6-knn-voter-classification-hw

November 2, 2023

1 Assignment: Voter classification using exit poll data

Fraida Fund

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

- **Net ID**: yl11221
- **Name**: Yinhao Liu

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 few sections of this notebook, I will show you how to prepare the data and and use a K nearest neighbors classifier for this task, including:

- getting the data and loading it into the workspace.
- preparing the data: dealing with missing data, encoding categorical data in numeric format, and splitting into training and test.

In the last few sections of the notebook, you will have to improve the basic model for better performance, using a custom distance metric and using feature selection or feature weighting. In these sections, you will have specific criteria to satisfy for each task.

However, you should also make sure your overall solution is good! An excellent solution to this problem will achieve greater than 80% validation accuracy. A great solution will achieve 75% or higher.

Grading note

- For full credit, you should achieve 75% or higher test accuracy overall in this notebook (i.e. when running your solution notebook from beginning to end).
- If your solution is in the top 3 for test accuracy (relative to your classmates), you'll also earn extra credit toward your overall course grade.

1.1 Import libraries

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import ShuffleSplit, KFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

np.set_printoptions(suppress=True)
```

1.2 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.

1.2.1 Download the data and documentation

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:

1. Visit the Roper Center website via NYU Libraries link. Click on the user icon in the top right of the page, and choose “Log in”.
2. For “Your Affiliation”, choose “New York University”.
3. Then, click on the small red text “Register” below the password input field. The email and password fields will be replaced by a new email field with two parts.
4. Enter your NYU email address in the email field, and then click the red “Register” button.
5. 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.
6. 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.
7. Then, open the Study Record for the 2016 National Election Day Exit Poll.
8. Click on the “Downloads” tab, and then click on the CSV data file in the “Datasets” section of this tab. Press “Accept” to accept the terms and conditions. Find the file `31116396_National2016.csv` in your browser’s default download location.
9. 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.

1.2.2 Upload into Colab filesystem

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.

[2]:
```
try:
  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])))

except:
  pass # not running in Colab
```

```
<IPython.core.display.HTML object>
```

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

1.2.3 Load data with pandas

Now, 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.

[3]:
```
df = pd.read_csv('31116396_National2016.csv')
df.head()
```

```
<ipython-input-3-d2daf1675d09>:1: DtypeWarning: Columns (85) have mixed types.
Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('31116396_National2016.csv')
```

[3]:
```
       ID             PRES                       HOU    WEIGHT @2WAYPRES16  \
0  135355  Hillary Clinton  The Democratic candidate  6.530935
1  135356  Hillary Clinton  The Democratic candidate  6.479016
2  135357  Hillary Clinton  The Democratic candidate  8.493230
3  135358  Hillary Clinton  The Democratic candidate  3.761814
4  135359  Hillary Clinton  The Democratic candidate  3.470473

     AGE   AGE3   AGE8  AGE45  AGE49  … TRUMPWOMEN TRUMPWOMENB UNIONHH12  \
0  18-29  18-29  18-24  18-44  18-49  …
1  18-29  18-29  25-29  18-44  18-49  …
2  30-44  30-59  30-39  18-44  18-49  …
3  30-44  30-59  30-39  18-44  18-49  …
4  45-65  30-59  45-49    45+  18-49  …

     VERSION VETVOTER WHITEREL WHNCLINC WHTEVANG WPROTBRN WPROTBRN3
0  Version 1                          No
1  Version 1                          No
2  Version 1                          No
3  Version 1                          No
4  Version 1                          No

[5 rows x 138 columns]
```

1.3 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.

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

Exit poll versions

This cell will tell us how many respondents answered each version of the survey:

[4]:
```python
df['VERSION'].value_counts()
```

[4]:
```
Version 2    5126
Version 1    5094
Version 3    4980
Version 4    4919
Version 5    4915
```

```
Name: VERSION, dtype: int64
```

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. Our classifier will need to handle that.

You may also notice that the data is *categorical*, not *numeric* - for each question, users choose their response from a finite set of possible answers. We will need to convert this type of data into something that our classifier can work with.

1.3.1 Label 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!

[5]:
```
df.describe(include='all')
```

[5]:
```
                   ID            PRES                        HOU  \
count    25034.000000           25034                      25034
unique            NaN               7                          5
top               NaN  Hillary Clinton  The Democratic candidate
freq              NaN           12126                      12041
mean    188663.858712             NaN                        NaN
std      27829.369563             NaN                        NaN
min     135355.000000             NaN                        NaN
25%     175885.250000             NaN                        NaN
50%     193824.500000             NaN                        NaN
75%     210374.500000             NaN                        NaN
max     226680.000000             NaN                        NaN

              WEIGHT @2WAYPRES16    AGE   AGE3   AGE8  AGE45  AGE49  …  \
count   25034.000000       25034  25034  25034  25034  25034  25034  …
unique           NaN           5      5      4      9      3      3  …
top              NaN               45-65  30-59  50-59    45+  18-49  …
freq             NaN       15568   9746  13697   5071  14436  12836  …
mean        1.003016         NaN    NaN    NaN    NaN    NaN    NaN  …
std         1.065169         NaN    NaN    NaN    NaN    NaN    NaN  …
min         0.047442         NaN    NaN    NaN    NaN    NaN    NaN  …
25%         0.525367         NaN    NaN    NaN    NaN    NaN    NaN  …
50%         0.745491         NaN    NaN    NaN    NaN    NaN    NaN  …
75%         1.031137         NaN    NaN    NaN    NaN    NaN    NaN  …
max        18.407688         NaN    NaN    NaN    NaN    NaN    NaN  …

       TRUMPWOMEN TRUMPWOMENB UNIONHH12    VERSION VETVOTER WHITEREL WHNCLINC  \
count       25034       25034     25034      25034    25034    25034    25034
unique          6           4         3          5        3        7        3
```

```
top                                                   Version 2
freq         20284         20284      20324         5126      20387      16441      15521
mean           NaN           NaN        NaN          NaN        NaN        NaN        NaN
std            NaN           NaN        NaN          NaN        NaN        NaN        NaN
min            NaN           NaN        NaN          NaN        NaN        NaN        NaN
25%            NaN           NaN        NaN          NaN        NaN        NaN        NaN
50%            NaN           NaN        NaN          NaN        NaN        NaN        NaN
75%            NaN           NaN        NaN          NaN        NaN        NaN        NaN
max            NaN           NaN        NaN          NaN        NaN        NaN        NaN

       WHTEVANG WPROTBRN WPROTBRN3
count     25034    25034     25034
unique        3        3         4
top
freq      20137    20503     22181
mean        NaN      NaN       NaN
std         NaN      NaN       NaN
min         NaN      NaN       NaN
25%         NaN      NaN       NaN
50%         NaN      NaN       NaN
75%         NaN      NaN       NaN
max         NaN      NaN       NaN

[11 rows x 138 columns]
```

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

Let's change that:

[6]:
```
df.replace(" ", float("NaN"), inplace=True)
```

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

[7]:
```
df.describe(include='all')
```

[7]:
```
                   ID             PRES                       HOU  \
count    25034.000000            24696                     23970
unique            NaN                6                         4
top               NaN  Hillary Clinton  The Democratic candidate
freq              NaN            12126                     12041
mean    188663.858712              NaN                       NaN
std      27829.369563              NaN                       NaN
min     135355.000000              NaN                       NaN
25%     175885.250000              NaN                       NaN
50%     193824.500000              NaN                       NaN
75%     210374.500000              NaN                       NaN
max     226680.000000              NaN                       NaN
```

```
           WEIGHT      @2WAYPRES16    AGE   AGE3   AGE8  AGE45  AGE49  … \
count  25034.000000           9466  24853  24853  24853  24853  24853  …
unique          NaN              4      4      3      8      2      2  …
top             NaN  Hillary Clinton  45-65  30-59  50-59    45+  18-49  …
freq            NaN           4611   9746  13697   5071  14436  12836  …
mean       1.003016            NaN    NaN    NaN    NaN    NaN    NaN  …
std        1.065169            NaN    NaN    NaN    NaN    NaN    NaN  …
min        0.047442            NaN    NaN    NaN    NaN    NaN    NaN  …
25%        0.525367            NaN    NaN    NaN    NaN    NaN    NaN  …
50%        0.745491            NaN    NaN    NaN    NaN    NaN    NaN  …
75%        1.031137            NaN    NaN    NaN    NaN    NaN    NaN  …
max       18.407688            NaN    NaN    NaN    NaN    NaN    NaN  …

