Assignment: Voter classification using exit poll data

TODO: Edit this cell to fill in your NYU Net ID and your name:

- **Net ID:** [redacted]
- **Name:** [redacted]

In this notebook, we will explore the problem of voter classification.

Given demographic data about a voter and their opinions on certain key issues, can we predict their vote in the 2016 U.S. presidential election? We will attempt this using a K nearest neighbor classifier.

In the first part of this notebook, I will show you how to train and use a K nearest neighbors classifier for this task. In the next part of the notebook, you will try to improve the basic model for better performance.

Import libraries

In [438]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm

from sklearn.model_selection import ShuffleSplit
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics.pairwise import nan_euclidean_distances
```

We will need to install a library that is not in the default Colab environment, which we can install with `pip`**:**

In [439]:

```
!pip install category_encoders
```

```
Requirement already satisfied: category_encoders in /usr/local/lib/python3.7/dist-package
s (2.2.2)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.7/dist-pack
ages (from category_encoders) (0.22.2.post1)
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.7/dist-packag
es (from category_encoders) (0.10.2)
Requirement already satisfied: patsy>=0.5.1 in /usr/local/lib/python3.7/dist-packages (fr
om category_encoders) (0.5.1)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.7/dist-packages (f
rom category_encoders) (1.19.5)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.7/dist-packages (fr
om category_encoders) (1.4.1)
Requirement already satisfied: pandas>=0.21.1 in /usr/local/lib/python3.7/dist-packages (
from category_encoders) (1.1.5)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages (fr
om pandas>=0.21.1->category_encoders) (2018.9)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-pa
ckages (from pandas>=0.21.1->category_encoders) (2.8.1)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from patsy>
=0.5.1->category_encoders) (1.15.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (fr
om scikit-learn>=0.20.0->category_encoders) (1.0.1)
```

In [440]:

```
import category_encoders as ce
```

Load data

The data for this notebook comes from the [U.S. National Election Day Exit Polls]().

Here's a brief description of how exit polls work.

Exit polls are conducted by Edison Research on behalf of a consortium of media organizations.

First, the member organizations decide what races to cover, what sample size they want, what questions should be asks, and other details. Then, sample precincts are selected, and local interviewers are hired and trained. Then, at those precincts, the local interviewer approaches a subset of voters as they exit the polls (for example, every third voter, or every fifth voter, depending on the required sample size).

When a voter is approached, they are asked if they are willing to fill out a questionnaire. Typically about 40-50% agree. (For those that decline, the interviewer visually estimates their age, race, and gender, and notes this information, so that the response rate by demographic is known and responses can be weighted accordingly in order to be more representative of the population.)

Voters that agree to participate are then given an form with 15-20 questions. They fill in the form (anonymously), fold it, and put it in a small ballot box.

Three times during the day, the interviewers will stop, take the questionnaires, compile the results, and call them in to the Edison Research phone center. The results are reported immediately to the media organizations that are consortium members.

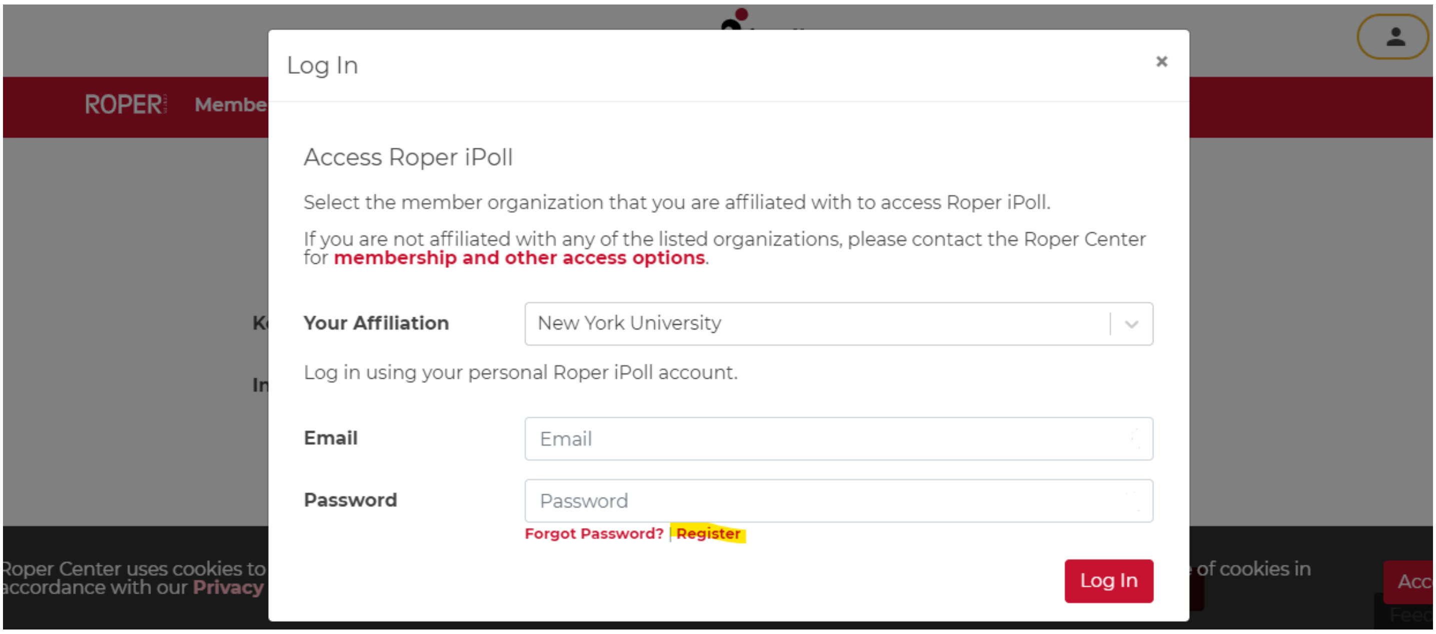
In addition to the poll of in-person voters, absentee and early voters (who are not at the polls on Election Day) are surveyed by telephone.

The exit poll data is not freely available on the web, but is available to those with institutional membership. You will be able to use your NYU email address to create an account with which you can download the exit poll data.

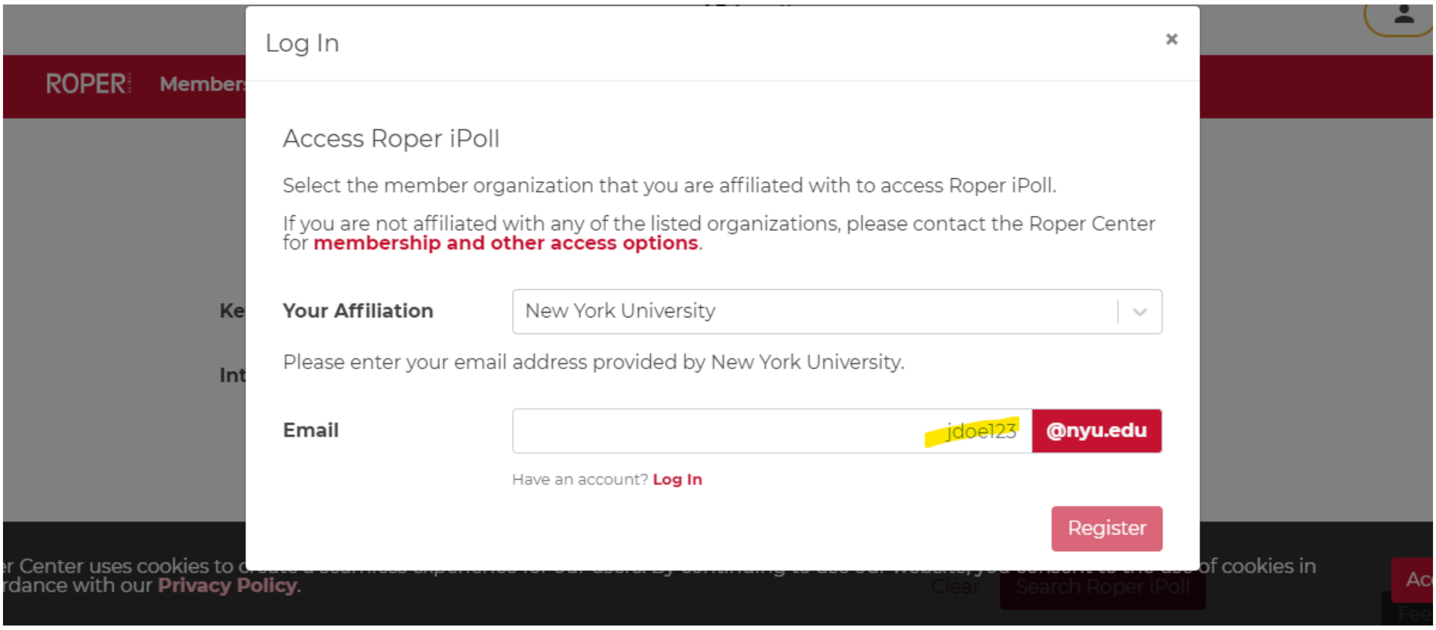
To get the data, visit the Roper Center website. Click on the user icon in the top right of the page, and choose "Log in".

For "Your Affiliation", choose "New York University".

Then, choose "Register" (highlighted in yellow below).



Enter your NYU email address (highlighted in yellow below) and then click "Register".

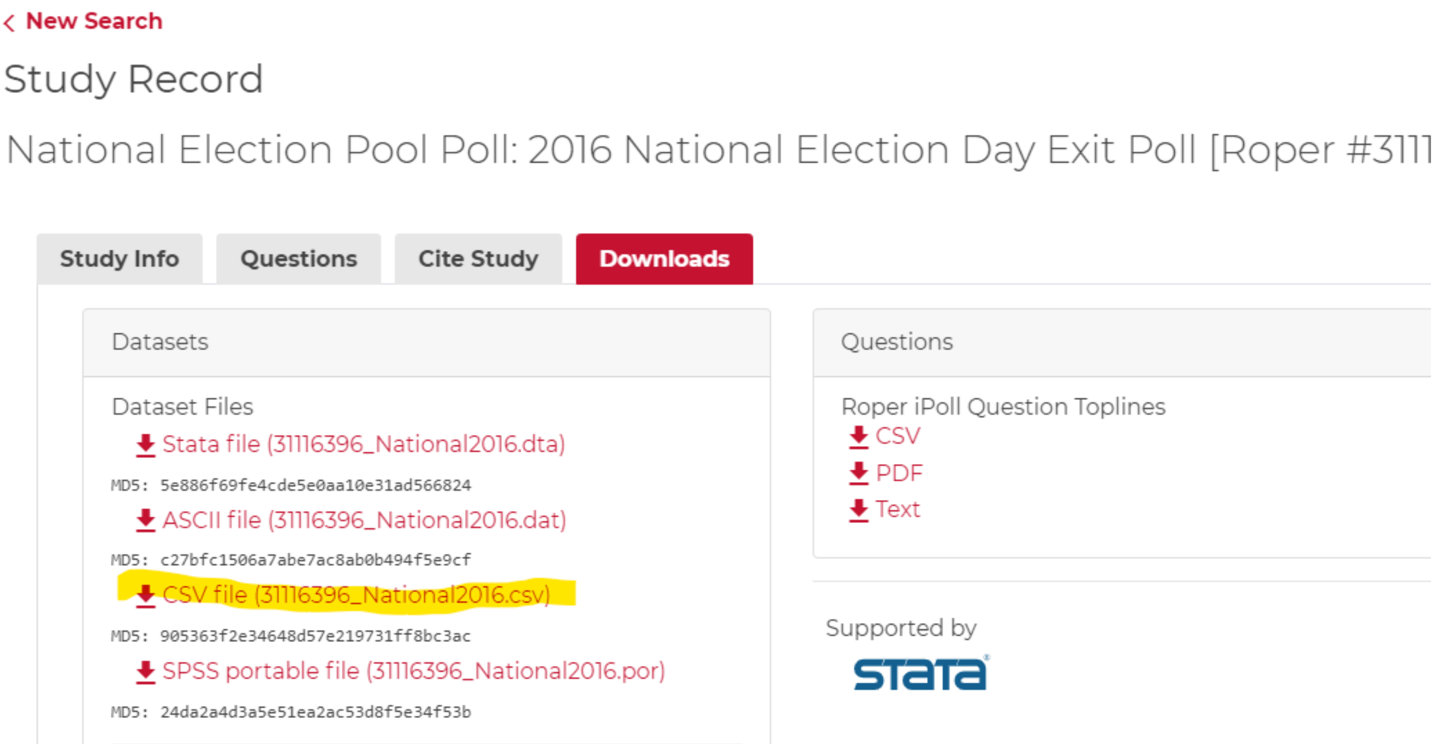


You will get an email at your NYU email address with the subject "Roper iPoll Account Registration". Open the email and click "Confirm Account" to create a password and finish your account registration.

Once you have completed your account registration, log in to Roper iPoll by clicking the user icon in the top right of the page, choosing "Log in", and entering your NYU email address and password.

**Then, open the Study Record for the [2016 National Election Day Exit Poll](). **

Click on the "Downloads" tab, and then click on the CSV data file (highlighted in the image below).



Accept the terms (click "Accept terms") and the file will be downloaded to your computer.

After you download the CSV file, scroll down a bit until you see the "Study Documentation, Questionnaire and Codebooks" PDF file. Download this file as well.

To get the data into Colab, run the following cell. Upload the CSV file you just downloaded (`31116396_National2016.csv`) to your Colab workspace. Wait until the uploaded has completely finished - it may take a while, depending on the quality of your network connection.

In [441]:

```
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

Choose File **No file selected**

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving 31116396_National2016.csv to 31116396_National2016 (2).csv
User uploaded file "31116396_National2016.csv" with length 26283642 bytes
```

Then, use the `read_csv` function in `pandas` to read in the file.

Also use `head` to view the first few rows of data and make sure that everything is read in correctly.

In [442]:

```
df = pd.read_csv('31116396_National2016.csv')
df.head()
```

Out[442]:

	ID	PRES	HOU	WEIGHT	@2WAYPRES16	AGE	AGE3	AGE8	AGE45	AGE49	AGE60	AGE65	AGEBLACK	A
0	135355	Hillary Clinton	The Democratic candidate	6.530935		18-29	18-29	18-24	18-44	18-49	18-29	18-24	Non-Black 18-29	I
1	135356	Hillary Clinton	The Democratic candidate	6.479016		18-29	18-29	25-29	18-44	18-49	18-29	25-29	Non-Black 18-29	I
2	135357	Hillary Clinton	The Democratic candidate	8.493230		30-44	30-59	30-39	18-44	18-49	30-44	30-39	Non-Black 30-44	
3	135358	Hillary Clinton	The Democratic candidate	3.761814		30-44	30-59	30-39	18-44	18-49	30-44	30-39	Non-Black 30-44	
4	135359	Hillary Clinton	The Democratic candidate	3.470473		45-65	30-59	45-49	45+	18-49	45-59	40-49	Black 45-59	

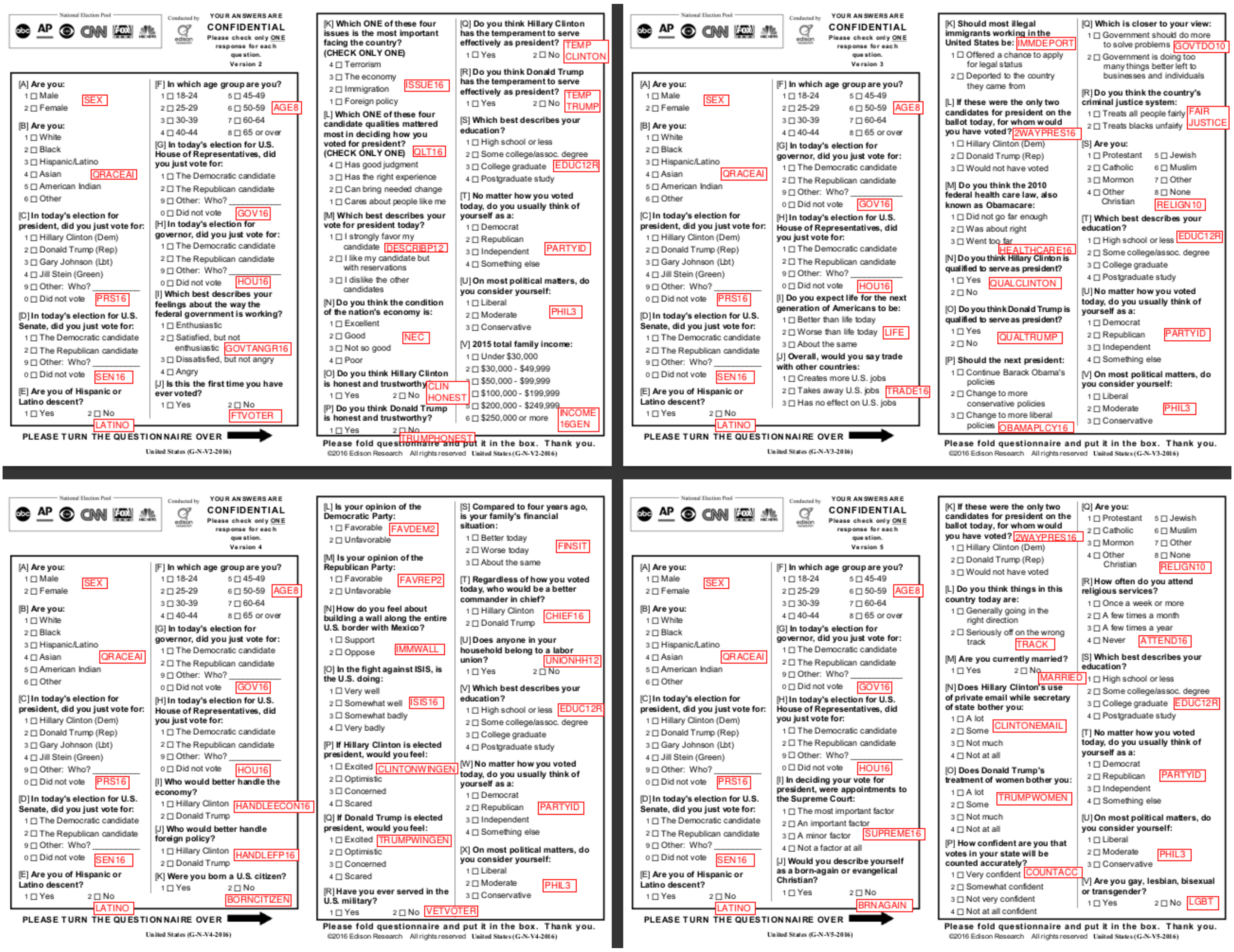
Prepare data

Survey data can be tricky to work with, because surveys often "branch"; the questions that are asked depends on a respondent's answers to other questions.

In this case, different respondents fill out different versions of the survey.

Review pages 7-11 of the "Study Documentation, Questionnaire, and Codebooks" PDF file you downloaded

earlier, which shows the five different questionnaire versions used for the 2016 exit polls.



In [443]:

```
df['VERSION'].value_counts()
```

Out[443]:

```
Version 2    5126
Version 1    5094
Version 3    4980
Version 4    4919
Version 5    4915
Name: VERSION, dtype: int64
```

In a red box next to each question, you can also see the name of the variable (column name) that the respondent's answer will be stored in.

Because each respondent answers different questions, for each row in the data, only some of the columns - the columns corresponding to questions included in that version of the survey - have data.

Missing data

Since each respondent only saw a subset of questions, we expect to see missing values in each column.

However, if we look at the **count** of values in each column, we see that there are no missing values - every column has the full count!

In [444]:

```
df.describe(include='all')
```

Out[444]:

	ID	PRES	HOU	WEIGHT	@2WAYPRES16	AGE	AGE3	AGE8	AGE45	AGE49	AGE60	AGE
count	25034.000000	25034	25034	25034.000000	25034	25034	25034	25034	25034	25034	25034	250
unique	NaN	7	5	NaN	5	5	4	9	3	3	5	
top	NaN	Hillary Clinton	The Democratic candidate	NaN		45-65	30-59	50-59	45+	18-49	45-59	50
freq	NaN	12126	12041	NaN	15568	9746	13697	5071	14436	12836	7490	73
mean	188663.858712	NaN	NaN	1.003016	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
std	27829.369563	NaN	NaN	1.065169	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
min	135355.000000	NaN	NaN	0.047442	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
25%	175885.250000	NaN	NaN	0.525367	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
50%	193824.500000	NaN	NaN	0.745491	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
75%	210374.500000	NaN	NaN	1.031137	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
max	226680.000000	NaN	NaN	18.407688	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N

This is because missing values are recorded as a single space, and not with a NaN.

Let's change that:

In [445]:

```
df.replace(" ", float("NaN"), inplace=True)
```

Now we can see an accurate count of the number of responses in each column:

In [446]:

```
df.describe(include='all')
```

Out[446]:

	ID	PRES	HOU	WEIGHT	@2WAYPRES16	AGE	AGE3	AGE8	AGE45	AGE49	AGE60	AGE
count	25034.000000	24696	23970	25034.000000	9466	24853	24853	24853	24853	24853	24853	248
unique	NaN	6	4	NaN	4	4	3	8	2	2	4	
top	NaN	Hillary Clinton	The Democratic candidate	NaN	Hillary Clinton	45-65	30-59	50-59	45+	18-49	45-59	50
freq	NaN	12126	12041	NaN	4611	9746	13697	5071	14436	12836	7490	73
mean	188663.858712	NaN	NaN	1.003016	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
std	27829.369563	NaN	NaN	1.065169	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
min	135355.000000	NaN	NaN	0.047442	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
25%	175885.250000	NaN	NaN	0.525367	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
50%	193824.500000	NaN	NaN	0.745491	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
75%	210374.500000	NaN	NaN	1.031137	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
max	226680.000000	NaN	NaN	18.407688	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N

Notice that *every* row has some missing data! So, we can't just remove rows with missing data and work with

Notice that *every* row has some missing data! So, we can't just remove rows with missing data and work with the complete data.

Instead, we'll have to make sure that the classifier we use is able to work with partial data. One important benefit of K nearest neighbors is that it can work well with data that has missing values, as long as we can think of a distance metric that behaves reasonably under these conditions.

Encode target variable as a binary variable

Our goal is to classify voters based on their vote in the 2016 presidential election, i.e. the value of the `PRES` column. We will restrict our attention to the candidates from the two major parties, so we will throw out the rows representing voters who chose other candidates:

In [447]:

```
df = df[df['PRES'].isin(['Donald Trump', 'Hillary Clinton'])]
df.reset_index(inplace=True, drop=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22798 entries, 0 to 22797
Columns: 138 entries, ID to WPROTBRN3
dtypes: float64(1), int64(2), object(135)
memory usage: 24.0+ MB
```

In [448]:

```
df.head()
```

Out[448]:

	ID	PRES	HOU	WEIGHT	@2WAYPRES16	AGE	AGE3	AGE8	AGE45	AGE49	AGE60	AGE65	AGEBLACK	A
0	135355	Hillary Clinton	The Democratic candidate	6.530935	NaN	18-29	18-29	18-24	18-44	18-49	18-29	18-24	Non-Black 18-29	l
1	135356	Hillary Clinton	The Democratic candidate	6.479016	NaN	18-29	18-29	25-29	18-44	18-49	18-29	25-29	Non-Black 18-29	l
2	135357	Hillary Clinton	The Democratic candidate	8.493230	NaN	30-44	30-59	30-39	18-44	18-49	30-44	30-39	Non-Black 30-44	
3	135358	Hillary Clinton	The Democratic candidate	3.761814	NaN	30-44	30-59	30-39	18-44	18-49	30-44	30-39	Non-Black 30-44	
4	135359	Hillary Clinton	The Democratic candidate	3.470473	NaN	45-65	30-59	45-49	45+	18-49	45-59	40-49	Black 45-59	

In [449]:

```
df['PRES'].value_counts()
```

Out[449]:

```
Hillary Clinton     12126
Donald Trump        10672
```

```
Name: PRES, dtype: int64
```

Now, we will transform the string value into a binary variable, and save the result in `y`.

In [450]:

```
y = df['PRES'].map({'Donald Trump': 1, 'Hillary Clinton': 0})
y.value_counts()
```

Out[450]:

```
0    12126
1    10672
Name: PRES, dtype: int64
```

Get training and test indices

We'll be working with many different subsets of this dataset, including different columns.

So instead of splitting up the data into training and test sets, we'll get an array of training indices and an array of test indices using `ShuffleSplit`. Then, we can use these arrays throughout this notebook.

In [451]:

```
idx_tr, idx_ts = next(ShuffleSplit(n_splits = 1, test_size = 0.3, random_state = 3).spli
t(df['PRES']))
```

I specified the state of the random number generator for repeatability, so that every time we run this notebook we'll have the same split. This makes it easier to discuss specific examples.

Now, we can use the `pandas` function `.iloc` to get the training and test parts of the data set for any column.

For example, if we want the training subset of `y`:

In [452]:

```
y[idx_tr]
```

Out[452]:

```
1349     1
14642    0
18106    0
19171    1
17962    0
        ..
6400     1
15288    0
11513    0
1688     1
5994     0
Name: PRES, Length: 15958, dtype: int64
```

or the test subset of `y`:

In [453]:

```
y.iloc[idx_ts]
```

Out[453]:

```
21876    1
17297    0
19295    0
8826     1
11357    0
```

```
        ..
9144     0
4409     0
6320     0
7824     0
4012     1
Name: PRES, Length: 6840, dtype: int64
```

Encode ordinal features

Next, we need to encode our features. All of the features are represented as strings, but we will have to transform them into something over which we can compute a meaningful distance measure.

Columns that have a logical order should be encoded using ordinal encoding, so that the distance metric will be meaningful.

For example, consider the `AGE` column:

In [454]:

```
df['AGE'].unique()
```

Out[454]:

```
array(['18-29', '30-44', '45-65', '65+', nan], dtype=object)
```

In [455]:

```
df['AGE'].value_counts()
```

Out[455]:

```
45-65    9067
30-44    5526
65+      4398
18-29    3649
Name: AGE, dtype: int64
```

What if we transform the `AGE` column using four binary columns: `AGE_18-29`, `AGE_30-44`, `AGE_45-65`, `AGE_65+`, with a 0 or a 1 in each column to indicate the respondent's age?

If we did this, we would lose meaningful information about the distance between ages; a respondent whose age is 18-29 would have the same distance to one whose age is 45-65 as to one whose age is 65+.

Instead, we will use ordinal encoding, which will represent `AGE` in a single column with *ordered* integer values.

First, we create an `OrdinalEncoder`:

In [456]:

```
enc_ord = ce.OrdinalEncoder(handle_missing='return_nan')
```

Then, we `fit` it by passing the columns that we wish to encode as ordinal values:

In [457]:

```
enc_ord.fit(df['AGE'])
```

Out[457]:

```
OrdinalEncoder(cols=['AGE'], drop_invariant=False, handle_missing='return_nan',
               handle_unknown='value',
               mapping=[{'col': 'AGE', 'data_type': dtype('O'),
                         'mapping': 18-29    1
30-44    2
45-65    3
65+      4
```

```
NaN      -2
dtype: int64}],
               return_df=True, verbose=0)
```

Finally, we use the "fitted" encoder to `transform` the data, and we save the result in `df_enc_ord`.

In [458]:

```
df_enc_ord = enc_ord.transform(df['AGE'])
df_enc_ord['AGE'].value_counts()
```

Out[458]:

```
3.0    9067
2.0    5526
4.0    4398
1.0    3649
Name: AGE, dtype: int64
```

We can pass more than one feature to our encoder, and it will encode all features. For example, let us consider the column `EDUC12R`, which includes the respondent's answer to the question:

> **Which best describes your education?**
>
> 1. **High school or less**
> 2. **Some college/assoc. degree**
> 3. **College graduate**
> 4. **Postgraduate study**

In [459]:

```
df['EDUC12R'].value_counts()
```

Out[459]:

```
Some college/assoc. degree    7134
College graduate              6747
Postgraduate study            4071
High school or less           3846
Name: EDUC12R, dtype: int64
```

We can try to fit the encoder using both `AGE` and `EDUC12R`:

In [460]:

```
features = ['EDUC12R', 'AGE']

enc_ord = ce.OrdinalEncoder(handle_missing='return_nan')
enc_ord.fit(df[features])
```

Out[460]:

```
OrdinalEncoder(cols=['EDUC12R', 'AGE'], drop_invariant=False,
               handle_missing='return_nan', handle_unknown='value',
               mapping=[{'col': 'EDUC12R', 'data_type': dtype('O'),
                         'mapping': Some college/assoc. degree    1
College graduate              2
Postgraduate study            3
High school or less           4
NaN                          -2
dtype: int64},
                        {'col': 'AGE', 'data_type': dtype('O'),
                         'mapping': 18-29    1
30-44    2
45-65    3
65+      4
NaN     -2
dtype: int64}],
```

```
                return_df=True, verbose=0)
```

For this column, the order that the encoder "guesses" is not the desired order - the "High school or less" answer should have the smallest value, followed by "Some college/assoc. degree", then "College graduate", then "Postgraduate study".

To address this, we will pass a dictionary that tells the encoder exactly how to map these columns so that they are in the desired order:

In [461]:

```
mapping_dict = {'col': 'AGE', 'mapping':
                {'18-29': 1,
                 '30-44': 2,
                 '45-65': 3,
                 '65+': 4}
                }, {'col': 'EDUC12R', 'mapping':
                  {'High school or less': 1,
                   'Some college/assoc. degree': 2,
                   'College graduate': 3,
                   'Postgraduate study': 4}
                    }

features = ['EDUC12R', 'AGE']

enc_ord = ce.OrdinalEncoder(handle_missing='return_nan', mapping=mapping_dict)
enc_ord.fit(df[features])
```

Out[461]:

```
OrdinalEncoder(cols=['EDUC12R', 'AGE'], drop_invariant=False,
               handle_missing='return_nan', handle_unknown='value',
               mapping=({'col': 'AGE',
                         'mapping': {'18-29': 1, '30-44': 2, '45-65': 3,
                                     '65+': 4}},
                        {'col': 'EDUC12R',
                         'mapping': {'College graduate': 3,
                                     'High school or less': 1,
                                     'Postgraduate study': 4,
                                     'Some college/assoc. degree': 2}}),
               return_df=True, verbose=0)
```

(Note: for certain features, some rows may have an "Omit" value. These should be mapped to -1, which we will convert to NaN later.)

Then, we can get the encoded version of these columns:

In [462]:

```
df_enc_ord = enc_ord.transform(df[features])
```

In [463]:

```
df_enc_ord['AGE'].value_counts()
```

Out[463]:

```
 3.0    9067
 2.0    5526
 4.0    4398
 1.0    3649
-1.0     158
Name: AGE, dtype: int64
```

In [464]:

```
df_enc_ord['EDUC12R'].value_counts()
```

Out[464]:

```
 2.0     7134
 3.0     6747
 4.0     4071
 1.0     3846
-1.0     1000
Name: EDUC12R, dtype: int64
```

Missing values were encoded as -1, which we can transform back to NaN:

In [465]:

```
df_enc_ord.replace(-1, float("NaN"), inplace=True)
df_enc_ord.isna().sum()
```

Out[465]:

```
EDUC12R    1000
AGE         158
dtype: int64
```

Note that the values in the encoded columns range from 1 to the number of categories.

For K nearest neighbors, the "importance" of each feature in determining the class label would be proportional to its scale. If we leave it as is, any feature with a larger range of possible values will be considered more "important!"

So, we will re-scale our encoded features to the unit interval:

In [466]:

```
for col in df_enc_ord.columns:
  df_enc_ord[col] = df_enc_ord[col]-df_enc_ord[col].min(skipna=True)
  df_enc_ord[col] = df_enc_ord[col]/df_enc_ord[col].max(skipna=True)
```

In [467]:

```
df_enc_ord.describe()
```

Out[467]:

	EDUC12R	AGE
count	21798.000000	22640.000000
mean	0.502202	0.542609
std	0.329376	0.323963
min	0.000000	0.000000
25%	0.333333	0.333333
50%	0.333333	0.666667
75%	0.666667	0.666667
max	1.000000	1.000000

Now, each feature is on the same scale - the value varies 0 to 1.

Encode categorical features

In the previous section, we encoded features that have a logical ordering.

Other categorical features, such as `RACE`, have no logical ordering.

In [468]:

```
df['RACE'].value_counts()
```

Out[468]:

```
White              15918
Black               2993
Hispanic/Latino     2210
Asian                686
Other                681
Name: RACE, dtype: int64
```

These should be encoded using one-hot encoding, which will create a new column for each unique value, and then put a 1 or 0 in each column to indicate the respondent's answer.

In [469]:

```
enc_oh = ce.OneHotEncoder(use_cat_names=True, handle_missing='return_nan')
enc_oh.fit(df['RACE'])
```

Out[469]:

```
OneHotEncoder(cols=['RACE'], drop_invariant=False, handle_missing='return_nan',
              handle_unknown='value', return_df=True, use_cat_names=True,
              verbose=0)
```

In [470]:

```
df_enc_oh = enc_oh.transform(df['RACE'])
```

In [471]:

```
df_enc_oh
```

Out[471]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	RACE_nan
0	1.0	0.0	0.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0
3	0.0	0.0	1.0	0.0	0.0	0.0
4	0.0	0.0	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...
22793	0.0	0.0	0.0	0.0	1.0	0.0
22794	0.0	0.0	0.0	1.0	0.0	0.0
22795	0.0	0.0	0.0	1.0	0.0	0.0
22796	0.0	0.0	0.0	0.0	1.0	0.0
22797	0.0	0.0	0.0	0.0	1.0	0.0

22798 rows × 6 columns

Note that we have some respondents for which this feature is not available. These respondents have a NaN in all `RACE` columns:

In [472]:

```
df_enc_oh.isnull().sum()
```

Out[472]:

```
RACE_Hispanic/Latino    310
RACE_Asian              310
RACE_Other              310
RACE_Black              310
RACE_White              310
RACE_nan                310
```

```
dtype: int64
```

So, we can drop the `RACE_nan` column.

(For certain columns, some rows may have an "Omit" value recorded. We would also drop `FEATURE_Omit` columns wherever they may occur, so that these will not be included in the distance computations.)

In [473]:

```
columns_to_drop = ['RACE_nan']
df_enc_oh.drop(columns_to_drop, axis=1, inplace=True)
```

In [474]:

```
df_enc_oh.head()
```

Out[474]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White
0	1.0	0.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0
3	0.0	0.0	1.0	0.0	0.0
4	0.0	0.0	0.0	1.0	0.0

Train a k nearest neighbors classifier

Now that we have a target variable, a couple of features, and training and test indices, let's see what happens if we try to train a K nearest neighbors classifier.

First, we'll prepare our feature data, by column-wise concatenating the ordinal-encoded feature columns and the one-hot-encoded feature columns:

In [475]:

```
X = pd.concat([df_enc_oh, df_enc_ord], axis=1)
```

Here are the summary statistics for the training data:

In [476]:

```
X.iloc[idx_tr].describe()
```

Out[476]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
count	15744.000000	15744.000000	15744.000000	15744.000000	15744.000000	15261.000000	15846.000000
mean	0.097561	0.030043	0.031885	0.133067	0.707444	0.503396	0.541398
std	0.296730	0.170712	0.175700	0.339657	0.454951	0.329551	0.324832
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.333333	0.333333
50%	0.000000	0.000000	0.000000	0.000000	1.000000	0.333333	0.666667
75%	0.000000	0.000000	0.000000	0.000000	1.000000	0.666667	0.666667
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

and for the test data:

In [477]:

```
X.iloc[idx_ts].describe()
```

Out[477]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
count	6744.000000	6744.000000	6744.000000	6744.000000	6744.000000	6537.000000	6794.000000
mean	0.099941	0.031584	0.026542	0.133155	0.708778	0.499414	0.545432
std	0.299943	0.174902	0.160753	0.339768	0.454359	0.328976	0.321933
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.333333	0.333333
50%	0.000000	0.000000	0.000000	0.000000	1.000000	0.333333	0.666667
75%	0.000000	0.000000	0.000000	0.000000	1.000000	0.666667	0.666667
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

This classifier will only use a few features, but we'll see how well we can do with those to start.

Baseline: "prediction by mode"

As a baseline against which to judge the performance of our classifier, let's find out the accuracy of a classifier that gives the majority class label (0) to all samples in our test set:

In [478]:

```
accuracy_score(y.iloc[idx_ts], np.repeat(0, len(y.iloc[idx_ts])))
```

Out[478]:

```
0.5321637426900585
```

A classifier trained on the data should do *at least* as well as the one that predicts the majority class label. Hopefully, we'll be able to do much better!

`KNeighborsClassifier` does not support data with NaNs

If we try to train a `KNeighborsClassifier` on our data using the default settings, it will fail with the error message

```
ValueError: Input contains NaN, infinity or a value too large for dtype('float64').
```

Un-comment these lines, run the cell, and see for yourself:

In [479]:

```
#clf = KNeighborsClassifier(n_neighbors=3)
#clf.fit(X.iloc[idx_tr], y.iloc[idx_tr])
```

This is because we have many missing values in our data:

In [480]:

```
X.isna().sum()
```

Out[480]:

```
RACE_Hispanic/Latino      310
RACE_Asian                310
RACE_Other                310
```

```
RACE_Black              310
RACE_White              310
EDUC12R                1000
AGE                     158
dtype: int64
```

The distance metric is not defined for vectors with missing values.

Writing our own `KNeighborsClassifier`

Although we cannot use the `sklearn` implementation of a `KNeighborsClassifier`, we can write our own. We need a few things:

- **a function that implements a distance metric**
- **a function that accepts a distance matrix and returns the indices of the K smallest values for each row**
- **a function that returns the majority vote of the training samples represented by those indices**

Let's start with the distance metric. Suppose we use an L1 distance computed over the features that are non-NaN for both samples:

In [481]:

```
def custom_distance(a, b):
  dif = np.abs(np.subtract(a,b))    # element-wise absolute difference
  # dif will have NaN for each element where either a or b is NaN
  l1 = np.nansum(dif, axis=1)  # sum of differences, treating NaN as 0
  return l1
```

The function above expects a vector for the first argument and a matrix for the second argument, and returns a vector.

For example: suppose you pass a test point x_t and a matrix of training samples where each row $x_0, \dots, x_n$ is another training sample. It will return a vector d_t with as many elements as there are training samples, and where the ith entry is the distance between the test point x_t and training sample x_i.

To see how to this function is used, let's consider an example with a small number of test samples and training samples.

Suppose we had this set of test data:

In [482]:

```
a_idx = np.array([10296, 510,4827,20937, 22501])
a = X.iloc[a_idx]
a
```

Out[482]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
10296	0.0	0.0	0.0	0.0	1.0	0.666667	0.666667
510	0.0	0.0	0.0	0.0	1.0	0.666667	1.000000
4827	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
20937	0.0	0.0	0.0	1.0	0.0	0.333333	0.333333
22501	NaN	NaN	NaN	NaN	NaN	1.000000	0.666667

and this set of training data:

In [483]:

```
b_idx = np.array([10379, 4343, 7359,  1028,  2266, 131, 11833, 14106,  6682,  4402, 11899,  5877, 11758, 13163])
b = X.iloc[b_idx]
b
```

Out[483]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
10379	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4343	0.0	1.0	0.0	0.0	0.0	0.666667	0.666667
7359	0.0	0.0	0.0	0.0	1.0	0.000000	0.000000
1028	0.0	0.0	0.0	1.0	0.0	0.000000	1.000000
2266	0.0	0.0	0.0	0.0	1.0	0.666667	1.000000
131	NaN	NaN	NaN	NaN	NaN	0.666667	1.000000
11833	0.0	0.0	0.0	0.0	1.0	0.000000	1.000000
14106	0.0	0.0	0.0	0.0	1.0	0.666667	0.000000
6682	0.0	0.0	0.0	0.0	1.0	0.000000	1.000000
4402	0.0	0.0	0.0	0.0	1.0	0.666667	0.333333
11899	0.0	0.0	0.0	0.0	1.0	0.000000	0.666667
5877	1.0	0.0	0.0	0.0	0.0	NaN	0.000000
11758	0.0	0.0	0.0	1.0	0.0	0.666667	0.666667
13163	0.0	0.0	0.0	1.0	0.0	0.666667	0.666667

We will set up a *distance matrix* in which to store the results. In the distance matrix, an entry in row i, column j represents the distance between row i of the test set and row j of the training set.

So the distance matrix should have as many rows as there are test samples, and as many columns as there are training samples.

In [484]:

```
distances_custom = np.zeros(shape=(len(a_idx), len(b_idx)))
distances_custom.shape
```

Out[484]:

```
(5, 14)
```

Instead of a conventional `for` loop, we will use a [tqdm](https://github.com/tqdm/tqdm) `for` loop. This library conveniently "wraps" the conventional `for` loop with a progress part, so we can see our progress while the loop is running.

In [485]:

```
# the first argument to tqdm, range(len(a_idx)), is the list we are looping over
for idx in tqdm(range(len(a_idx)),  total=len(a_idx), desc="Distance matrix"):
  distances_custom[idx] = custom_distance(X.iloc[a_idx[idx]], X.iloc[b_idx])
```

```
Distance matrix: 100%|██████████| 5/5 [00:00<00:00, 695.90it/s]
```

In [486]:

```
np.set_printoptions(precision=2) # show at most 2 decimal places
print(distances_custom)
```

```
[[0.   2.   1.33 3.   0.33 0.33 1.   0.67 1.   0.33 0.67 2.67 2.   2.  ]
 [0.   2.33 1.67 2.67 0.   0.   0.67 1.   0.67 0.67 1.   3.   2.33 2.33]
 [0.   2.33 1.   2.67 0.67 0.67 0.67 1.   0.67 0.67 0.33 2.67 2.33 2.33]
 [0.   2.67 2.67 1.   3.   1.   3.   2.67 3.   2.33 2.67 2.33 0.67 0.67]
 [0.   0.33 1.67 1.33 0.67 0.67 1.33 1.   1.33 0.67 1.   0.67 0.33 0.33]]
```

Now, for each test sample, we can:

- **get an array of indices from the *distance matrix*, sorted in order of increasing distance**
- **get the list of the K nearest neighbors as the first K elements from that list,**
- **and then from those entries - which are indices with respect to the distance matrix - get the corresponding indices in `X`:**

In [487]:

```
k = 3
distances_sorted = np.array([np.argsort(row) for row in distances_custom])
nn_lists = distances_sorted[:, :k]
nn_lists_idx = b_idx[nn_lists]
```

For example, here was the first test sample:

In [488]:

```
X.iloc[[10296]]
```

Out[488]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
10296	0.0	0.0	0.0	0.0	1.0	0.666667	0.666667

and here are its closest neighbors among the training samples:

In [489]:

```
X.iloc[nn_lists_idx[0]]
```

Out[489]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
10379	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4402	0.0	0.0	0.0	0.0	1.0	0.666667	0.333333
2266	0.0	0.0	0.0	0.0	1.0	0.666667	1.000000

their corresponding values in `y`:

In [490]:

```
y.iloc[nn_lists_idx[0]]
```

Out[490]:

```
10379    1
4402     0
2266     0
Name: PRES, dtype: int64
```

and their distances:

In [491]:

```
distances_custom[0, nn_lists[0]]
```

Out[491]:

```
array([0.  , 0.33, 0.33])
```

and so the predicted vote for the first test sample would be:

In [492]:

```
y.iloc[nn_lists_idx[0]].mode()
```

Out[492]:

```
0    0
dtype: int64
```

You may have noticed a problem: training sample 10379, which has all NaN values, has zero distance to *every* test sample according to our distance function. (Note that the first column in the distance matrix, corresponding to the first training sample, is all zeros.)

This means that this sample will be a "nearest neighbor" of *every* test sample! But, it's not necessarily similar to those other test samples. We just *don't have any information* by which to judge how similar it is to other samples.

The case with an all-NaN training sample is a bit extreme, but it illustrates how our simple distance metric is problematic in other situations as well:

- **If a sample has only NaN values for the features we decide to include, its distance to every other sample is 0 and it will be considered a "nearest neighbor" to everyone.**
- **If two samples have no non-NaN features in common - for example, if sample a is NaN for every feature where sample b is non-NaN - the distance between them will be 0, and they will be considered very similar, even though we just don't have any information by which to judge how similar they are.**
- **Even for samples that have non-NaN features in common, our distance metric is problematic because it doesn't care how much the two samples have in common - only how many features they disagree on.**

For example, consider these two samples from the original data:

In [493]:

```
pd.set_option('display.max_columns', 150)
disp_features = ['AGE8', 'RACE', 'REGION', 'SEX', 'SIZEPLAC', 'STANUM', 'EDUC12R', 'EDUC
COLL','INCOME16GEN', 'ISSUE16', 'QLT16', 'VERSION']
df.iloc[[0,1889]][disp_features]
```

Out[493]:

	AGE8	RACE	REGION	SEX	SIZEPLAC	STANUM	EDUC12R	EDUCCOLL	INCOME16GEN	ISSUE16
0	18-24	Hispanic/Latino	West	Female	Suburbs	California	Some college/assoc. degree	No college degree	Under $30,000	Foreign policy
1889	NaN	NaN	West	Female	Suburbs	California	NaN	NaN	NaN	NaN

These two samples have some things in common:

- **female**
- **from suburban California**

but we don't know much else about what they have in common or what they disagree on.

Our distance metric will consider them very similar, because they are identical with respect to every feature that is available in both samples.

On the other hand, consider these two samples:

In [494]:

```
df.iloc[[0,14826]][disp_features]
```

Out[494]:

	AGE8	RACE	REGION	SEX	SIZEPLAC	STANUM	EDUC12R	EDUCCOLL	INCOME16GEN	ISSUE16
0	18-24	Hispanic/Latino	West	Female	Suburbs	California	Some college/assoc. degree	No college degree	Under $30,000	Foreign policy
14826	18-24	Hispanic/Latino	South	Female	Rural	Oklahoma	High school or less	No college degree	Under $30,000	Foreign policy

These two samples have many more things in common:

- female
- Latino
- age 18-24
- no college degree
- income less then $30,000
- consider foreign policy to be the major issue facing the country
- consider "Has good judgment" to be the most important quality in deciding their presidential vote.

However, they also have some differences:

- some college/associate degree vs. high school education or less
- suburban California vs. rural Oklahoma

so the distance metric will consider them *less* similar than the previous pair.

Using our K nearest neighbors classifier on the test data

Later, we'll have to fix those issues we identified with the custom distance metric, but for now, we will proceed without changing it.

Now that we understand how our custom distance function works, let's compute the distance between every *test* sample and every *training* sample. We'll store the results in `distances_custom`.

In [495]:

```
distances_custom = np.zeros(shape=(len(idx_ts), len(idx_tr)))
distances_custom.shape
```

Out[495]:

```
(6840, 15958)
```

Loop over the indices in the *test* set that we set up earlier to compute the distance vector for each test sample:

In [496]:

```
for idx in tqdm(range(len(idx_ts)),  total=len(idx_ts), desc="Distance matrix"):
  distances_custom[idx] = custom_distance(X.iloc[idx_ts[idx]], X.iloc[idx_tr])
```

```
Distance matrix: 100%|██████████| 6840/6840 [00:14<00:00, 484.62it/s]
```

Then, we can compute the K nearest neighbors using those distances:

In [497]:

```
k = 3

# get nn indices in distance matrix
distances_sorted = np.array([np.argsort(row) for row in distances_custom])
nn_lists = distances_sorted[:, :k]

# get nn indices in training data matrix
nn_lists_idx = idx_tr[nn_lists]
```

```python
# predict using mode of nns
y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
```

In [498]:

```python
accuracy_score(y.iloc[idx_ts], y_pred)
```

Out[498]:

```
0.5641812865497076
```

This classifier seems to improve over the "prediction by mode" classifier! But, there is an important, fundamental issue that we should fix.

Handling ties

If you look at the lists of nearest neighbors, you may notice something unexpected. Some training samples appear very frequently, even hundreds of times, among the K closest neighbors.

For example, here are the nearest neighbors for the first 50 test samples. Do you see any repetition?

In [499]:

```python
print(nn_lists_idx[0:50])
```

```
[[ 2718 17530  3796]
 [ 5620 14376 19699]
 [21119 19449  7843]
 [18684  2099  1027]
 [19615 15863  3361]
 [13922 11211  8939]
 [  876 10379  1883]
 [ 3741 11553  7785]
 [  348   688 14534]
 [20904 22104  7114]
 [ 8049 17354  8123]
 [12554  1275  9068]
 [19615 15863  3361]
 [15980  2276  2161]
 [12554 12658 19609]
 [  876 10379  1883]
 [12554 12658 19609]
 [  876 10379  1883]
 [15015  8809 10151]
 [ 6045 19904 14233]
 [22248  4229  5671]
 [20913 21541 18999]
 [ 1349  5942  7648]
 [  876 10379  1883]
 [ 1349  5942  7648]
 [19787 20978 19094]
 [ 8049 17354  8123]
 [15980  2276  2161]
 [ 9298 15448 10395]
 [20913 21541 18999]
 [18684  2099  1027]
 [ 1349  5942  7648]
 [  876 10379  1883]
 [13922 11211  8939]
 [ 5749 18953 16004]
 [ 6045 19904 14233]
 [  876 10379  1883]
 [13085 19765 14192]
 [  876 10379  1883]
 [ 1349  5942  7648]
 [ 1349  5942  7648]
 [15980  2276  2161]
 [ 1349 10056 17430]
 [ 8049 17354  8123]
```

```
 [15015  8809 10151]
 [  876 10379  1883]
 [19384 21526 20069]
 [15742  8630    68]
 [13922 11211  8939]
 [16614   863    27]]
```

You might find that these three samples appear very often as nearest neighbors:

In [500]:

```
X.iloc[[876, 10379,  1883]]
```

Out[500]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
876	0.0	0.0	0.0	0.0	1.0	0.333333	NaN
10379	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1883	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667

But other samples that have the same distance, do not appear in the nearest neighbors list at all:

In [501]:

```
X[X['RACE_Hispanic/Latino'].eq(0) & X['RACE_Asian'].eq(0) & X['RACE_Other'].eq(0)
  & X['RACE_Black'].eq(0) &  X['RACE_White'].eq(1)
  & (X['EDUC12R'].eq(1/3.0) | pd.isnull(X['EDUC12R']))
  & (X['AGE'].eq(2/3.0) | pd.isnull(X['AGE']))    ]
```

Out[501]:

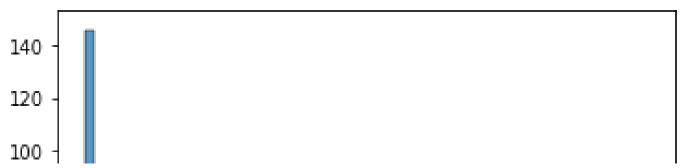
	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	EDUC12R	AGE
6	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
8	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
12	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
16	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
17	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
...	...	...	...	...	...	...	...
22726	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
22732	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
22751	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
22757	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667
22764	0.0	0.0	0.0	0.0	1.0	0.333333	0.666667

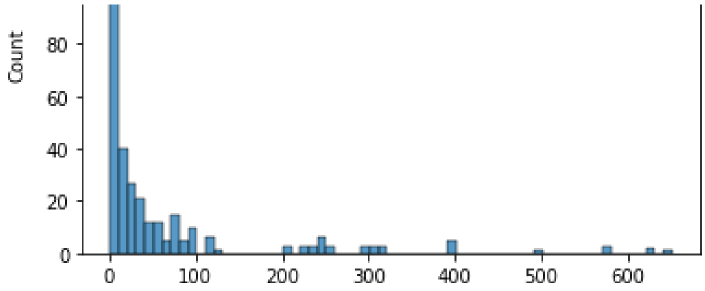
2251 rows × 7 columns

Looking at the frequency with which each training sample is returned, we can see the extent of the problem. Some training samples appear as a nearest neighbor more than 500 times!

In [502]:

```
vals, counts = np.unique(nn_lists_idx.ravel(), return_counts=True)
sns.histplot(counts, binwidth=10);
```





If a sample is returned as a nearest neighbor very often because it happens to be closer to the test points than other points, that would be OK. But in this case, that's not what is going on.

We are using `argsort` to get the K smallest distances to each test point. However, if there are more than K training samples that are at the minimum distance for a particular test point (i.e. a tie of more than K values, all having the minimum distance), `argsort` will return the first K of those in order of their index in the distance matrix (their order in `idx_tr`).

This means that some training samples will be returned much more often than others, simply because of their index.

To fix this, we will use an alternative, `lexsort`, that sorts first by the second argument, then by the first argument; and we will pass a random array as the first argument:

In [503]:

