results_file = args.results

    df = pd.read_pickle(args.results)

    templates = get_sorted_templates(df)
    print(templates)
    # calculate mutual information

    # get model and dataset
    model = df.model.unique()[0]
    dataset = df.dataset.unique()[0]
    # make 'plots' if missing
    if not os.path.exists('../plots'):
        os.mkdir('../plots')
    # save to plots/dataset_model
    model = model.replace('/', '-')
    plot_comparisons(df, save=True, filename=f'../plots/{dataset}_{model}.png')
    pass
```

1.10 Study 3: Second Order Correlations - ANES 2020 Template

To replicate Study 3, an entire set of ANES data must be generated for each target demographic variable of interest; we used the following shell script:

```
#!/bin/bash

python ./openai_test_interview.py gender
python ./openai_test_interview.py race
python ./openai_test_interview.py age
python ./openai_test_interview.py education
python ./openai_test_interview.py patriotism
python ./openai_test_interview.py discuss_politics
python ./openai_test_interview.py political_interest
python ./openai_test_interview.py ideology
python ./openai_test_interview.py pid7
python ./openai_test_interview.py votechoice_2016
python ./openai_test_interview.py voted_2016
python ./openai_test_interview.py church_goer
```

which then calls the following main script. This generates each "interview" and then queries GPT-3:

```
import sys
import numpy as np
import openai
import time
import scipy.stats as stats
import pandas as pd
import pickle
from tqdm import tqdm

# for cost analysis
from transformers import GPT2Tokenizer

from common import *  # reuse from study2 code

openai.api_key = "PUT YOUR KEY HERE"

age_map = {}
for ind in range(100):
    age_map[ind]=str(ind)

questions = {

    'V161342': {
        'desc':'gender',
        'vals': {1:'male',2:'female'},
        'question': 'Interviewer: What is your gender? Are you "male" or "female"?',
        },

    'V161310x': {
        'desc':'race',
        'vals': {1:'white',2:'black',3:'asian',5:'hispanic'},
        'question': 'Interviewer: I am going to read you a list of four race categories. What race do you ' + \
                    'consider yourself to be? "White", "Black", "Asian", or "Hispanic"?',
        },
```

```
'V161267': {
    'desc':'age',
    'vals': age_map,
    'question': 'Interviewer: What is your age in years?',
    },

'V161270': {
    'desc':'education',
    'vals': {
        1:'high school',
        2:'high school',
        3:'high school',
        4:'high school',
        5:'high school',
        6:'high school',
        7:'high school',
        8:'high school',
        9:'high school',
        10:'some college',
        11:'some college',
        12:'some college',
        13:'a four-year college degree',
        14:'an advanced degree',
        15:'an advanced degree',
        16:'an advanced degree',
    },
    'question': 'Interviewer: What is the highest level of school you have completed, or the highest ' + \
                'degree you have received? Is it "high school", "some college", "a four-year college ' + \
                'degree", or "an advanced degree"?',
    },

'V161244': {
    'desc':'church_goer',
    'vals': {1:'yes',2:'no'},
    'question': 'Interviewer: Lots of things come up that keep people from attending religious ' + \
                'services even if they want to. Thinking about your life these days, do you ever' + \
                'attend religious services? Please respond with "yes" or "no".',
    },

'V162125x': {
    'desc':'patriotism',
    'vals': {1:"extremely good", 2:"moderately good", 3:"a little good",
             4:"neither good nor bad", 5:"a little bad", 6:"moderately bad", 7:"extremely bad"},
    'question': 'Interviewer:  When you see the American flag flying, how does it make you feel? Does ' + \
                'it make you feel "extremely good", "moderately good", "a little good", "neither good ' + \
                'nor bad", "a little bad", "moderately bad", or "extremely bad"?'
    },

'V162174': {
    'desc':'discuss_politics',
    'vals': {1:'yes',2:'no'},
    'question': 'Interviewer: Do you ever discuss politics with your family and friends? Please respond ' + \
                'with "Yes" or "No".'
    },