       TRUMPWOMEN    TRUMPWOMENB UNIONHH12    VERSION VETVOTER  \
count        4750           4750      4710      25034     4647
unique          5              3         2          5        2
top         A lot  A lot or some        No  Version 2       No
freq         2481           3424      3771       5126     4040
mean          NaN            NaN       NaN        NaN      NaN
std           NaN            NaN       NaN        NaN      NaN
min           NaN            NaN       NaN        NaN      NaN
25%           NaN            NaN       NaN        NaN      NaN
50%           NaN            NaN       NaN        NaN      NaN
75%           NaN            NaN       NaN        NaN      NaN
max           NaN            NaN       NaN        NaN      NaN

                 WHITEREL WHNCLINC    WHTEVANG WPROTBRN   WPROTBRN3
count                8593     9513        4897     4531        2853
unique                  6        2           2        2           3
top     White Protestants       No  All others       No  All others
freq                 3038     8136        3627     3605        1357
mean                  NaN      NaN         NaN      NaN         NaN
std                   NaN      NaN         NaN      NaN         NaN
min                   NaN      NaN         NaN      NaN         NaN
25%                   NaN      NaN         NaN      NaN         NaN
50%                   NaN      NaN         NaN      NaN         NaN
75%                   NaN      NaN         NaN      NaN         NaN
max                   NaN      NaN         NaN      NaN         NaN

[11 rows x 138 columns]
```

Notice that *every* row has some missing data! If we drop the rows with missing values, we're left with an empty data frame (0 rows):

[8]:
```
df.dropna()
```

[8]: Empty DataFrame
```
Columns: [ID, PRES, HOU, WEIGHT, @2WAYPRES16, AGE, AGE3, AGE8, AGE45, AGE49,
AGE60, AGE65, AGEBLACK, AGEBYRACE, AGEBYRACE08, ATTEND16, ATTEND16B, ATTREL,
BACKSIDE, BORNCITIZEN, BREAK12, BREAK12A, BREAK12B, BRNAGAIN, CALL, CDNUM,
CHIEF16, CLINHONEST, CLINTONEMAIL, CLINTONEMAILB, CLINTONWINGEN, CLINTONWINGENB,
COUNT2, COUNTACC, CUBAN3, DESCRIBP12, EDUC12R, EDUCCOLL, EDUCHS, EDUCWHITE,
EDUCWHITEBYSEX, FAIRJUSTICE, FAVDEM2, FAVHCLIN16, FAVPRES16, FAVREP2, FAVTRUMP,
FINSIT, FTVOTER, GOVTANGR16, GOVTANGR16B, GOVTDO10, HANDLEECON16, HANDLEFP16,
HEALTHCARE16, HONEST16, IMMDEPORT, IMMWALL, INC100K, INC50K, INCOME3,
INCOME16GEN, INCWHITE, ISIS16, ISIS16B, ISSUE16, LATINO, LGBT, LIFE, MARRIED,
MORMON, NEC, NEC2, OBAMA2, OBAMA4, OBAMAPLCY16, OVER45, OVER65, PARTY,
PARTYBLACK, PARTYBYRACE, PARTYGENDER, PARTYID, PARTYWHITE, PHIL3, PRECINCT,
PTYIDEO, PTYIDEO7, QLT16, QRACE3, QRACEAI, QRACEAK, QRACEHI, QTYPE, QUALCLINTON,
QUALIFIED16, QUALTRUMP, RACE, RACE2B, RACEAI, …]
Index: []

[0 rows x 138 columns]
```

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

1.3.2 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:

[9]:
```
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
```

[10]:
```
df['PRES'].value_counts()
```

[10]:
```
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`. We will build a binary classifier that predicts `1` if it thinks a sample is Trump voter, and `0` if it thinks a sample is a Clinton voter.

[11]:
```
y = df['PRES'].map({'Donald Trump': 1, 'Hillary Clinton': 0})
y.value_counts()
```

[11]:
```
0    12126
1    10672
Name: PRES, dtype: int64
```

1.3.3 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 which users select an option from the following:

[12]:
```
df['AGE'].unique()
```

[12]:
```
array(['18-29', '30-44', '45-65', '65+', nan], dtype=object)
```

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+. Logically, we expect that a respondent whose age is 18-29 is most similar to the other 18-29 respondents, less similar to the 30-44 respondents, even less similar to the 45-65 respondents, and least similar to the 65+ respondents.

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

First, we define a dictionary that maps each possible value to an integer.

[13]:
```
mapping_dict_age = {'18-29': 1,
                    '30-44': 2,
                    '45-65': 3,
                    '65+': 4}
```

Then we can create a new data frame, `df_enc_ord`, by calling `map` on the original `df['AGE']` and passing this mapping dictionary. We will also specify that the index should be the same as the original data frame:

[14]:
```
df_enc_ord = pd.DataFrame( {'AGE': df['AGE'].map( mapping_dict_age) },
    index = df.index
)
```

We can extend this approach to encode more than one ordinal feature. 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

[15]:
```
df['EDUC12R'].value_counts()
```

[15]:
```
Some college/assoc. degree    7134
College graduate              6747
Postgraduate study            4071
High school or less           3846
Name: EDUC12R, dtype: int64
```

We can map both `AGE` and `EDUC12R` to ordinal-encoded columns in a new data frame:

[16]:
```
mapping_dict_age = {'18-29': 1,
                    '30-44': 2,
                    '45-65': 3,
                    '65+': 4}
mapping_dict_educ12r =  {'High school or less': 1,
                      'Some college/assoc. degree': 2,
                      'College graduate': 3,
                      'Postgraduate study': 4}
df_enc_ord = pd.DataFrame( {
    'AGE': df['AGE'].map( mapping_dict_age) ,
    'EDUC12R': df['EDUC12R'].map( mapping_dict_educ12r)
    },
    index = df.index
)
```

Note that the order matters - the "High school or less" answer should have the smallest value, followed by "Some college/assoc. degree", then "College graduate", then "Postgraduate study".

Also note that missing values are still treated as missing (not mapped to some value) - this is going to be important, since we are going to design a distance metric that treats missing values sensibly:

[17]:
```
df_enc_ord.isna().sum()
```

[17]:
```
AGE         158
EDUC12R    1000
dtype: int64
```

There's one more important step before we can use our ordinal-encoded values with KNN.

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 (because the value of the feature is used directly in the distance metric). If we leave it as is, any feature with a larger range of possible values will be considered more "important!", i.e. would count more in the distance metric.

So, we will re-scale our encoded features to the unit interval. We can do this with the `MinMaxScaler` in `sklearn`.

(Note: in general, you'd "fit" scalers etc. on only the training data, not the test data! In this case, however, the min and max in the training data is just due to our encoding, and will definitely be the same as the test data, so it doesn't really matter.)

[18]:
```
scaler = MinMaxScaler()

# first scale in numpy format, then convert back to pandas df
df_scaled = scaler.fit_transform(df_enc_ord)
df_enc_ord = pd.DataFrame(df_scaled, columns=df_enc_ord.columns)
```

[19]:
```
df_enc_ord.describe()
```

[19]:
```
                AGE       EDUC12R
count  22640.000000  21798.000000
mean       0.542609      0.502202
std        0.323963      0.329376
min        0.000000      0.000000
25%        0.333333      0.333333
50%        0.666667      0.333333
75%        0.666667      0.666667
max        1.000000      1.000000
```

[20]:
```
df_enc_ord['EDUC12R'].value_counts()
```

[20]:
```
0.333333    7134
0.666667    6747
1.000000    4071
0.000000    3846
Name: EDUC12R, dtype: int64
```

[21]:
```
df_enc_ord.isna().sum()
```

[21]:
```
AGE         158
EDUC12R    1000
dtype: int64
```

Later, you'll design a model with more ordinal features. For this initial demo, though, we'll stick to just those two - age and education - and continue to the next step.

1.3.4 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. It would be wrong to assign an ordered mapping to these features. 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.

(Note: for features that have two possible values - binary features - either categorical encoding or one-hot encoding would be valid in this case!)

[22]:
```
df['RACE'].value_counts()
```

[22]:
```
White              15918
Black               2993
Hispanic/Latino     2210
Asian                686
Other                681
Name: RACE, dtype: int64
```

We can one-hot encode this column using the `get_dummies` function in `pandas`.