```
k = 3
r_matrix = np.random.random(size=(distances_custom.shape))
nn_lists = np.array([np.lexsort((r, row))[:k] for r, row in zip(r_matrix,distances_custo
m)])
nn_lists_idx = idx_tr[nn_lists]
y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
```

Now, we don't see nearly as much repitition of individual training samples among the nearest neighbors:

In [504]:

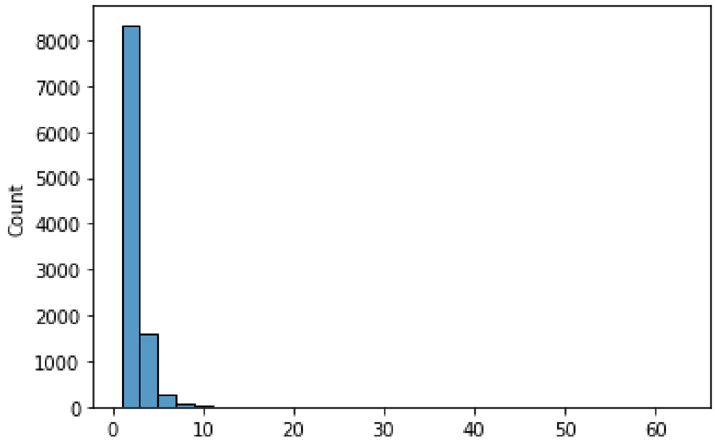
```
print(nn_lists_idx[0:50])
```

```
[[11504  5745 22125]
 [ 1516 18394 12470]
 [ 8904 18391  6465]
 [ 7310 15206 11355]
 [18068 11397 15548]
 [ 1888 14137  1198]
 [ 9830 17507 12954]
 [11029 21346 17689]
 [10538   507  7053]
 [16893  4486 13853]
 [20730 18340 10296]
 [18578 16569 15341]
 [ 8879  1889  7032]
 [19056 19164 10709]
 [ 1947  3549 16693]
 [  544 19363  8970]
 [10558 13421  6281]
 [19596 19535  1759]
 [18465 18315 16485]
 [20262  1472  4044]
 [15919 22518  3897]
 [15977 11115 16579]
 [ 2107 22578  9981]
 [ 5058   162  9924]
 [ 9540 21157   484]
 [ 3884 21529  4592]
 [12748  9090  4214]
 [22746 19799  5572]
 [10543  2179 22081]
```

```
 [17722  7990 18668]
 [ 1889  4307  9184]
 [  725 12601  3388]
 [ 4094  4719  3604]
 [   83  4660  4530]
 [ 5814 20427 18253]
 [ 8105 13070  4472]
 [18489 22726 11452]
 [11646  7899  8469]
 [ 8262 12023  1251]
 [  338 11561   372]
 [ 7451  9237  5614]
 [22725 17236  4283]
 [ 4836 19936 12172]
 [13106 20513  9662]
 [14689  4696   682]
 [ 9239  3368  8922]
 [14261 10053  5466]
 [13743 20043 10298]
 [  639 10979  4580]
 [ 7789  2726 20863]]
```

In [505]:

```
vals, counts = np.unique(nn_lists_idx.ravel(), return_counts=True)
sns.histplot(counts, binwidth=2);
```



Let's get the accuracy of this solution:

In [506]:

```
accuracy_score(y.iloc[idx_ts], y_pred)
```

Out[506]:

```
0.6035087719298246
```

Depending on the train-test split, the new classifier may have better performance (as it did in this case), or similar performance to the first classifier.

But conceptually, it is more sound, and less "fragile" - less sensitive to the draw of training data.

Using K-fold CV to select the number of neighbors

As a next step, to improve the classifier performance, we can use K-fold CV to select the number of neighbors. Note that we do not have to re-compute the distances inside each iteration of the loop, we can use pre-computed distances, so this is much faster than you might expect!

In [507]:

```
# pre-compute a distance matrix of training vs. training data
distances_kfold = np.zeros(shape=(len(idx_tr), len(idx_tr)))
```

```
for idx in tqdm(range(len(idx_tr)),  total=len(idx_tr), desc="Distance matrix"):
  distances_kfold[idx] = custom_distance(X.iloc[idx_tr[idx]], X.iloc[idx_tr])
```

```
Distance matrix: 100%|██████████| 15958/15958 [00:32<00:00, 490.85it/s]
```

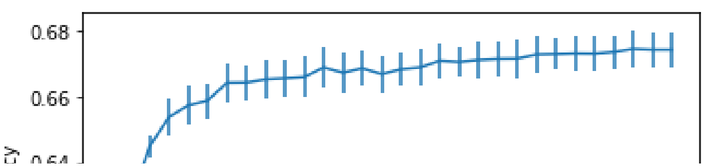
In [508]:

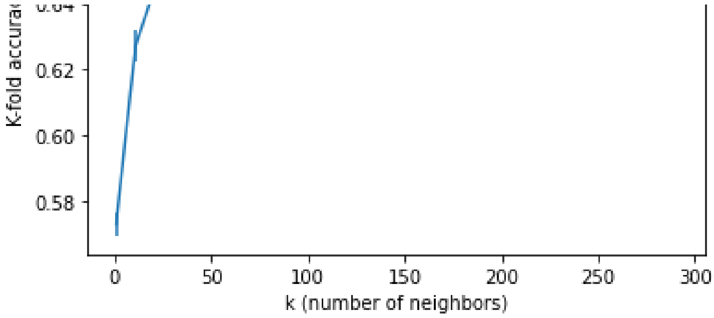
```
from sklearn.model_selection import KFold

n_fold = 5
k_list = np.arange(1, 301, 10)
n_k = len(k_list)
acc_list = np.zeros((n_k, n_fold))

kf = KFold(n_splits=5)

print(kf)

for isplit, idx_k in enumerate(kf.split(idx_tr)):

  print("Iteration %d" % isplit)

  # Outer loop: select training vs. validation data (out of training data!)
  idx_tr_k, idx_val_k = idx_k

  # get target variable values for validation data
  y_val_kfold = y.iloc[idx_tr[idx_val_k]]

  # get distance matrix for validation set vs. training set
  distances_val_kfold   = distances_kfold[idx_val_k[:, None], idx_tr_k]

  # generate a random matrix for tie breaking
  r_matrix = np.random.random(size=(distances_val_kfold.shape))

  # loop over the rows of the distance matrix and the random matrix together with zip
  # for each pair of rows, return sorted indices from distances_val_kfold
  distances_sorted = np.array([np.lexsort((r, row)) for r, row in zip(r_matrix,distances
_val_kfold)])

  # Inner loop: select value of K, number of neighbors
  for idx_k, k in enumerate(k_list):

    # now we select the indices of the K smallest, for different values of K
    # the indices in  distances_sorted are with respect to distances_val_kfold
    # from those - get indices in idx_tr_k, then in X
    nn_lists_idx = idx_tr[idx_tr_k[distances_sorted[:,:k]]]

    # get validation accuracy for this value of k
    y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
    acc_list[idx_k, isplit] = accuracy_score(y_val_kfold, y_pred)
```

```
KFold(n_splits=5, random_state=None, shuffle=False)
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

In [509]:

```
plt.errorbar(x=k_list, y=acc_list.mean(axis=1), yerr=acc_list.std(axis=1)/np.sqrt(n_fold
-1));

plt.xlabel("k (number of neighbors)");
plt.ylabel("K-fold accuracy");
```





Using this, we can find a better choice for K.

In [510]:

```
best_k = k_list[np.argmax(acc_list.mean(axis=1))]
print(best_k)
```

```
271
```

And compute the accuracy of the overall classifier on the test data, using this K.

In [511]:

```
r_matrix = np.random.random(size=(distances_custom.shape))
nn_lists = np.array([np.lexsort((r, row))[:best_k] for r, row in zip(r_matrix,distances_
custom)])
nn_lists_idx = idx_tr[nn_lists]
y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
```

In [512]:

```
accuracy_score(y.iloc[idx_ts], y_pred)
```

Out[512]:

```
0.6767543859649123
```

Improve on the basic classifier

In the sections above, I showed you how to use a K nearest neighbors classifier to predict the vote of a test sample based on three features: race, education, and age.

For this assignment, you will try to improve the performance of your classifier in three ways:

- **by adding more features**
- **by improving the distance metric**
- **by using feature selection or feature weights**

You can be creative in selecting your approach to each of these three - there isn't one right answer! But, you'll have to explain and justify your decisions.

Use more features

First, you will improve the model by additional features that you think may be relevant.

But, do *not* use questions that directly ask the participants how they feel about individual candidates, or about their party affiliation or political leaning.

Your choices for additional features include:

- **More demographic information:** `INCOME16GEN`, `MARRIED`, `RELIGN10`, `ATTEND16`, `LGBT`, `VETVOTER`, `SEX`
- **Opinions about political issues and about what factors are most important in determining which candidate to vote for:** `TRACK`, `SUPREME16`, `FINSIT`, `IMMWALL`, `ISIS16`, `LIFE`, `TRADE16`, `HEALTHCARE16`,

`GOVTDO10`, `GOVTANGR16`, `QLT16`, `ISSUE16`, `NEC`

Refer to the PDF documentation to see the question and the possible answers corresponding to each of these features. You may also choose to do some exploratory data analysis, to help you understand these features better.

For your convenience, here are all the possible answers to those survey questions:

In [513]:

```
features = ['INCOME16GEN', 'MARRIED', 'RELIGN10', 'ATTEND16', 'LGBT', 'VETVOTER',
            'SEX', 'TRACK', 'SUPREME16',  'FINSIT', 'IMMWALL', 'ISIS16', 'LIFE',
            'TRADE16', 'HEALTHCARE16', 'GOVTDO10', 'GOVTANGR16', 'QLT16',
            'ISSUE16', 'NEC']

for f in features:
  print(f)
  print(df[f].value_counts())
  print("****************************************************")
```

```
INCOME16GEN
$50,000-$99,999      2606
$100,000-$199,999    2015
$30,000-$49,999      1586
Under $30,000        1385
$250,000 or more      495
$200.000-$249,999     350
Name: INCOME16GEN, dtype: int64
****************************************************
MARRIED
Yes    5182
No     3611
Name: MARRIED, dtype: int64
****************************************************
RELIGN10
Other christian    1996
Catholic           1792
Protestant         1784
None               1137
Other               577
Jewish              196
Mormon              114
Muslim               71
Name: RELIGN10, dtype: int64
****************************************************
ATTEND16
Once a week or more    1411
A few times a year     1206
Never                   916
A few times a month     697
Name: ATTEND16, dtype: int64
****************************************************
LGBT
No     4007
Yes     194
Name: LGBT, dtype: int64
****************************************************
VETVOTER
No     3673
Yes     562
Name: VETVOTER, dtype: int64
****************************************************
SEX
Female    12620
Male      10129
Name: SEX, dtype: int64
****************************************************
TRACK
Seriously off on the wrong track         2614
Generally going in the right direction    1549
Omit                                       156
```

```
Name: TRACK, dtype: int64
**************************************************
SUPREME16
An important factor          2153
The most important factor     971
Not a factor at all           607
A minor factor                607
Omit                          131
Name: SUPREME16, dtype: int64
**************************************************
FINSIT
About the same    1716
Better today      1427
Worse today       1164
Omit                58
Name: FINSIT, dtype: int64
**************************************************
IMMWALL
Oppose     2400
Support    1785
Omit        180
Name: IMMWALL, dtype: int64
**************************************************
ISIS16
Somewhat well     1633
Somewhat badly    1200
Very badly        1055
Very well          282
Omit               195
Name: ISIS16, dtype: int64
**************************************************
LIFE
Better than life today    1837
Worse than life today     1376
About the same            1147
Omit                       202
Name: LIFE, dtype: int64
**************************************************
TRADE16
Takes away U.S. jobs          1939
Creates more U.S. jobs        1818
Has no effect on U.S. jobs     471
Omit                           334
Name: TRADE16, dtype: int64
**************************************************
HEALTHCARE16
Went too far              1995
Did not go far enough     1401
Was about right            844
Omit                       189
Name: HEALTHCARE16, dtype: int64
**************************************************
GOVTDO10
Government is doing too many things better left to businesses and individuals    2126
Government should do more to solve problems                                      2082
Omit                                                                              221
Name: GOVTDO10, dtype: int64
**************************************************
GOVTANGR16
Dissatisfied, but not angry        2066
Satisfied, but not enthusiastic    1170
Angry                               990
Enthusiastic                        327
Omit                                 81
Name: GOVTANGR16, dtype: int64
**************************************************
QLT16
Can bring needed change       3660
Has the right experience      2028
Has good judgment             1707
Cares about people like me    1304
Omit                           290
```

```
Name: QLT16, dtype: int64
**************************************************
ISSUE16
The economy       4832
Terrorism         1647
Foreign policy    1111
Immigration       1051
Omit               348
Name: ISSUE16, dtype: int64
**************************************************
NEC
Not so good    1881
Good           1540
Poor            874
Excellent       153
Omit             56
Name: NEC, dtype: int64
**************************************************
```

It is up to you to decide which features to include in your model. However, you must include

- **at least four features that are encoded using an ordinal encoder (and you should include an explicit mapping for these), and**
- **at least four features that are encoded using one-hot encoding.**

(If you decide to use the features I used above, they do "count" as part of the four. For example, you can use age, education, and two additional ordinal-encoded features, and race and three other one-hot-encoded features.)

To Do 1: Encode ordinal features

In the following cells, prepare your ordinal-encoded features as demonstrated in the "Encode ordinal features" section earlier in this notebook.

Use at least four features that are encoded using an ordinal encoder. (You can choose which features to include, but they should be features for which the values have a logical ordering that should be preserved in the distance computations!)

Make sure to explicitly specify the mappings for these, so that you can be sure that they are encoded using the correct logical order, and use other "best practices" described in that section where applicable.

Save the ordinal-encoded columnns in a data frame called `df_enc_ord`.