'V162256': {
    'desc':'political_interest',
    'vals': {1:"very interested", 2:"somewhat interested", 3:"not very interested", 4:"not at all interested"},
    'question': 'Interviewer: How interested would you say you are in politics? Are you "very interested", ' + \
                '"somewhat interested", "not very interested", or "not at all interested"?',
    },
```

```
    'V161126': {
        'desc':'ideology',
        'vals': {1:"extremely liberal", 2:"liberal", 3:"slightly liberal",
                 4:"moderate", 5:"slightly conservative", 6:"conservative", 7:"extremely conservative"
        },
        'question': 'Interviewer: When asked about your political ideology, would you say you are "extremely '+ \
                    'liberal", "liberal", "slightly liberal", "moderate", "slightly conservative", ' + \
                    '"conservative", or "extremely conservative"?',
        },

    'V161155': {
        'desc':'pid3',
        'vals': {1:'Democrat',2:'Republican',3:'Independent'},
        'question': 'Interviewer: Generally speaking, do you usually think of yourself as a "Democrat", a ' + \
                    '"Republican", or an "Independent"?',
        },

    'V161158x': {
        'desc':'pid7',
        'vals': {1:"strong democrat", 2:"not very strong democrat",
                 3:"independent, but closer to the Democratic party", 4:"independent",
                 5:"independent, but closer to the Republican party", 6:"not very strong Republican",
                 7:"strong Republican"},
        'question': 'Interviewer: Which would you say best describes your partisan identification. ' + \
                    'Would you say you are a "strong democrat", "not very strong democrat", ' + \
                    '"independent, but closer to the Democratic party", "independent", "independent, ' + \
                    'but closer to the Republican party", "not very strong Republican", or "strong Republican"?',
        },

    'V162031x': {
        'desc':'voted_2016',
        'vals': {0:'no',1:'yes'},
        'question': 'Interviewer: Did you vote in the 2016 general election? Please answer with "yes" or "no".'
        },

    'V162062x': {
        'desc':'votechoice_2016',
        'vals': {1:"Hillary Clinton", 2:"Donald Trump", 42:"someone else"},
        'question': 'Interviewer: Which presidential candidate did you vote for in the 2016 presidential ' + \
        'election, "Hillary Clinton", "Donald Trump", or "someone else"?',
        },

    }

#
# ============================================================================================
#

def render_question( s, q, last_q = False ):
    txt = ''

    if q == 'V161155':

        if s['V161155'] == 2:
            txt += 'Interviewer: Thinking about your identification with the Republican party, would you say ' + \
                   'it is "strong" or a "not very strong"?\n'
            if last_q:
                txt += f"Me:"
                return txt

            if s['V161156'] == 1:
                txt += f"Me: strong\n\n"
```

```
            elif s['V161156'] == 2:
                txt += f"Me: not very strong\n\n"

        if s['V161155'] == 1:
            txt += 'Interviewer: Thinking about your identification with the Democratic party, would you say ' + \
                   'it is "strong" or a "not very strong"?\n'
            if last_q:
                txt += f"Me:"
                return txt

            if s['V161156'] == 1:
                txt += f"Me: strong\n\n"
            elif s['V161156'] == 2:
                txt += f"Me: not very strong\n\n"

        if s['V161155'] == 'Independent':
            txt += 'Interviewer: Do you think of yourself as "closer to the Republican Party", "closer to the ' + \
                   'Democratic party", or "closer to neither party"?\n
'
            if last_q:
                txt += f"Me:"
                return txt

            if s['V161157'] == 1:
                txt += "Me: closer to the Republican Party\n\n"
            elif s['V161157'] == 2:
                txt += "Me: closer to neither party\n\n"
            elif s['V161157'] == 3:
                txt += "Me: closer to the Democratic party\n\n"


    # XXX note, we don't check to make sure the ANES answer is "valid"
    if last_q:
        txt += questions[q]['question'] + "\n"
        txt += f"Me:"
        return txt

    if s[q] in questions[q]['vals']:
        txt += questions[q]['question'] + "\n"
        txt += f"Me: {questions[q]['vals'][ s[q] ]}\n\n"
        return txt
    else:
        return txt

def find_q( questions, hrq ):
    for q in questions.keys():
        if questions[q]['desc'] == hrq:
            return q
    error("not found!")