[23]:
```
df_enc_oh = pd.get_dummies(df['RACE'], prefix='RACE' )
```

[24]:
```
df_enc_oh.describe()
```

[24]:
```
        RACE_Asian    RACE_Black  RACE_Hispanic/Latino    RACE_Other  \
count  22798.00000  22798.000000          22798.000000  22798.000000
mean       0.03009      0.131283              0.096938      0.029871
std        0.17084      0.337717              0.295880      0.170235
min        0.00000      0.000000              0.000000      0.000000
25%        0.00000      0.000000              0.000000      0.000000
50%        0.00000      0.000000              0.000000      0.000000
75%        0.00000      0.000000              0.000000      0.000000
max        1.00000      1.000000              1.000000      1.000000

         RACE_White
count  22798.000000
mean       0.698219
std        0.459041
min        0.000000
25%        0.000000
50%        1.000000
75%        1.000000
max        1.000000
```

Note that we added a `RACE` prefix to each column name - this prevents overlap between columns, e.g. if we also encoded another feature where "Other" was a possible answer. And, it helps us relate the new columns back to the original survey question that they answer.

For this survey data, we want to preserve information about missing values - if a sample did not have a value for the `RACE` feature, we want it to have a NaN in all `RACE` columns. We can assign

NaN to those rows as follows:

[25]:
```
df_enc_oh.loc[df['RACE'].isnull(), df_enc_oh.columns.str.startswith("RACE_")] =␣
↪float("NaN")
```

Now, for respondents where this feature is not available, we have a NaN in all `RACE` columns:

[26]:
```
df_enc_oh.isnull().sum()
```

[26]:
```
RACE_Asian              310
RACE_Black              310
RACE_Hispanic/Latino    310
RACE_Other              310
RACE_White              310
dtype: int64
```

1.3.5 Stack columns

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

[27]:
```
X = pd.concat([df_enc_oh, df_enc_ord], axis=1)
```

1.3.6 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.

[28]:
```
idx_tr, idx_ts = next(ShuffleSplit(n_splits = 1, test_size = 0.3, random_state␣
↪= 3).split(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`:

[29]:
```
y.iloc[idx_tr]
```

[29]:
```
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:

[30]:
```
y.iloc[idx_ts]
```

[30]:
```
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
```

Here are the summary statistics for the training data:

[31]:
```
X.iloc[idx_tr].describe()
```

[31]:
```
         RACE_Asian    RACE_Black  RACE_Hispanic/Latino    RACE_Other  \
count  15744.000000  15744.000000          15744.000000  15744.000000
mean       0.030043      0.133067              0.097561      0.031885
std        0.170712      0.339657              0.296730      0.175700
min        0.000000      0.000000              0.000000      0.000000
25%        0.000000      0.000000              0.000000      0.000000
50%        0.000000      0.000000              0.000000      0.000000
75%        0.000000      0.000000              0.000000      0.000000
max        1.000000      1.000000              1.000000      1.000000

         RACE_White           AGE      EDUC12R
count  15744.000000  15846.000000  15261.000000
mean       0.707444      0.541398     0.503396
std        0.454951      0.324832     0.329551
min        0.000000      0.000000     0.000000
25%        0.000000      0.333333     0.333333
50%        1.000000      0.666667     0.333333
75%        1.000000      0.666667     0.666667
max        1.000000      1.000000     1.000000
```

1.4 Train a k nearest neighbors classifier

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

1.4.1 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:

```
[32]: y_pred_baseline = np.repeat(0, len(y.iloc[idx_ts]))
      accuracy_score(y.iloc[idx_ts], y_pred_baseline)
```

```
[32]: 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!

1.4.2 `KNeighborsClassifier` does not support data with NaNs

We've previously seen the `sklearn` implementation of a `KNeighborsClassifier`. However, that won't work for this problem. 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').`

See for yourself:

```
[33]: # 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. And, as we explained previously, dropping rows with missing values is not a good option for this example.

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

and we have to be prepared to address complications at each stage!

1.4.3 Distance metric

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

```
[34]: 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 $x_0, \ldots, x_n$ 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 `a` (sampling some specific examples from the real data):

[35]:
```
a_idx = np.array([10296, 510,4827,20937, 22501])
a = X.iloc[a_idx]
a
```

[35]:
```
       RACE_Asian  RACE_Black  RACE_Hispanic/Latino  RACE_Other  RACE_White  \
10296         0.0         0.0                   0.0         0.0         1.0
510           0.0         0.0                   0.0         0.0         1.0
4827          0.0         0.0                   0.0         0.0         1.0
20937         0.0         1.0                   0.0         0.0         0.0
22501         NaN         NaN                   NaN         NaN         NaN

            AGE   EDUC12R
10296  0.666667  0.666667
510    1.000000  0.666667
4827   0.666667  0.333333
20937  0.333333  0.333333
22501  0.666667  1.000000
```

and this set of training data `b`:

[36]:
```
b_idx = np.array([10379, 4343, 7359,  1028,  2266, 131, 11833, 14106,  6682, 
 ↪4402, 11899,  5877, 11758, 13163])
b = X.iloc[b_idx]
b
```

[36]:
```
       RACE_Asian  RACE_Black  RACE_Hispanic/Latino  RACE_Other  RACE_White  \
10379         NaN         NaN                   NaN         NaN         NaN
4343          1.0         0.0                   0.0         0.0         0.0
7359          0.0         0.0                   0.0         0.0         1.0
1028          0.0         1.0                   0.0         0.0         0.0
2266          0.0         0.0                   0.0         0.0         1.0
131           NaN         NaN                   NaN         NaN         NaN
11833         0.0         0.0                   0.0         0.0         1.0
```

```
14106     0.0     0.0        0.0     0.0     1.0
6682      0.0     0.0        0.0     0.0     1.0
4402      0.0     0.0        0.0     0.0     1.0
11899     0.0     0.0        0.0     0.0     1.0
5877      0.0     0.0        1.0     0.0     0.0
11758     0.0     1.0        0.0     0.0     0.0
13163     0.0     1.0        0.0     0.0     0.0

            AGE   EDUC12R
10379       NaN       NaN
4343   0.666667  0.666667
7359   0.000000  0.000000
1028   1.000000  0.000000
2266   1.000000  0.666667
131    1.000000  0.666667
11833  1.000000  0.000000
14106  0.000000  0.666667
6682   1.000000  0.000000
4402   0.333333  0.666667
11899  0.666667  0.000000
5877   0.000000       NaN
11758  0.666667  0.666667
13163  0.666667  0.666667
```

We need to compute the distance from each sample in the test data `a`, to each sample in the training data `b`.

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.

[37]:
```python
distances_custom = np.zeros(shape=(len(a_idx), len(b_idx)))
distances_custom.shape
```

[37]:
```
(5, 14)
```

Now that we have the distance matrix set up, we're ready to fill it in with distance values. We will loop over each sample in the test set, and call the distance function passing that test sample and the entire training set.

Instead of a conventional `for` loop, we will use a 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.

[38]:
```python
# 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]].values, X.
 ↪iloc[b_idx].values)
```

```
Distance matrix: 100%|          | 5/5 [00:00<00:00, 1124.12it/s]
```

Let's look at those distances now:

```
[39]: 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]]
```

1.4.4 Find most common class of k nearest neighbors

Now that we have this distance matrix, 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,
- from those entries - which are indices with respect to the distance matrix - get the corresponding indices in `X` and `y`,
- and then predict the class of the test sample as the most common value of `y` among the nearest neighbors.

```
[40]: k = 3
      # array of indices sorted in order of increasing distance
      distances_sorted = np.array([np.argsort(row) for row in distances_custom])
      # first k elements in that list = indices of k nearest neighbors
      nn_lists = distances_sorted[:, :k]
      # map indices in distance matrix back to indices in `X` and `y`
      nn_lists_idx = b_idx[nn_lists]
      # for each test sample, get the mode of `y` values for the nearest neighbors
      y_pred =  [y.iloc[nn].mode()[0] for nn in nn_lists_idx]
```

1.4.5 Example: one test sample

For example, this was the first test sample:

```
[41]: X.iloc[[10296]]
```

```
[41]:        RACE_Asian  RACE_Black  RACE_Hispanic/Latino  RACE_Other  RACE_White  \
      10296         0.0         0.0                   0.0         0.0         1.0

                  AGE   EDUC12R
      10296  0.666667  0.666667
```

Here is its distance to each of the training samples in our "mini" training set:

```
[42]: distances_custom[0]
```

```
[42]: array([0.  , 2.  , 1.33, 3.  , 0.33, 0.33, 1.  , 0.67, 1.  , 0.33, 0.67,
       2.67, 2.  , 2.  ])
```

and here's the sorted list of indices from that distance matrix - i.e. the index of the training sample with the smallest distance, the index of the training sample with the second-smallest distance, and so on.