4 Ordinal features used are:

Feature 1: Education Qualification (EDUC12R)

Feature 2: In the fight against ISIS, how's US doing (ISIS16)

Feature 3: 2015 Total Family Income (INCOME16GEN)

Feature 4: Trade with other countries (TRADE16)

In [514]:

```
# TODO 1 - Encode at least four ordinal features
features = ['ISIS16', 'EDUC12R', 'INCOME16GEN', 'TRADE16']

enc_ord = ce.OrdinalEncoder(handle_missing='return_nan')
enc_ord.fit(df[features])
```

Out[514]:

```
OrdinalEncoder(cols=['ISIS16', 'EDUC12R', 'INCOME16GEN', 'TRADE16'],
               drop_invariant=False, handle_missing='return_nan',
               handle_unknown='value',
               mapping=[{'col': 'ISIS16', 'data_type': dtype('O'),
                         'mapping': NaN              -2
```

```
                              mapping': NaN                      2
Somewhat well      2
Somewhat badly     3
Omit               4
Very well          5
Very badly         6
dtype: int64},
                        {'col': 'EDUC12R', 'data_type': dtype('O'),
                         'mapping': Some college/assoc. degree    1

College grad...
High school or less          4
NaN                         -2
dtype: int64},
                        {'col': 'INCOME16GEN', 'data_type': dtype('O'),
                         'mapping': Under $30,000          1

$30,000-$49,999      2
$50,000-$99,999      3
$100,000-$199,999    4
NaN                 -2
$200.000-$249,999    6
$250,000 or more     7
dtype: int64},
                        {'col': 'TRADE16', 'data_type': dtype('O'),
                         'mapping': NaN                            -2

Creates more U.S. jobs         2
Has no effect on U.S. jobs     3
Takes away U.S. jobs           4
Omit                           5
dtype: int64}],
               return_df=True, verbose=0)
```

In [515]:

```
mapping_dict = {'col': 'ISIS16', 'mapping':
                {'Very badly': 1,
                 'Somewhat badly': 2,
                 'Somewhat well': 3,
                 'Very well': 4}
                }, {'col': 'EDUC12R', 'mapping':
                  {'High school or less': 1,
                   'Some college/assoc. degree': 2,
                   'College graduate': 3,
                   'Postgraduate study': 4}
                    }, {'col': 'INCOME16GEN', 'mapping':
                  {'$30,000-$49,999': 1,
                  '$50,000-$99,999': 2,
                  '$100,000-$199,999': 3,
                  '$250,000 or more': 4}
                }, {'col': 'TRADE16', 'mapping':
                 {'Takes away U.S. jobs': 1,
                 'Has no effect on U.S. jobs': 2,
                 'Creates more U.S. jobs': 3 }
                }

features = ['ISIS16', 'EDUC12R', 'INCOME16GEN', 'TRADE16']

enc_ord = ce.OrdinalEncoder(handle_missing='return_nan', mapping=mapping_dict)
enc_ord.fit(df[features])
```

Out[515]:

```
OrdinalEncoder(cols=['ISIS16', 'EDUC12R', 'INCOME16GEN', 'TRADE16'],
               drop_invariant=False, handle_missing='return_nan',
               handle_unknown='value',
               mapping=({'col': 'ISIS16',
                         'mapping': {'Somewhat badly': 2, 'Somewhat well': 3,
                                     'Very badly': 1, 'Very well': 4}},
                        {'col': 'EDUC12R',
                         'mapping': {'College graduate': 3,
                                     'High school or less': 1,
                                     'Postgraduate study': 4,
                                     'Some college/assoc. degree': 2}},
                        {'col': 'INCOME16GEN',
```

```
                             'mapping': {'$100,000-$199,999': 3,
                                         '$250,000 or more': 4,
                                         '$30,000-$49,999': 1,
                                         '$50,000-$99,999': 2}},
                            {'col': 'TRADE16',
                             'mapping': {'Creates more U.S. jobs': 3,
                                         'Has no effect on U.S. jobs': 2,
                                         'Takes away U.S. jobs': 1}}),
                    return_df=True, verbose=0)
```

In [516]:

```
df_enc_ord = enc_ord.transform(df[features])
```

In [517]:

```
df_enc_ord['ISIS16'].value_counts()
```

Out[517]:

```
-1.0    18628
 3.0     1633
 2.0     1200
 1.0     1055
 4.0      282
Name: ISIS16, dtype: int64
```

In [518]:

```
df_enc_ord['EDUC12R'].value_counts()
```

Out[518]:

```
 2.0    7134
 3.0    6747
 4.0    4071
 1.0    3846
-1.0    1000
Name: EDUC12R, dtype: int64
```

In [519]:

```
df_enc_ord['INCOME16GEN'].value_counts()
```

Out[519]:

```
-1.0    16096
 2.0     2606
 3.0     2015
 1.0     1586
 4.0      495
Name: INCOME16GEN, dtype: int64
```

In [520]:

```
df_enc_ord['TRADE16'].value_counts()
```

Out[520]:

```
-1.0    18570
 1.0     1939
 3.0     1818
 2.0      471
Name: TRADE16, dtype: int64
```

In [521]:

```
df_enc_ord.replace(-1, float("NaN"), inplace=True)
df_enc_ord.isna().sum()
```

Out[521]:

```
ISIS16         18628
```

```
EDUC12R         1000
INCOME16GEN    16096
TRADE16        18570
dtype: int64
```

Once you are finished processing the ordinal-encoded columns, print the names of the columns, and use `describe` to check the count of each column. Make sure that the range of each column is 0-1. Also make sure that missing values and "Omit" values are recorded as NaN.

In [522]:

```
df_enc_ord.columns
```

Out[522]:

```
Index(['ISIS16', 'EDUC12R', 'INCOME16GEN', 'TRADE16'], dtype='object')
```

In [523]:

```
for col in df_enc_ord.columns:
  df_enc_ord[col] = df_enc_ord[col]-df_enc_ord[col].min(skipna=True)
  df_enc_ord[col] = df_enc_ord[col]/df_enc_ord[col].max(skipna=True)
```

In [524]:

```
df_enc_ord.describe(include='all')
```

Out[524]:

	ISIS16	EDUC12R	INCOME16GEN	TRADE16
count	4170.000000	21798.000000	6702.000000	4228.000000
mean	0.424620	0.502202	0.403909	0.485691
std	0.305561	0.329376	0.295903	0.471166
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.333333	0.333333	0.000000
50%	0.333333	0.333333	0.333333	0.500000
75%	0.666667	0.666667	0.666667	1.000000
max	1.000000	1.000000	1.000000	1.000000

To Do 2: Encode categorical features

In the following cells, prepare the features that should be one-hot encoded, as demonstrated in the "Encode categorical features" section earlier in this notebook. Make sure to use any "best practices" described in that section where applicable.

Use at least four features that are encoded using a one-hot encoder. (You can choose which features to include, but they should be features for which the values have *no* logical ordering.)

Save the ordinal-encoded columnns in `df_enc_oh`.

4 One-hot encoded features used are:

Feature 1: Race (RACE)

Feature 2: Which politial party supporter you think yourself as (PARTYID)

Feature 3: Which issue is more important to you to be addressed (ISSUE16)

Feature 4: Opinion about Obama's policies (OBAMAPLCY16)

In [525]:

```
# TODO 2 - encode at least four one-hot-encoded features
features = ['RACE', 'PARTYID', 'ISSUE16', 'OBAMAPLCY16']

enc_oh = ce.OneHotEncoder(use_cat_names=True, handle_missing='return_nan')
enc_oh.fit(df[features])
```

Out[525]:

```
OneHotEncoder(cols=['RACE', 'PARTYID', 'ISSUE16', 'OBAMAPLCY16'],
              drop_invariant=False, handle_missing='return_nan',
              handle_unknown='value', return_df=True, use_cat_names=True,
              verbose=0)
```

In [526]:

```
df_enc_oh = enc_oh.transform(df[features])
```

In [527]:

```
df_enc_oh.head()
```

Out[527]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	RACE_nan	PARTYID_Democrat	PARTYID_Ind
0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	
1	1.0	0.0	0.0	0.0	0.0	0.0	1.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	1.0	0.0	0.0	1.0	

In [528]:

```
df_enc_oh.isnull().sum()
```

Out[528]:

```
RACE_Hispanic/Latino                                 310
RACE_Asian                                           310
RACE_Other                                           310
RACE_Black                                           310
RACE_White                                           310
RACE_nan                                             310
PARTYID_Democrat                                    1047
PARTYID_Independent                                 1047
PARTYID_Something else                              1047
PARTYID_Republican                                  1047
PARTYID_nan                                         1047
ISSUE16_Foreign policy                             13809
ISSUE16_The economy                                13809
ISSUE16_Terrorism                                  13809
ISSUE16_Immigration                                13809
ISSUE16_Omit                                       13809
ISSUE16_nan                                        13809
OBAMAPLCY16_nan                                    18369
OBAMAPLCY16_Change to more conservative policies   18369
OBAMAPLCY16_Change to more liberal policies        18369
OBAMAPLCY16_Omit                                   18369
OBAMAPLCY16_Continue Barack Obama's policies       18369
dtype: int64
```

In [529]:

```
columns_to_drop = ['RACE_nan', 'PARTYID_nan', 'ISSUE16_nan', 'ISSUE16_Omit', 'OBAMAPLCY16_nan', 'OBAMAPLCY16_Omit']
df_enc_oh.drop(columns_to_drop, axis=1, inplace=True)
```

In [530]:

```
df_enc_oh.head()
```

Out[530]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	PARTYID_Democrat	PARTYID_Independent	P
0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	
1	1.0	0.0	0.0	0.0	0.0	1.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	1.0	
3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	1.0	0.0	1.0	0.0	

Print the columns of your one-hot encoded features. Make sure you have dropped the columns corresponding with NaN and "Omit" in the title, which should not be included in the distance computations. (You should already represent NaNs directly in the data.)

In [531]:

```
df_enc_oh.columns
```

Out[531]:

```
Index(['RACE_Hispanic/Latino', 'RACE_Asian', 'RACE_Other', 'RACE_Black',
       'RACE_White', 'PARTYID_Democrat', 'PARTYID_Independent',
       'PARTYID_Something else', 'PARTYID_Republican',
       'ISSUE16_Foreign policy', 'ISSUE16_The economy', 'ISSUE16_Terrorism',
       'ISSUE16_Immigration',
       'OBAMAPLCY16_Change to more conservative policies',
       'OBAMAPLCY16_Change to more liberal policies',
       'OBAMAPLCY16_Continue Barack Obama's policies'],
      dtype='object')
```

In [532]:

```
df_enc_oh.describe(include='all')
```

Out[532]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	PARTYID_Democrat	PARTYID_Indeper
count	22488.000000	22488.000000	22488.000000	22488.000000	22488.000000	21751.000000	21751.00
mean	0.098275	0.030505	0.030283	0.133093	0.707844	0.406372	0.20
std	0.297692	0.171976	0.171368	0.339683	0.454764	0.491167	0.40
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
50%	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	0.00
75%	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	0.00
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00

Create a combined data matrix

In [533]:

```
X = pd.concat([df_enc_oh, df_enc_ord], axis=1)
```

In [534]:

```
X.describe(include='all')
```

Out[534]:

	RACE_Hispanic/Latino	RACE_Asian	RACE_Other	RACE_Black	RACE_White	PARTYID_Democrat	PARTYID_Indeper
count	22488.000000	22488.000000	22488.000000	22488.000000	22488.000000	21751.000000	21751.00
mean	0.098275	0.030505	0.030283	0.133093	0.707844	0.406372	0.20
std	0.297692	0.171976	0.171368	0.339683	0.454764	0.491167	0.40
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
50%	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	0.00
75%	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	0.00
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00

To Do 3: Describe your choice of features

In a text cell, explain the features you have chosen to add to the model.

- Why did you select this particular set of features?
- Do you have reason to believe these specific features will be predictive of 2016 presidential vote? Explain.

Ans: For Ordinal features I have selected:

EDUC12R, because this will help us determine how educated, informed the voting population is. Education can be leveraged to help enhance an individual's economic rationality.

ISIS16, given the history, I feel this feature should be added in the data matrix. People's opinion about how the situation was handled previously will be playing a role in their vote for the next president.

INCOME16GEN, obviously with every new president we have new taxation policies, financial budget, etc and this affects each and every person irrespective of their income. Therefore, it is very important to know the income categories where most of the population lies and thus we can know why a specific president was voted maybe because of his liberal policies.

TRADE16, trade is very important for any country it can either create or takeaway jobs. The choice depends upon person to person, for a local it might be good but for a bussinessman it may not be.

For Categotical features I have selected:

RACE, this feature can help us determine which race liked which president and hsi policies the most. We can draw some good conclusions out of this feature.

PARTY_ID, our beliefs, liking affect our decision making and hence I opted for this feature.

ISSUE16, when voting for the next president it is important to know which issue the US citizens want to be addressed the most and what all issues the next president is going to focus on. For a vote, majority of the times these choices intersect.

OBAMAPLCY16, before 2016 elections, Obama was elected twice and he was one of the most loved presidents. Therefore, it makes sense to include this feature so as to know what US citizens expect from their next president by knowing their opinion about Obama policies.