def build_interview( s, human_readable_omit=None ):
    txt = ''

    human_readable_question_order = [ 'gender', 'race', 'age', 'education', 'church_goer',
              'patriotism', 'discuss_politics', 'political_interest', 'ideology', 'pid7',
              'voted_2016', 'votechoice_2016' ]

    omit = None
    if human_readable_omit:
        omit = find_q( questions, human_readable_omit )

    for hrq in human_readable_question_order:
```

```
        q = find_q( questions, hrq )
        if q == omit:
            continue
        if hrq=='votechoice_2016' and s['V162031x'] == 0:
            continue

        txt += render_question( s, q, last_q=False )

    if human_readable_omit:
        txt += render_question( s, omit, last_q=True )

    return txt

#
# ============================================================================================
# ============================================================================================
#

def cost_approximation(prompt, engine="davinci", tokenizer=None):
    possible_engines = ["davinci", "curie", "babbage", "ada"]
    assert engine in possible_engines, f"{engine} is not a valud engine"

    if tokenizer==None:
        tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

    num_tokens = len(tokenizer(prompt)['input_ids'])

    if engine == "davinci":
        cost = (num_tokens / 1000) * 0.0600
    elif engine == "curie":
        cost = (num_tokens / 1000) * 0.0060
    elif engine == "babbage":
        cost = (num_tokens / 1000) * 0.0012
    else:
        cost = (num_tokens / 1000) * 0.0008

    return cost, num_tokens

#
# ============================================================================================
# ============================================================================================
#

def strcompare( s1, s2 ):
    s1 = s1.lower().strip()
    s2 = s2.lower().strip()
    return s1.startswith(s2) or s2.startswith(s1)

print( "=================================================" )
print( "RUNNING WITH " + sys.argv[1] )
print( "=================================================" )


df = pd.read_csv( '../anes2016/anes_timeseries_2016_rawdata.txt', sep='|')

final_results = []

#hr_omit = 'votechoice_2016'
#hr_omit = 'church_goer'
hr_omit = sys.argv[1]

omit = find_q( questions, hr_omit )
```

```
costs = []
numtoks = []
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

for ind, row in tqdm( df.iterrows() ):
    #print( "================================================\n" )

    id = row['V160001_orig']

    prompt = build_interview( row,  human_readable_omit=hr_omit )

    cost, numtok = cost_approximation( prompt, engine="davinci", tokenizer=tokenizer )
    costs.append( cost )
    numtoks.append( numtok )

    #print(prompt )
    full_results = do_query( prompt, max_tokens = 5, temperature=0.001 )

    samp_response = full_results[1]['choices'][0]['text']

    #clean_r = samp_response.replace('\n',' ')
    #print( f"{questions['V161158x']['vals'][row['V161158x']]} -> {clean_r}", end='' )

    coded_response = -1
    for valnum in questions[omit]['vals'].keys():
        if strcompare( questions[omit]['vals'][valnum], samp_response ):
            #print( f" -> {questions[omit]['vals'][valnum]} -> {valnum}", end='' )
            coded_response = valnum
    #print('')

    final_results.append( {
      "id":id,
      "prompt":prompt,
      "sampled_response":samp_response,
      "coded_response":coded_response,
      "full_results":full_results} )

print( "Total cost: ", np.sum(np.array(costs)) )
print( "Averge numtok: ", np.mean(np.array(numtoks)) )

pickle.dump( final_results, open("./heatmap_backstory_" + hr_omit + "_full_results.pkl","wb") )

# ------------------

fout = open( "./heatmap_" + hr_omit + "_results.csv", "w" )

keys = ['V160001_orig'] + list(questions.keys())

print( ",".join(keys) + ",gpt3_coded_response", file=fout )

for ind, row in tqdm( df.iterrows() ):
    print( ",".join( [str(row[k]) for k in keys] ) + f",{final_results[ind]['coded_response']}", file=fout )

fout.close()
```