```
[43]: distances_sorted[0]
```

```
[43]: array([ 0,  4,  5,  9,  7, 10,  6,  8,  2,  1, 12, 13, 11,  3])
```

The indices (in the "mini" training sample) of the 3 nearest neighbors to this test sample are:

```
[44]: nn_lists[0]
```

```
[44]: array([0, 4, 5])
```

which corresponds to the following sample indices in the complete data X:

```
[45]: nn_lists_idx[0]
```

```
[45]: array([10379,  2266,   131])
```

So, its closest neighbors in the "mini" training set are:

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

```
[46]:        RACE_Asian  RACE_Black  RACE_Hispanic/Latino  RACE_Other  RACE_White  \
10379         NaN         NaN                   NaN         NaN         NaN
2266          0.0         0.0                   0.0         0.0         1.0
131           NaN         NaN                   NaN         NaN         NaN

       AGE   EDUC12R
10379  NaN       NaN
2266   1.0  0.666667
131    1.0  0.666667
```

and their corresponding values in y are:

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

```
[47]: 10379    1
2266     0
131      1
Name: PRES, dtype: int64
```

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

```
[48]: y.iloc[nn_lists_idx[0]].mode().values
```

[48]: array([1])

1.4.6 Example: entire test set

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`.

[49]:
```
distances_custom = np.zeros(shape=(len(idx_ts), len(idx_tr)))
distances_custom.shape
```

[49]: (6840, 15958)

To compute the distance vector for each test sample, loop over the indices in the *test* set:

[50]:
```
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]].values, X.
↪iloc[idx_tr].values)
```

```
Distance matrix: 100%|      | 6840/6840 [00:11<00:00, 603.73it/s]
```

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

[51]:
```
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]
```

[52]:
```
accuracy_score(y.iloc[idx_ts], y_pred)
```

[52]: 0.5307017543859649

That is… not great.

1.4.7 Problems with our simple classifier

The one-sample example we saw above is enough to illustrate some basic problems with our classifier, and to explain some of the reasons for its poor performance:

- the distance metric does not really tell us how *similar* two samples are, when there are samples with missing values,

- and the way that ties are handled - when there are multiple samples in the training set with the same distance - is not ideal.

We'll discuss both of these, but we'll only fix the second one in this section. Part of *your* assignment will be to address the issue with the custom distance metric in your solution.

In the example with the "mini" training and test sets, 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 *really* similar to those other test samples. We just *don't have any information* by which to judge how similar it is to other samples. These values are *unknown*, not *similar*.

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. In general, when there are no missing values, for a pair of samples each feature is either *similar* or *different*. Thus a metric like L1 distance, which explicitly measures the extent to which features are *different*, also implicitly captures the extent to which features are *similar*. When samples can have missing values, though, for a pair of samples each feature is either *similar*, *different*, or *unknown* (one or both samples is missing that value). In this case, a distance metric that only measures the extent of *difference* (like L1 or L2 distance) does not capture whether the features that are not different are *similar* or *unknown*. (Our custom distance metric, which is an L1 distance, treats values that are *unknown* as if they are *similar* - neither one increases the distance.) Similarly, a distance metric that only measures the extent of *similarity* would not capture whether the features that are not similar are *different* or *unknown*.

So when there are NaNs, our custom distance metric does not quite behave the way we want - we want distance between two samples to decrease with more similarity, and to increase with more differences. Our distance metric only considers difference, not similarity.

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

[53]:
```
pd.set_option('display.max_columns', 150)
disp_features = ['AGE8', 'RACE', 'REGION', 'SEX', 'SIZEPLAC', 'STANUM',␣
 ↪'EDUC12R', 'EDUCCOLL','INCOME16GEN', 'ISSUE16', 'QLT16', 'VERSION']
df.iloc[[0,1889]][disp_features]
```

[53]:
```
       AGE8             RACE REGION     SEX SIZEPLAC      STANUM  \
0     18-24  Hispanic/Latino   West  Female  Suburbs  California
1889    NaN              NaN   West  Female  Suburbs  California

                     EDUC12R           EDUCCOLL    INCOME16GEN  \
0     Some college/assoc. degree  No college degree  Under $30,000
1889                         NaN                NaN            NaN

              ISSUE16              QLT16    VERSION
0     Foreign policy  Has good judgment  Version 1
1889             NaN                NaN  Version 3
```

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.

[54]:
```
custom_distance(X.iloc[[0]].values, X.iloc[[1889]].values)
```

[54]:
```
array([0.])
```

On the other hand, consider these two samples:

[55]:
```
df.iloc[[0,14826]][disp_features]
```

[55]:
```
        AGE8            RACE REGION     SEX SIZEPLAC      STANUM  \
0      18-24  Hispanic/Latino   West  Female  Suburbs  California
14826  18-24  Hispanic/Latino  South  Female    Rural    Oklahoma

                           EDUC12R          EDUCCOLL    INCOME16GEN  \
0      Some college/assoc. degree  No college degree  Under $30,000
14826         High school or less  No college degree  Under $30,000

              ISSUE16              QLT16    VERSION
0      Foreign policy  Has good judgment  Version 1
14826  Foreign policy  Has good judgment  Version 2
```

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, even though they have a lot in common.

[56]:
```
custom_distance(X.iloc[[0]].values, X.iloc[[14826]].values)
```

[56]:
```
array([0.33])
```

A better distance metric will consider the level of disagreement between samples *and* the level of agreement. That will be part of your assignment - to write a new `custom_distance`.

Now, let's consider the second issue - how ties are handled.

Notice that in the example with the "mini" training and test sets, for the first test sample, there was one sample with 0 distance and 3 samples with 0.33 distance. The three nearest neighbors are the sample with 0 distance, and the *first 2* of the 3 samples with 0.33 distance.

In other words: ties are broken in favor of the samples that happen to have lower indices in the data.

On a larger scale, that means that some samples will have too much influence - they will appear over and over again as nearest neighbors, just because they are earlier in the data - while some samples will not appear as nearest neighbors at all simply because of this tiebreaker behavior.

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.

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

[57]:
```
print(nn_lists_idx[0:50])
```

```
[[ 2718  5524 10918]
 [10543 18617 18008]
 [20376  9109 10028]
 [ 8075 18949  9328]
 [15349 17812 10954]
 [10434  1109 19999]
 [21832  1229 20568]
 [13670 10344  9431]
 [ 4029 19789 19689]
 [20904 22075  3261]
 [ 8049 16074  2580]
 [12554  8237 17857]
 [15349 17812 10954]
 [ 1889 19501 14478]
 [12554  3707 19698]
 [21832  1229 20568]
 [12554  3707 19698]
 [21832  1229 20568]
 [21256 20149 20221]
 [ 4085 20155 22261]
 [ 5092  1741    86]
 [ 7954 21636 19520]
 [ 1349 10550  8801]
 [21832  1229 20568]
 [ 1349 10550  8801]
 [ 1348  6500 16854]
 [ 8049 16074  2580]
```

```
[ 1889 19501 14478]
[19073  7325  5681]
[ 7954 21636 19520]
[ 8075 18949  9328]
[ 1349 10550  8801]
[21832  1229 20568]
[10434  1109 19999]
[ 4815 12456 21213]
[ 4085 20155 22261]
[21832  1229 20568]
[18278 17012 10432]
[21832  1229 20568]
[ 1349 10550  8801]
[ 1349 10550  8801]
[ 1889 19501 14478]
[ 1349 10056 17430]
[ 8049 16074  2580]
[21256 20149 20221]
[21832  1229 20568]
[12893  9942  8931]
[ 1365    68 12088]
[10434  1109 19999]
[ 8728   731 13016]]
```

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

[58]:
```python
X.iloc[[876, 10379,  1883]]
```

[58]:
```
       RACE_Asian  RACE_Black  RACE_Hispanic/Latino  RACE_Other  RACE_White  \
876           0.0         0.0                   0.0         0.0         1.0
10379         NaN         NaN                   NaN         NaN         NaN
1883          0.0         0.0                   0.0         0.0         1.0

            AGE   EDUC12R
876         NaN  0.333333
10379       NaN       NaN
1883   0.666667  0.333333
```

But other samples that have the same distance - that are actually identical in `X`! - do not appear in the nearest neighbors list at all:

[59]:
```python
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['AGE'])  ]
```

[59]:
```
       RACE_Asian  RACE_Black  RACE_Hispanic/Latino  RACE_Other  RACE_White  \
34            0.0         0.0                   0.0         0.0         1.0
```