In conclusion, I believe, features like ISIS16, TRADE16, ISSUE16 and OBAMAPLCY16 will be very helpful in predicting 2016 presidential votes. Features like ISIS16, TRADE16 and ISSUE16 will help us determine which qualities, agendas they want from their next president. And since Obama was the last president and was elected twice, so taking into account his policies will help us know people's opinion about Obama policies

and do they expect the same from their next president too?
- Do you think these features will give you good "coverage" across the respondents? (For example, do you have at least one feature from each version of the survey?)

  Ans: Yes, I think the features that I have chosen produce good coverage. I have atleast one feature for both ordinal and categorical features from each version of the survey. Income, Education, Race and Party_id features being the most common in each version of the survey.

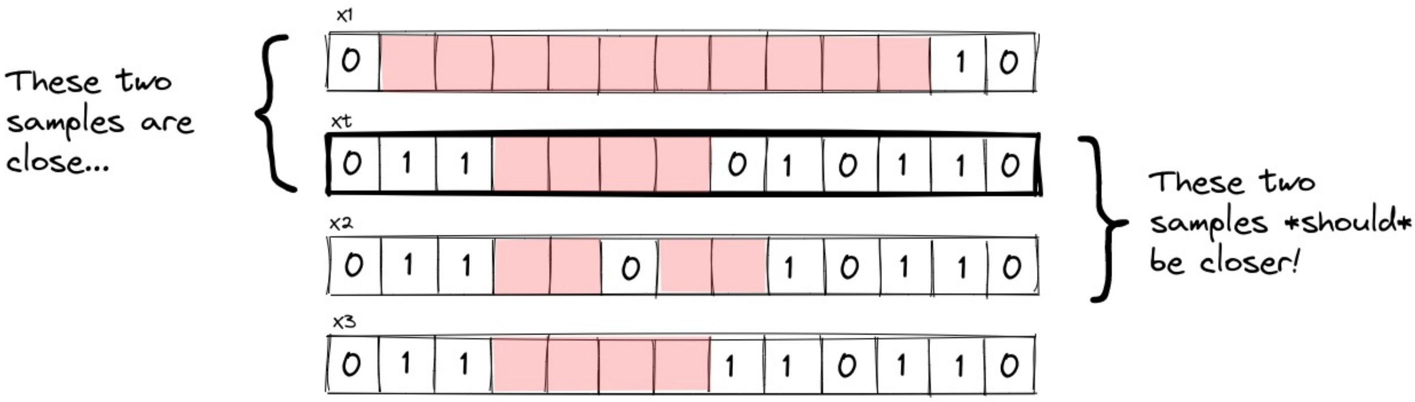
  I have selected features like:

- ISSUE16, TRADE16, ISIS16, and OBAMAPLCY16 which focus on future polices, agenda to be focused on.
- INCOME16, EDUC12R, RACE which tell us about background of people, and,
- PARTYID about their beliefs and liking.

Design a custom distance metric

Next, you should improve on the basic distance metric we used above. You can design any distance metric you think is appropriate (there is no one right answer to this question)!, but it must meet these criteria:

- it should handle NaN values in a reasonable way. Remember that a NaN does not mean two samples are *different* with respect to a feature; it means you don't have any information about whether they agree or disagree.
- samples should be considered closer if they have more features in common (assuming the same number of features that disagree).
- **optional:** you may decide that in some cases, samples with many features in common but a few small disagreements, should be considered closer than samples with few features in common but no disagreements.



For example, consider the image above, with a test sample (with bold outline) and three training samples. Red squares indicate missing values.

Training sample x_1 and training sample x_2 both have no disagreements with the test sample x_t. According to our basic L1 distance metric, they should both have 0 distance. However, in your modified metric, training sample x_2 should be considered closer to the test sample x_t, because it has more features in common.

Training sample x_3 has many features in common with the test sample x_t, but also one disagreement. You can decide which should be considered a closer neighbor of x_t: x_1 or x_3. But, you should explain your choice and justify your decision in the explanation.

To Do 4: Implement a custom distance metric

In [535]:

```
def custom_distance(a, b):
  dif = np.abs(np.subtract(a,b))     # element-wise absolute difference
  # dif will have NaN for each element where either a or b is NaN
  l1 = np.nansum(dif, axis=1)  # sum of differences, treating NaN as 0
  return l1
```

In [536]:

```
def custom_distance2(a, b):
  dif = np.abs(np.subtract(a,b))     # element-wise absolute difference
  # dif will have NaN for each element where either a or b is NaN
  l1 = np.nansum(dif, axis=1)  # sum of differences, treating NaN as 0
  dist = l1 + np.sum(np.isnan(dif), axis = 1 )    #adding no. of NAN to l1
  return dist
```

In [537]:

```
a = np.array([0, 1, np.nan, 0, 1, 1])
b = np.array([1, 1, 0, np.nan, 0, 1])
c = np.array([1, 1, 1, 1, 1, 1])
d = np.array([0, 0, 0, 0, 0, 0])
e = np.array([np.nan, np.nan, np.nan, np.nan, np.nan, np.nan])

b_arr = np.row_stack(([b,c], [d, e]))

print(custom_distance2(a, b_arr))
```

```
[4. 3. 4. 6.]
```

To Do 5: Describe your distance metric and justify your design choices

Describe your distance metric. First, write down an exact expression for

$d(a,b)$ **=** $\sum_{i=1}^{n}$ **{ |a_i - b_i| if a_i AND b_i != NAN , 1 if a_i OR b_i == NAN }**

Explain *why* you chose this function, and how it satisfies the criteria above.

Ans: I choose this function because:

- **it does not ignore the NANs in an array. It handles them by adding them them to the final distance formula.**
- **Like our previous matrix in this notebook, if a sample has all nans in it, Eg - X.iloc[10379], it will seem to be closer to every data point no matter what, as the diff will be zero, hence the final distance will come out to be zero. But as per my distance metic it will not, it will be farther as we are adding them to our final formula.**
- **As per this function, samples with more features in common will be closer as their difference will be zero.**
- **Also, this fucnction does not count NANs to be features i.e. neither an aggreement not an disagreement.**
- **This fucntion also takes care of the times when we have sample1 with many features in common and few disagreements and sample2 with few features in common and no disagreement. In this case sample 1 will be closer than sample 2. We can look this in the example below.**

Use several *specific examples* from the data to show how your distance function produces more meaningful distances than the previous "naive" distance metric. Compare and contrast the previous "naive" distance metric and your new distance metric on these examples.

In [538]:

```
a = np.array([0, 1, np.nan, 0, 1, 1])
b = np.array([1, 1, 0, np.nan, 0, 1])
c = np.array([1, 1, 1, 1, 1, 1])
d = np.array([0, 0, 0, 0, 0, 0])
e = np.array([np.nan, np.nan, np.nan, np.nan, np.nan, np.nan])
f = np.array(([0, 1, 0, 0, 1, 1]))
g = np.array(([0, 1, 1, 1, 0, 0]))

b_arr = np.row_stack(([b,c], [d, e], [f, g]))

print(custom_distance(a, b_arr))
print(custom_distance2(a, b_arr))
```

```
[2. 2. 3. 0. 0. 3.]
[4. 3. 4. 6. 1. 4.]
```

Observations:

- From the naive distance metric, sample_e is the closest point to sample_a but it should not be. The NANs are being considered as an agreement. As per my metric the NANs are being added hence the distance is 6 rather than 0.
- Sample_f is closest to sample_a as it has the most points in common with sample_a. We can also observe that as per naive metric the distnce for sample_f is 0 as it is taking nan as an agreement. But my per metric2 it is 1 i.e. 1 Nan
- Comparing sample_f and sample_g, sample_f is closer to sample_a as it has more agreements than sample_g.

Use feature selection or feature weights for better performance

Because the K nearest neighbor classifier weights each feature equally in the distance metric, including features that are not relevant for predicting the target variable can actually make performance worse.

To improve performance, you could either:

- use a subset of features that are most important, or
- use feature weights, so that more important features are scaled up and less important features are scaled down.

Feature selection has another added benefit - if you use fewer features, than you also get a faster inference time.

There are a few general approaches to feature selection:

- **Wrapper methods** use the ML model on the data, and select relevant features based the model performance. (For example, we might train a linear regression on different combinations of features, and then select the one that has the best performance on a validation set.)
- **Filter methods** use statistical characteristics of the data to select the features that are more useful for predicting the target variable. (For example, we might select the features that have the highest correlation with the target variable.)
- **Embedded methods** do feature selection "automatically" as part of the model training. (LASSO is an example of this type of feature selection.)

We also need to decide whether we want to take the dependencies between features into account, or not.

With **univariate feature selection**, we consider each feature independently. For example, we might score each feature according to its correlation with the target variable, then pick the features with the highest scores.

The problem with univariate feature selection is that some features may carry redundant information. In that case, we don't gain much from having both features in our model, but both will have similar scores.

As an alternative to univariate feature selection, we might consider **greedy feature selection**, where we start with a small number of features and then add features one at a time:

- Let S^{t-1} be the set of selected features at time $t-1$.
- Compute the score for all combinations of the current set of features + one more feature
- For the next time step S^t, add the feature that gave you the best score.
- Stop when you have added all features, or if adding another feature decreases the score.

Feature weighting does not have the benefit of faster inference time, but it does have the advantage of not throwing out useful information.

As with feature selection, there are both wrapper methods and filter methods, but filter methods tend to be much easier to compute.

There are many options for feature selection or feature weighting, and you can choose anything that seems reasonable to you - there isn't one right answer here! But, you will have to explain and justify your choice. For full credit, your design decisions should be well supported by the data.

full credit, your design decisions should be well supported by the data.

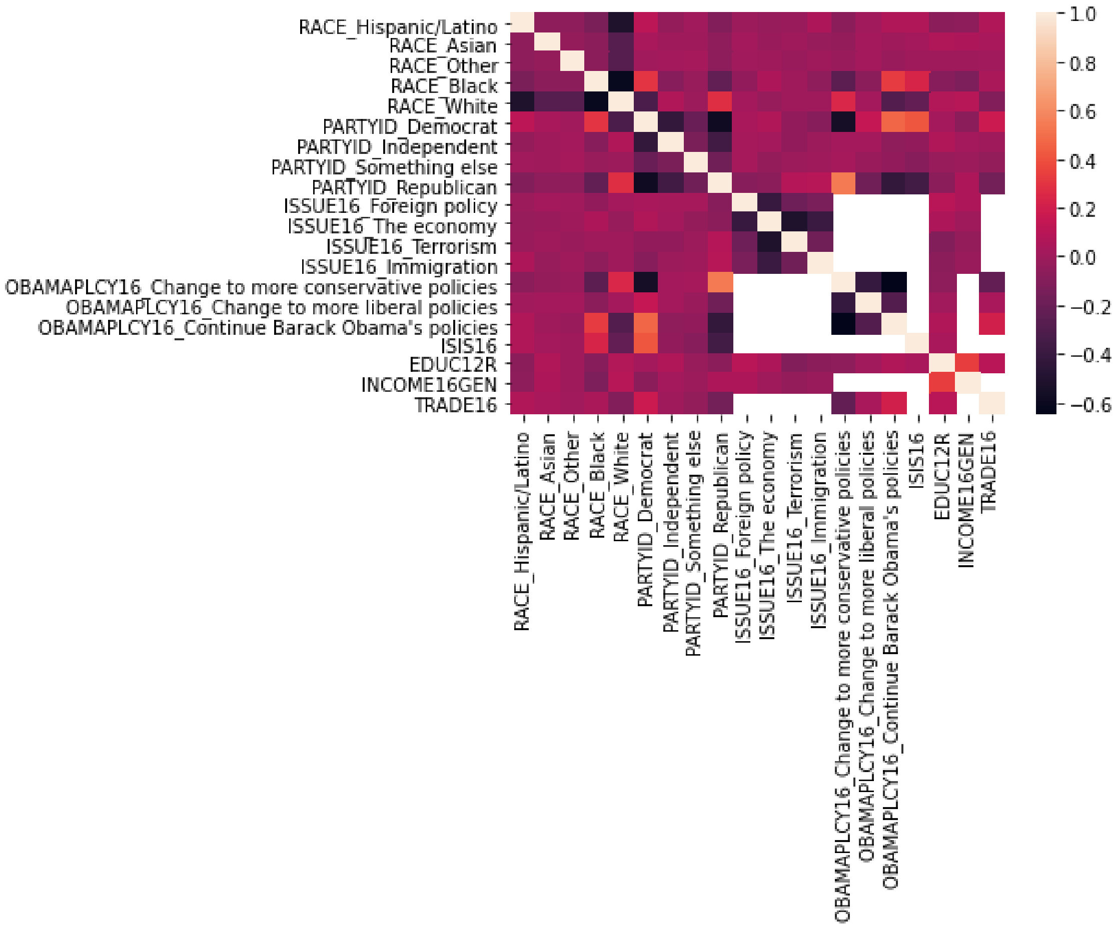
To Do 6: Implement feature selection or feature weighting

In [539]:

```
# TODO 6 - use either feature selection or feature weighting.
corr = X.corr()
sns.heatmap(corr)
```

Out[539]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0518062410>
```



In [562]:

```
from sklearn import preprocessing

X2 = X.fillna(0)
min_max_scaler = preprocessing.MinMaxScaler(feature_range =(0, 1))
X_after_scaling = min_max_scaler.fit_transform(X2)
print ("\nAfter min max Scaling : \n", X_after_scaling)

Standardisation = preprocessing.StandardScaler()
X_after_Standardisation = Standardisation.fit_transform(X2)
print ("\nAfter Standardisation : \n", X_after_Standardisation)
```