```
876      0.0      0.0               0.0      0.0      1.0
923      0.0      0.0               0.0      0.0      1.0
1220     0.0      0.0               0.0      0.0      1.0
1618     0.0      0.0               0.0      0.0      1.0
2887     0.0      0.0               0.0      0.0      1.0
3726     0.0      0.0               0.0      0.0      1.0
3816     0.0      0.0               0.0      0.0      1.0
5760     0.0      0.0               0.0      0.0      1.0
6052     0.0      0.0               0.0      0.0      1.0
7233     0.0      0.0               0.0      0.0      1.0
10785    0.0      0.0               0.0      0.0      1.0
11436    0.0      0.0               0.0      0.0      1.0
12425    0.0      0.0               0.0      0.0      1.0
13282    0.0      0.0               0.0      0.0      1.0
14781    0.0      0.0               0.0      0.0      1.0
15603    0.0      0.0               0.0      0.0      1.0

       AGE   EDUC12R
34     NaN  0.333333
876    NaN  0.333333
923    NaN  0.333333
1220   NaN  0.333333
1618   NaN  0.333333
2887   NaN  0.333333
3726   NaN  0.333333
3816   NaN  0.333333
5760   NaN  0.333333
6052   NaN  0.333333
7233   NaN  0.333333
10785  NaN  0.333333
11436  NaN  0.333333
12425  NaN  0.333333
13282  NaN  0.333333
14781  NaN  0.333333
15603  NaN  0.333333
```

A better tiebreaker behavior would be to randomly sample from neighbors with equal distance. Fortunately, this is an easy fix:

- We had been 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`).
- Now, 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:

[60]:
```
k = 3
# make a random matrix
```

```
r_matrix = np.random.random(size=(distances_custom.shape))
# sort using lexsort - first sort by distances_custom, then by random matrix in␣
 ↪case of tie
nn_lists = np.array([np.lexsort((r, row))[: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]
```

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

[61]:
```
print(nn_lists_idx[0:50])
```

```
[[10543   412 16993]
 [ 6046 20961 21469]
 [ 1889  5718 20577]
 [18828  1026 18730]
 [16629 15434 20170]
 [ 1889   698  5160]
 [12089  3738 20171]
 [ 5104  1089 15307]
 [22022  4908  4453]
 [18010  2231 10580]
 [15605 10346 21843]
 [ 5843   283 17935]
 [16629 11501 22627]
 [ 7746  9391 15430]
 [17145   213 17792]
 [18014 18038  4906]
 [ 2982 18544  9508]
 [13584  6313  4488]
 [ 7522  6407  9130]
 [17914 11720 15403]
 [ 9197  6070  9946]
 [18543  3976 17714]
 [ 4750  4542 10943]
 [19803  7665 19539]
 [ 6451 16443  1715]
 [ 5536 13412  3897]
 [18983 19975 17106]
 [11885 18827  7069]
 [ 2370 15496 20793]
 [18260  4691 17722]
 [16701 22083  7084]
 [12137  6023  9562]
 [ 1215 20672 11871]
 [14746  8610  1461]
```

```
 [ 8760  2537  3651]
 [ 7688 11718 21701]
 [17149 16511 13123]
 [ 6560 21404 13490]
 [20253 21344 14640]
 [ 1367 14619 12959]
 [ 9488  4901   954]
 [13040 19116 19164]
 [ 7284 20606 16294]
 [ 8601 11339 14863]
 [22117  7400  8089]
 [15551  3247 17419]
 [18658 11802 22669]
 [13386   935 13743]
 [ 2234 11207 11979]
 [11753   543 21952]]
```

Let's get the accuracy of *this* classifier, with the better tiebreaker behavior:

[62]:
```
accuracy_score(y.iloc[idx_ts], y_pred)
```

[62]:
```
0.6052631578947368
```

This classifier is less "fragile" - less sensitive to the draw of training data.

(Depending on the random draw of training and test data, it may or may not have better performance for a particular split - but on average, across all splits of training and test data, it should be better.)

1.4.8 Use K-fold CV to select the number of neighbors

In the previous example, we set the number of neighbors to 3, rather than letting this value be dictated by the data.

As a next step, to improve the classifier performance, we can use K-fold CV to select the number of neighbors. Note that depending how we do it, this can be *very* computationally expensive, or it can be not much more computationally expensive than just fixing the number of neighbors ourselves.

The most expensive part of the algorithm is computing the distance to the training samples. This is $O(nd)$ for each test sample, where n is the number of training samples and d is the number of features. If we can make sure this computation happens only once, instead of once per fold, this process will be fast.

Here, we pre-compute our distance matrix for *every* training sample:

[63]:
```
# 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]].values, X.
 ↪iloc[idx_tr].values)
```

```
Distance matrix: 100%|      | 15958/15958 [00:26<00:00, 607.84it/s]
```

Now, we'll use K-fold CV.

In each fold, as always, we'll further divide the training data into validation and training sets.

Then, we'll select the *rows* of the pre-computed distance matrix corresponding to the *validation* data in this fold, and the *columns* of the pre-computed distance matrix corresponding to the *training* data in this fold.

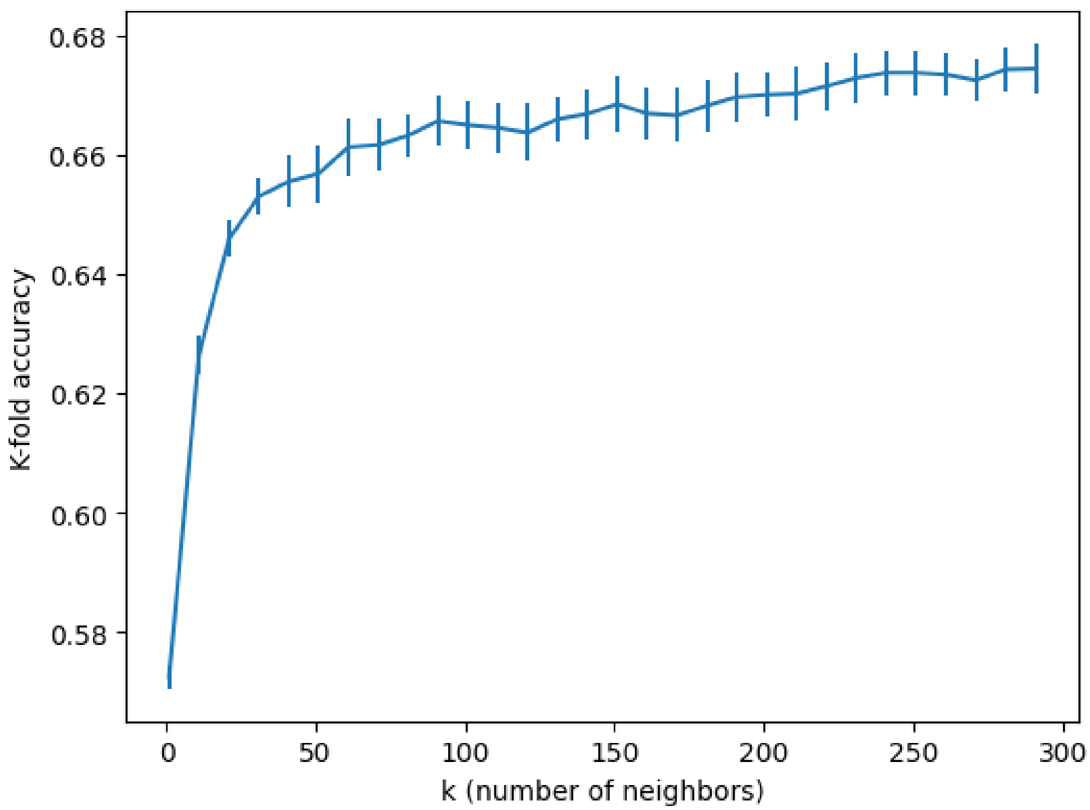
[64]:
```
n_fold = 5
k_list = np.arange(1, 301, 10)
n_k = len(k_list)
acc_list = np.zeros((n_k, n_fold))

kf = KFold(n_splits=5, shuffle=True)

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)
```

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Here's how the validation accuracy changes with number of neighbors:

[65]:
```
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 (number of neighbors):

[66]:
```
best_k = k_list[np.argmax(acc_list.mean(axis=1))]
print(best_k)
```

```
291
```

Now, let's re-run our KNN algorithm using the entire training set and this `best_k` number of neighbors, and check its accuracy?

[67]:
```
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]
```

[68]:
```
accuracy_score(y.iloc[idx_ts], y_pred)
```

[68]: 0.6761695906432749

1.4.9 Summarizing our basic classifier

Our basic classifier:

- uses three features (age, race, and education) to predict a respondent's vote
- doesn't mind if there are NaNs in the data (unlike the `sklearn` implementation, which throws an error)
- uses a random tiebreaker if there are multiple training samples with the same distance to the test sample
- uses the number of neighbors with the best validation accuracy, according to K-fold CV.

But, there are some outstanding issues:

- we have only used three features, out of many more available features.
- the distance metric only cares about the degree of disagreement (difference) between two samples, and doesn't balance it against the degree of agreement (similarity).

For this assignment, you will create an even better classifier by improving on those two issues.

1.5 Create a better classifier

In the remaining sections of this notebook, you'll need to fill in code to:

- implement a custom distance metric
- encode more features
- implement feature selection or feature weighting
- "train" and evaluate your final classifier, including K-Fold CV to select the best value for number of neighbors.

1.5.1 Create a better distance metric

Your first task is to improve on the basic distance metric we used above. There is no one correct answer - there are many ways to compute a distance - but for full credit, your distance metric should satisfy the following criteria:

1. if two samples are identical, the distance between them should be zero.
2. as the extent of *difference* between two samples increases, the distance should increase.
3. as the extent of *similarity* between two samples increases, the distance should decrease.

4. if in a pair of samples one or both have a NaN value for a given feature, the similarity or difference of this feature is *unknown*. Your distance metric should compute a smaller distance for a pair of samples with many similarities (even if there is some small difference) than for a pair of samples with mostly unknown similarity.

You should also avoid explicit `for` loops inside the `custom_distance` function - use efficient `numpy` functions instead. Note that `numpy` includes many functions that are helpful when working with arrays that have NaN values, including mathematical functions like sum, product, max and min, and logic functions like isnan.

Implement your distance metric

[69]:
```
# TODO - implement distance metric

def custom_distance(a, b):
    # Element-wise absolute difference
    dif = np.abs(np.subtract(a, b))

    # Use the mask to sum only the non-NaN differences
    total_difference = np.nansum(dif, axis=1)

    # Count the number of NaNs and apply the fixed penalty
    num_nans = np.sum(np.isnan(dif), axis=1)
    total_penalty = 0.1 * num_nans

    dist = total_difference + total_penalty
    return dist
```

Test cases for your distance metric You can use these test samples to check your work. (But, your metric should also satisfy the criteria in general - not only for these specific cases!)

First criteria: if two samples are identical, the distance between them should be zero.

[70]:
```
a = np.array([[0, 1, 0,       1, 0, 0.3]] )  # A0 - test sample
b = np.array([[0, 1, 0,       1, 0, 0.3]] )  # B0 - same as A0, should have 0␣
 ↪distance
```

[71]:
```
distances_ex = np.zeros(shape=(len(a), len(b)))
for idx, a_i in enumerate(a):
  distances_ex[idx] = custom_distance(a_i, b)

print(distances_ex)
```

```
[[0.]]
```

Second criteria: as the extent of *difference* between two samples increases, the distance should increase.

These should have *increasing* distance:

```
[72]: a = np.array([[0, 1, 0,      1, 0, 0.3]] )  # A0 - test sample
      b = np.array([[0, 1, 0,      1, 0, 0.3],               # B0 - same as A0, should␣
       ↪have 0 distance
             [0, 1, 0,      1, 0, 0.5],              # B1 - has one small␣
       ↪difference, should have larger distance than B0
             [0, 1, 0,      1, 0, 1  ],              # B2 - has more difference,␣
       ↪should have larger distance than B1
             [0, 0, 0,      1, 0, 0  ],              # B3 - has even more difference
             [1, 0, 1,      0, 1, 0  ]])             # B4 - has the most difference
```

```
[73]: distances_ex = np.zeros(shape=(len(a), len(b)))
      for idx, a_i in enumerate(a):
        distances_ex[idx] = custom_distance(a_i, b)

      print(distances_ex)
```

```
[[0.  0.2 0.7 1.3 5.3]]
```

These should have *decreasing* distance:

```
[74]: a = np.array([[0, 1, 0, 1, 0, 1]] )            # A0 - test sample
      b = np.array([[1, 0, 1, 0, 1, 0],              # B0 - completely different,␣
       ↪should have large distance
             [1, 0, 1, 0, 1, np.nan],         # B1 - less difference than B0, should␣
       ↪have less distance
             [1, 0, 1, 0, np.nan, np.nan]])   # B2 - even less difference than B1,␣
       ↪should have less distance
```

```
[75]: distances_ex = np.zeros(shape=(len(a), len(b)))
      for idx, a_i in enumerate(a):
        distances_ex[idx] = custom_distance(a_i, b)

      print(distances_ex)
```

```
[[6.  5.1 4.2]]
```

Third criteria: as the extent of *similarity* between two samples increases, the distance should decrease.

These should have *increasing* distance:

```
[76]: a = np.array([[0, 1, 0, 1, 0, 0.3]] )  # A0 - test sample
      b = np.array([[0, 1, 0, 1, 0, 0.3],              # B0 - same as A0, should have␣
       ↪0 distance
             [0, 1, 0, 1, 0, np.nan],            # B1 - has less similarity than B0,␣
       ↪should have larger distance
             [0, 1, 0, 1, np.nan, np.nan],       # B2 - has even less similarity,␣
       ↪should have larger distance
```

```
        [0, np.nan, np.nan, np.nan, np.nan, np.nan]])     # B3 - has least␣
↪similarity, should have larger distance
```

[77]:
```
distances_ex = np.zeros(shape=(len(a), len(b)))
for idx, a_i in enumerate(a):
  distances_ex[idx] = custom_distance(a_i, b)

print(distances_ex)
```

```
[[0.  0.1 0.2 0.5]]
```

Fourth criteria: if in a pair of samples one or both have a NaN value for a given feature, the similarity or difference of this feature is *unknown*. Your distance metric should compute a smaller distance for a pair of samples with many similarities (even if there is some small difference) than for a pair of samples with mostly unknown similarity.

These should have *increasing* distance:

[78]:
```
a = np.array([[0, np.nan, 0, 1, np.nan, 0.3]] )  # A0 - test sample
b = np.array([[0, np.nan, 0, 1, 0,      0.5],      # B0 - three similar␣
↪features, one small difference
        [0, np.nan, np.nan, np.nan, np.nan, np.nan]])  # B1 - much less␣
↪similarity than B0, should have larger distance
```

[79]:
```
distances_ex = np.zeros(shape=(len(a), len(b)))
for idx, a_i in enumerate(a):
  distances_ex[idx] = custom_distance(a_i, b)

print(distances_ex)
```

```
[[0.4 0.5]]
```

1.5.2 Encode more features

Our basic classifier used three features: age, race, and education. But there are many more features in this data that may be predictive of vote:

- 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`

in addition to the features `AGE`, `RACE`, and `EDUC12R`.

You will try to improve the model by adding some of these features.

(Note that we will *not* use questions that directly ask the participants how they feel about individual candidates, or about their party affiliation or political leaning. These features are a close proxy for the target variable, and we're going to assume that these are not available to the model.)

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:

[80]:
```
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 encode at least eight features, including:

- at least four features that are encoded using an ordinal encoder because they have a logical order (and you should include an explicit mapping for these), and
- at least four features that are encoded using one-hot encoding because they have no logical order.

Binary features - features that can take on only two values - "count" toward either category.

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

Encode ordinal features In the following cells, prepare your ordinal encoded features as demonstrated in the "Prepare data > 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 either binary features, or features for which the values have a logical ordering that should be preserved in the distance computations!)

Also:

- Save the ordinal-encoded columnns in a data frame called `df_enc_ord`.
- You should explicitly specify the mappings for these, so that you can be sure that they are encoded using the correct logical order.
- For some questions, there is also an "Omit" answer - if a respondent left that question blank on the questionnaire, the value for that question will be "Omit". Since "Omit" has no logical place in the order, we're going to treat these as missing values: don't include "Omit" in your `mapping_ord` dictionary, and then these Omit values will be encoded as NaN.
- Make sure to scale each column to the range 0-1, as demonstrated in the "Prepare data > Encode ordinal features" section earlier in this notebook.

[100]:
```
df['IMMWALL'].unique()
```

[100]:
```
array([nan, 'Oppose', 'Support', 'Omit'], dtype=object)
```

[102]:
```
# TODO - encode ordinal features