```
After min max Scaling :
 [[1.   0.   0.   ... 0.33 0.   0.  ]
 [1.   0.   0.   ... 0.67 0.   0.  ]
 [0.   1.   0.   ... 0.67 0.33 0.  ]
 ...
 [0.   0.   0.   ... 1.   0.   0.  ]
 [0.   0.   0.   ... 0.33 0.   0.  ]
 [0.   0.   0.   ... 0.33 0.   0.  ]]

After Standardisation :
 [[ 3.05 -0.18 -0.18 ... -0.43 -0.49 -0.33]
 [ 3.05 -0.18 -0.18 ...  0.55 -0.49 -0.33]
 [-0.33  5.68 -0.18 ...  0.55  0.88 -0.33]
```

```
 ...
 [-0.33 -0.18 -0.18 ...  1.54 -0.49 -0.33]
 [-0.33 -0.18 -0.18 ... -0.43 -0.49 -0.33]
 [-0.33 -0.18 -0.18 ... -0.43 -0.49 -0.33]]
```

In [566]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score as acc
from mlxtend.feature_selection import SequentialFeatureSelector as sfs

# RF classifier to use in feature selection
clf = RandomForestClassifier(n_estimators=100, n_jobs=-1)

# Step forward feature selection
sfs1 = sfs(clf,
           k_features=5,
           forward=True,
           floating=False,
           verbose=2,
           scoring='accuracy',
           cv=5)

# Perform SFFS
sfs1 = sfs1.fit(X2, y)
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    3.6s remaining:    0.0s
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:   48.0s finished

[2021-07-23 08:24:33] Features: 1/5 -- score: 0.7955083640225812[Parallel(n_jobs=1)]: Usi
ng backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    2.5s remaining:    0.0s
[Parallel(n_jobs=1)]: Done  19 out of  19 | elapsed:   47.0s finished

[2021-07-23 08:25:20] Features: 2/5 -- score: 0.8098953679438781[Parallel(n_jobs=1)]: Usi
ng backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    2.5s remaining:    0.0s
[Parallel(n_jobs=1)]: Done  18 out of  18 | elapsed:   45.5s finished

[2021-07-23 08:26:06] Features: 3/5 -- score: 0.8125272162639545[Parallel(n_jobs=1)]: Usi
ng backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    2.7s remaining:    0.0s
[Parallel(n_jobs=1)]: Done  17 out of  17 | elapsed:   45.7s finished

[2021-07-23 08:26:52] Features: 4/5 -- score: 0.8278354652259073[Parallel(n_jobs=1)]: Usi
ng backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    2.7s remaining:    0.0s
[Parallel(n_jobs=1)]: Done  16 out of  16 | elapsed:   45.3s finished

[2021-07-23 08:27:37] Features: 5/5 -- score: 0.8380995947864835
```

In [568]:

```
feat_cols = X2.columns[list(sfs1.k_feature_idx_)]
print(feat_cols)
```

```
Index(['RACE_White', 'PARTYID_Democrat', 'PARTYID_Republican',
       'OBAMAPLCY16_Change to more conservative policies', 'EDUC12R'],
      dtype='object')
```

In [569]:

```
print(sfs1.k_score_)
```

```
0.8380995947864835
```

In [576]:

```
X_trans = X2[feat_cols]
```

In [578]:

```
X_trans.describe(include = "all")
```

Out[578]:

	RACE_White	PARTYID_Democrat	PARTYID_Republican	OBAMAPLCY16_Change to more conservative policies	EDUC12R
count	22798.000000	22798.000000	22798.000000	22798.000000	22798.000000
mean	0.698219	0.387709	0.313712	0.093166	0.480174
std	0.459041	0.487238	0.464011	0.290671	0.338094
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.333333
50%	1.000000	0.000000	0.000000	0.000000	0.333333
75%	1.000000	1.000000	1.000000	0.000000	0.666667
max	1.000000	1.000000	1.000000	1.000000	1.000000

To Do 7: Describe your feature selection/weighting and justify your design choices

Explain your approach to feature selection or feature weighting. What did you do in this section? Why do you think this was a good choice for this problem?

Ans: In this section, I opted for step forward feature selection which is a wrapper method. I opted for feature selection instead of feature weighting because in our survey data there are/can be some features which are/can be redundant and irrelevant features. We don't know there relevance as such. In the 5 survey sets, there are some features which are common in all sets like education, income, race which contribute to the model a lot while there are some features which can be seen only in one survey set therefore for me it didn't make sense to opt for feature weighting.

Step forward feature selection is a wrapper method which starts with evaluating each and every feature and selects the best k features that we have mentioned.

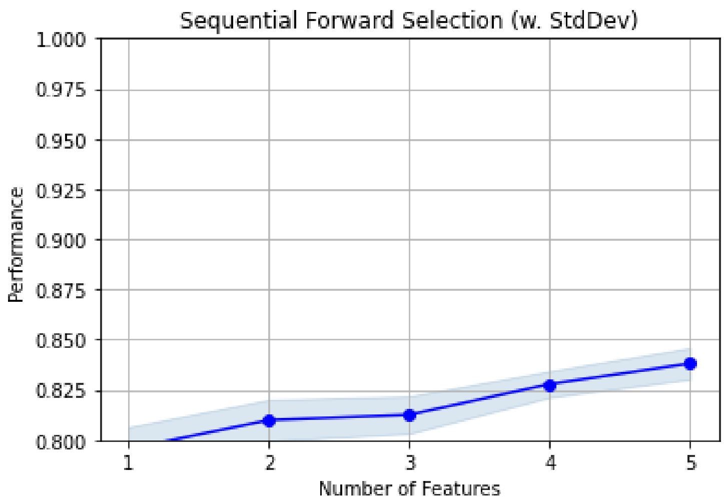
For full credit, you must show that your design decisions are supported by the data.

Were the results of the feature selection of feature weighting procedure surprising or unexpected in any way?

Ans:

In [592]:

```
fig1 = plot_sfs(sfs1.get_metric_dict(), kind='std_dev')
plt.ylim([0.8, 1])
plt.title('Sequential Forward Selection (w. StdDev)')
plt.grid()
plt.show()
```



In [593]:

```
print(sfs1.k_score_)
```

```
0.8380995947864835
```

We can see with each feature being added to the model, the model's performance can be seen increasing.

Evaluate your final classifier

Finally, train a K nearest neighbors classifier, using the approach shown earlier in this notebook, but with:

- **your custom distance metric**
- **your feature matrix with additional ordinal-encoded and one-hot-encoded features, and the results of your feature selection or feature weighting**

In [579]:

```
distances_custom = np.zeros(shape=(len(idx_ts), len(idx_tr)))
distances_custom.shape
```

Out[579]:

```
(6840, 15958)
```

In [ ]:

```
for idx in tqdm(range(len(idx_ts)),  total=len(idx_ts), desc="Distance matrix"):
  distances_custom[idx] = custom_distance2(X_trans.iloc[idx_ts[idx]], X_trans.iloc[idx_t
r])
```

In [584]:

```
k = 3

# get nn indices in distance matrix
distances_sorted = np.array([np.argsort(row) for row in distances_custom])
nn_lists = distances_sorted[:, :k]

# get nn indices in training data matrix
nn_lists_idx = idx_tr[nn_lists]

# predict using mode of nns
y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
```

In [585]:

```
accuracy_score(y.iloc[idx_ts], y_pred)
```

Out[585]:

```
0.7972222222222223
```

To Do 8: Select K (number of neighbors) for your final classifier

Once you have made your other design choices, you need to choose the value of K (the number of neighbors.

For full credit, use cross validation to select K, and plot the mean validation accuracy for each candidate model.

If you can't use cross validation, you will get partial credit for selecting a reasonable value and justifying your choice.

Make sure *not* to use your test set to determine the best K, since this is part of the training process.

In [ ]:

```
# TODO 8 - select the number of neighbors
```

```
# TODO 6 - select the number of neighbors

# pre-compute a distance matrix of training vs. training data
distances_kfold = np.zeros(shape=(len(idx_tr), len(idx_tr)))

for idx in tqdm(range(len(idx_tr)),  total=len(idx_tr), desc="Distance matrix"):
  distances_kfold[idx] = custom_distance2(X_trans.iloc[idx_tr[idx]], X_trans.iloc[idx_tr
])
```

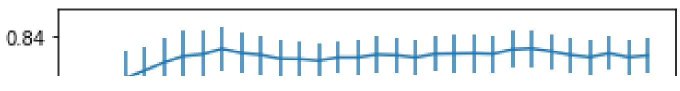
In [587]:

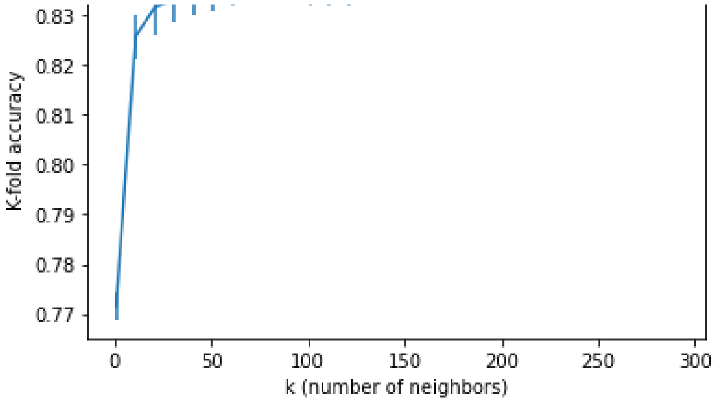
```
from sklearn.model_selection import KFold

n_fold = 5
k_list = np.arange(1, 301, 10)
n_k = len(k_list)
acc_list = np.zeros((n_k, n_fold))

kf = KFold(n_splits=5)

print(kf)

for isplit, idx_k in enumerate(kf.split(idx_tr)):

  print("Iteration %d" % isplit)

  # Outer loop: select training vs. validation data (out of training data!)
  idx_tr_k, idx_val_k = idx_k

  # get target variable values for validation data
  y_val_kfold = y.iloc[idx_tr[idx_val_k]]

  # get distance matrix for validation set vs. training set
  distances_val_kfold   = distances_kfold[idx_val_k[:, None], idx_tr_k]

  # generate a random matrix for tie breaking
  r_matrix = np.random.random(size=(distances_val_kfold.shape))

  # loop over the rows of the distance matrix and the random matrix together with zip
  # for each pair of rows, return sorted indices from distances_val_kfold
  distances_sorted = np.array([np.lexsort((r, row)) for r, row in zip(r_matrix,distances
_val_kfold)])

  # Inner loop: select value of K, number of neighbors
  for idx_k, k in enumerate(k_list):

    # now we select the indices of the K smallest, for different values of K
    # the indices in  distances_sorted are with respect to distances_val_kfold
    # from those - get indices in idx_tr_k, then in X
    nn_lists_idx = idx_tr[idx_tr_k[distances_sorted[:,:k]]]

    # get validation accuracy for this value of k
    y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
    acc_list[idx_k, isplit] = accuracy_score(y_val_kfold, y_pred)
```

```
KFold(n_splits=5, random_state=None, shuffle=False)
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

In [588]:

```
plt.errorbar(x=k_list, y=acc_list.mean(axis=1), yerr=acc_list.std(axis=1)/np.sqrt(n_fold
-1));

plt.xlabel("k (number of neighbors)");
plt.ylabel("K-fold accuracy");
```





In [615]:

```
best_k = k_list[np.argmax(acc_list.mean(axis=1))]
print(best_k)
```

```
231
```

To Do 9: Evaluate your final classifier on the test set

Finally, evaluate the classifier accuracy on the test set. Print the test accuracy. Are you able to achieve at least 80% accuracy?

In [590]:

```
# TODO 9 - Evaluate on test set

r_matrix = np.random.random(size=(distances_custom.shape))
nn_lists = np.array([np.lexsort((r, row))[:best_k] for r, row in zip(r_matrix,distances_
custom)])
nn_lists_idx = idx_tr[nn_lists]
y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
```

In [625]:

```
y[idx_ts].describe(include = 'all')
```

Out[625]:

```
count    6840.000000
mean        0.467836
std         0.499001
min         0.000000
25%         0.000000
50%         0.000000
75%         1.000000
max         1.000000
Name: PRES, dtype: float64
```

In [591]:

```
accuracy_score(y.iloc[idx_ts], y_pred)
```

Out[591]:

```
0.8364035087719298
```

To Do 10: Discussion

a) Discuss the final classifier you developed. Does it perform well? Do you have ideas that you think could make it better? Do you think other models we studied, such as a logistic regression classifier, would be a better choice for this task?

Ans: Yes, the classifier developed performs really well, it gives us an accuracy score of 83%. I can make my model better my selecting better ordinal and categorical features and implementing a better feature selection

algo because right now my model can't learn some of the features well.

I think for this dataset, KNN is suited well because for logistic regression we need proper selection of features. In our model, external feature selection is implemented which can vary.

b) Look at some specific examples where your model does poorly. Do you notice any systematic problems?

Ans: My model fails when people vote for Trade, ISIS, Income, Issue. My model is incapable of learning these features.

This model also fails when data has the same inputs but are mapped to different outputs. Like President vs Obama plot. Such kind of plots will give us same kind of result for both presidents(Trump and Clinton) so it will get really hard to predict the election result.

c) In the examples where the model does not predict the correct 2016 vote, is it because the test sample has a different vote than training samples that are generally very similar? Or is it because the nearest neighbors are not really very similar to the test sample? Show specific examples to support your answer.

Ans: It is because the nearest neighbors are not very similar to the test sample.