# set up mapping dictionary and list of features to encode with ordinal encoding
mapping_dict_income = {'Under $30,000': 1, '$30,000-$49,999': 2,␣
↪'$50,000-$99,999':3, '$100,000-$199,999':4, '$200.000-$249,999':5, '$250,000␣
↪or more':6}
mapping_dict_married = {'Yes':1, 'No':2}
mapping_dict_lgbt = {'Yes':1, 'No':2}
mapping_dict_sex = {'Female':1, 'Male':2}
mapping_dict_nec = {'Omit':0, 'Poor':1, 'Not so good':2, 'Good':3, 'Excellent':
↪4}
mapping_dict_health = {'Omit':0, 'Was about right':1, 'Did not go far enough':
↪2, 'Went too far':3}
mapping_dict_life = {'Omit':0, 'Worse than life today':1, 'About the same':2,␣
↪'Better than life today':3}
mapping_dict_immwall = {'Omit':0, 'Support':1, 'Oppose':2}
# use map to get the encoded columns, save in df_enc_ord
df_enc_ord = pd.DataFrame( {
    'INCOME16GEN': df['INCOME16GEN'].map( mapping_dict_income),
    'MARRIED': df['MARRIED'].map( mapping_dict_married),
    'LGBT': df['LGBT'].map( mapping_dict_lgbt),
    'SEX': df['SEX'].map( mapping_dict_sex),
    'NEC': df['NEC'].map( mapping_dict_nec),
    'HEALTHCARE16': df['HEALTHCARE16'].map( mapping_dict_health),
    'LIFE': df['LIFE'].map( mapping_dict_life),
    'IMMWALL': df['IMMWALL'].map( mapping_dict_immwall)
    },
    index = df.index
)

# scale each column to the range 0-1
scaler = MinMaxScaler()
df_scaled = scaler.fit_transform(df_enc_ord)
df_enc_ord = pd.DataFrame(df_scaled, columns=df_enc_ord.columns)
```

Look at the encoded data to check your work:

[103]:
```
df_enc_ord.describe()
```

[103]:
```
       INCOME16GEN      MARRIED         LGBT           SEX          NEC  \
count  8437.000000  8793.000000  4201.000000  22749.000000  4504.000000
mean      0.396302     0.410668     0.953821      0.445250     0.547735
std       0.266629     0.491983     0.209899      0.497004     0.206953
```

```
min        0.000000     0.000000     0.000000      0.000000     0.000000
25%        0.200000     0.000000     1.000000      0.000000     0.500000
50%        0.400000     0.000000     1.000000      0.000000     0.500000
75%        0.600000     1.000000     1.000000      1.000000     0.750000
max        1.000000     1.000000     1.000000      1.000000     1.000000

       HEALTHCARE16         LIFE      IMMWALL
count   4429.000000  4562.000000  4365.000000
mean       0.724844     0.670832     0.754296
std        0.294658     0.312952     0.288304
min        0.000000     0.000000     0.000000
25%        0.666667     0.333333     0.500000
50%        0.666667     0.666667     1.000000
75%        1.000000     1.000000     1.000000
max        1.000000     1.000000     1.000000
```

Encode categorical features In the following cells, prepare your categorical encoded features as demonstrated in the "Prepare data > Encode categorical features" section earlier in this notebook.

Use at least four features that are encoded using an categorical encoder. (You can choose which features to include, but they should be either binary features, or features for which the values do *not* have a logical ordering that should be preserved in the distance computations!)

Also:

- Save the categorical-encoded columnns in a data frame called `df_enc_oh`.
- For some questions, there is also an "Omit" answer - if a respondent left that question blank on the questionnaire, the value for that question will be "Omit". We're going to treat these as missing values. Before encoding the NaN values, you should drop the column corresponding to the "Omit" value from the data frame.

[104]:
```
# TODO - encode categorical features



# use get_dummies to get the encoded columns, stack and save in df_enc_oh
df_enc_oh = pd.get_dummies(df[['ISSUE16', 'QLT16', 'RELIGN10',␣
 ↪'GOVTDO10']],prefix=['ISSUE16', 'QLT16', 'RELIGN10', 'GOVTDO10'])

# drop the Omit columns, if any of these are in the data frame
df_enc_oh.drop(['ISSUE16_Omit', 'QLT16_Omit',␣
 ↪'TRACK_Omit','IMMWALL_Omit','GOVTDO10_Omit'],
                axis=1, inplace=True, errors='ignore')

# if a respondent did not answer a question, make sure they have NaN in all the␣
 ↪columns corresonding to that question
df_enc_oh.loc[df['ISSUE16'].isnull(), df_enc_oh.columns.str.
 ↪startswith("ISSUE16_")] = float("NaN")
```

```
df_enc_oh.loc[df['QLT16'].isnull(), df_enc_oh.columns.str.startswith("QLT16_")]␣
↪= float("NaN")
df_enc_oh.loc[df['RELIGN10'].isnull(), df_enc_oh.columns.str.
↪startswith("RELIGN10_")] = float("NaN")
df_enc_oh.loc[df['GOVTDO10'].isnull(), df_enc_oh.columns.str.
↪startswith("GOVTDO10_")] = float("NaN")
```

Stack columns Now, we'll create a combined data frame with all of the encoded features:

[105]:
```
X = pd.concat([df_enc_oh, df_enc_ord], axis=1)
```

[106]:
```
X.describe()
```

[106]:
```
       ISSUE16_Foreign policy  ISSUE16_Immigration  ISSUE16_Terrorism  \
count             8989.000000          8989.000000        8989.000000
mean                 0.123596             0.116921           0.183224
std                  0.329138             0.321344           0.386872
min                  0.000000             0.000000           0.000000
25%                  0.000000             0.000000           0.000000
50%                  0.000000             0.000000           0.000000
75%                  0.000000             0.000000           0.000000
max                  1.000000             1.000000           1.000000

       ISSUE16_The economy  QLT16_Can bring needed change  \
count          8989.000000                    8989.000000
mean              0.537546                       0.407164
std               0.498616                       0.491333
min               0.000000                       0.000000
25%               0.000000                       0.000000
50%               1.000000                       0.000000
75%               1.000000                       1.000000
max               1.000000                       1.000000

       QLT16_Cares about people like me  QLT16_Has good judgment  \
count                       8989.000000              8989.000000
mean                           0.145066                 0.189899
std                            0.352187                 0.392243
min                            0.000000                 0.000000
25%                            0.000000                 0.000000
50%                            0.000000                 0.000000
75%                            0.000000                 0.000000
max                            1.000000                 1.000000

       QLT16_Has the right experience  RELIGN10_Catholic  RELIGN10_Jewish  \
count                     8989.000000        7667.000000      7667.000000
mean                         0.225609           0.233729         0.025564
std                          0.418006           0.423229         0.157841
```

```
min                          0.000000          0.000000         0.000000
25%                          0.000000          0.000000         0.000000
50%                          0.000000          0.000000         0.000000
75%                          0.000000          0.000000         0.000000
max                          1.000000          1.000000         1.000000

       RELIGN10_Mormon  RELIGN10_Muslim  RELIGN10_None  RELIGN10_Other  \
count      7667.000000      7667.000000    7667.000000     7667.000000
mean          0.014869         0.009260       0.148298        0.075258
std           0.121036         0.095791       0.355418        0.263824
min           0.000000         0.000000       0.000000        0.000000
25%           0.000000         0.000000       0.000000        0.000000
50%           0.000000         0.000000       0.000000        0.000000
75%           0.000000         0.000000       0.000000        0.000000
max           1.000000         1.000000       1.000000        1.000000

       RELIGN10_Other christian  RELIGN10_Protestant  \
count               7667.000000          7667.000000
mean                   0.260337             0.232686
std                    0.438847             0.422571
min                    0.000000             0.000000
25%                    0.000000             0.000000
50%                    0.000000             0.000000
75%                    1.000000             0.000000
max                    1.000000             1.000000

       GOVTDO10_Government is doing too many things better left to businesses
and individuals  \
count                                        4429.000000
mean                                            0.480018
std                                             0.499657
min                                             0.000000
25%                                             0.000000
50%                                             0.000000
75%                                             1.000000
max                                             1.000000

       GOVTDO10_Government should do more to solve problems  INCOME16GEN  \
count                                        4429.000000     8437.000000
mean                                            0.470084        0.396302
std                                             0.499161        0.266629
min                                             0.000000        0.000000
25%                                             0.000000        0.200000
50%                                             0.000000        0.400000
75%                                             1.000000        0.600000
max                                             1.000000        1.000000
```

```
          MARRIED         LGBT          SEX          NEC  HEALTHCARE16  \
count  8793.000000  4201.000000  22749.000000  4504.000000   4429.000000
mean      0.410668     0.953821      0.445250     0.547735      0.724844
std       0.491983     0.209899      0.497004     0.206953      0.294658
min       0.000000     0.000000      0.000000     0.000000      0.000000
25%       0.000000     1.000000      0.000000     0.500000      0.666667
50%       0.000000     1.000000      0.000000     0.500000      0.666667
75%       1.000000     1.000000      1.000000     0.750000      1.000000
max       1.000000     1.000000      1.000000     1.000000      1.000000

             LIFE      IMMWALL
count  4562.000000  4365.000000
mean      0.670832     0.754296
std       0.312952     0.288304
min       0.000000     0.000000
25%       0.333333     0.500000
50%       0.666667     1.000000
75%       1.000000     1.000000
max       1.000000     1.000000
```

1.5.3 Feature selection or feature weighting

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.

Grading note For full credit,

- Your solution should not select *all* of the features (if using feature selection). Or, your solution should not assign the same weight to all features (if using feature weighting).
- Your solution should not select/weight highly the features that are least useful for predicting the target variable.
- Your solution *should* select/weight highly the features are are most useful for predicting the target variable.
- Your implementation should satisfy the requirements above generally, not only for this specific data. (It will be evaluated on other data.)
- Your solution must be well justified.

There are many options for feature selection or feature weighting, and you can choose anything that seems reasonable to you and meets the requirements above - there isn't one right answer here!

But, you will have to explain and justify your choice. In our lesson on feature selection/weighting, we discussed two parts to the problem of identifying the best subset of features:

- **Search**: you will have to describe the search strategy you use to determine the features or feature subsets to evaluate.
- **Evaluate**: you will have to describe the approach you use to evaluate the "goodness" of a feature or feature subset. Since this dataset has the added complication of missing values, you should also make sure to explain how you handle missing values in your evaluation.

And, you will have to describe the approach you used to select the best **number** of features to include or best **size** of feature subset (if you are using feature selection, not feature weighting).

For full credit, you will have to convince me that the approach you selected is a good match for (1) the data, and (2) the learning model.

In the following cell, implement feature selection or feature weighting, and return the results in `X_trans`:

- If you use feature selection, `X_trans` should have all of the rows of `X`, but only a subset of its columns. You should create a variable `feat_inc` which is a list of all of the features you want to include in the model.
- If you use feature weighting, `X_trans` should have the same dimensions of `X`, but instead of each column being in the range 0-1, each column will be scaled according to its importance (more important features will be scaled up, less important features will be scaled down). You should create a variable `feat_wt` which has a weight for every feature in `X`. Then, you'll multiply `X` by `feat_wt` to get `X_trans`.

Some important notes:

- The goal is to write code to find the feature selection or feature weighting, not to find it by manual inspection! Don't hard-code any values.
- Although `X_trans` will include all rows of the data, you should not use the test data in the process of finding `feat_inc` or `feat_wt`! Feature selection and feature weighting are considered part of model fitting, and so only the training data may be used in this process.
- For the "search" part of the optimization, you should not use any `sklearn` function or equivalent from another library - write pure Python+numpy code to implement the search yourself. For the "evaluate" part of the optimization, you are free to use an `sklearn` function, but make sure you understand what it does and are sure it is a good fit for the data and the model!

[107]:
```
idx_tr, idx_ts = next(ShuffleSplit(n_splits = 1, test_size = 0.3, random_state␣
↪= 3).split(df['PRES']))
```

[108]:
```
idx_tr
```

[108]:
```
array([ 1349, 14642, 18106, …, 11513,  1688,  5994])
```

[109]:
```
def feature_weighting_correlation(X_train, y_train):
    # Step 1: Calculate correlation coefficients between each feature and the␣
↪target
    correlations = X_train.corrwith(y_train, method='spearman')
```

```
    # Step 2: Take absolute value
    absolute_correlations = correlations.abs()

    # Step 3: Normalize the weights to be between 0 and 1
    feat_wt = absolute_correlations / absolute_correlations.max()

    return feat_wt
```

[110]:
```
feat_wt = feature_weighting_correlation(X.iloc[idx_tr], y.iloc[idx_tr])
```

[111]:
```
feat_wt
```

[111]:
```
ISSUE16_Foreign policy
0.167293
ISSUE16_Immigration
0.222820
ISSUE16_Terrorism
0.119490
ISSUE16_The economy
0.137402
QLT16_Can bring needed change
0.909448
QLT16_Cares about people like me
0.122329
QLT16_Has good judgment
0.333502
QLT16_Has the right experience
0.659901
RELIGN10_Catholic
0.015452
RELIGN10_Jewish
0.120069
RELIGN10_Mormon
0.122282
RELIGN10_Muslim
0.100504
RELIGN10_None
0.283757
RELIGN10_Other
0.140996
RELIGN10_Other christian
0.099976
RELIGN10_Protestant
0.236322
GOVTDO10_Government is doing too many things better left to businesses and
individuals    0.804072
```

```
GOVTDO10_Government should do more to solve problems
0.750402
INCOME16GEN
0.097869
MARRIED
0.216685
LGBT
0.180297
SEX
0.196784
NEC
0.730207
HEALTHCARE16
0.940651
LIFE
0.343321
IMMWALL
1.000000
dtype: float64
```

[112]:
```
X_trans = X.multiply(feat_wt)
```

Check your work:

[113]:
```
X_trans.describe()
```

[113]:
```
       ISSUE16_Foreign policy  ISSUE16_Immigration  ISSUE16_Terrorism  \
count             8989.000000          8989.000000        8989.000000
mean                 0.020677             0.026052           0.021893
std                  0.055062             0.071602           0.046227
min                  0.000000             0.000000           0.000000
25%                  0.000000             0.000000           0.000000
50%                  0.000000             0.000000           0.000000
75%                  0.000000             0.000000           0.000000
max                  0.167293             0.222820           0.119490

       ISSUE16_The economy  QLT16_Can bring needed change  \
count          8989.000000                    8989.000000
mean              0.073860                       0.370295
std               0.068511                       0.446842
min               0.000000                       0.000000
25%               0.000000                       0.000000
50%               0.137402                       0.000000
75%               0.137402                       0.909448
max               0.137402                       0.909448

       QLT16_Cares about people like me  QLT16_Has good judgment  \
```

count	8989.000000	8989.000000
mean	0.017746	0.063332
std	0.043083	0.130814
min	0.000000	0.000000
25%	0.000000	0.000000
50%	0.000000	0.000000
75%	0.000000	0.000000
max	0.122329	0.333502

	QLT16_Has the right experience	RELIGN10_Catholic	RELIGN10_Jewish \
count	8989.000000	7667.000000	7667.000000
mean	0.148880	0.003612	0.003069
std	0.275843	0.006540	0.018952
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000
max	0.659901	0.015452	0.120069

	RELIGN10_Mormon	RELIGN10_Muslim	RELIGN10_None	RELIGN10_Other \
count	7667.000000	7667.000000	7667.000000	7667.000000
mean	0.001818	0.000931	0.042081	0.010611
std	0.014801	0.009627	0.100853	0.037198
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000
max	0.122282	0.100504	0.283757	0.140996

	RELIGN10_Other christian	RELIGN10_Protestant \
count	7667.000000	7667.000000
mean	0.026028	0.054989
std	0.043874	0.099863
min	0.000000	0.000000
25%	0.000000	0.000000
50%	0.000000	0.000000
75%	0.099976	0.000000
max	0.099976	0.236322

	GOVTDO10_Government is doing too many things better left to businesses and individuals \
count	4429.000000
mean	0.385969
std	0.401760
min	0.000000
25%	0.000000
50%	0.000000

```
75%                                                       0.804072
max                                                       0.804072

       GOVTDO10_Government should do more to solve problems  INCOME16GEN  \
count                                        4429.000000     8437.000000
mean                                            0.352752        0.038786
std                                             0.374571        0.026095
min                                             0.000000        0.000000
25%                                             0.000000        0.019574
50%                                             0.000000        0.039147
75%                                             0.750402        0.058721
max                                             0.750402        0.097869

           MARRIED         LGBT           SEX          NEC  HEALTHCARE16  \
count  8793.000000  4201.000000  22749.000000  4504.000000   4429.000000
mean      0.088985     0.171971      0.087618     0.399960      0.681825
std       0.106605     0.037844      0.097802     0.151119      0.277171
min       0.000000     0.000000      0.000000     0.000000      0.000000
25%       0.000000     0.180297      0.000000     0.365104      0.627101
50%       0.000000     0.180297      0.000000     0.365104      0.627101
75%       0.216685     0.180297      0.196784     0.547655      0.940651
max       0.216685     0.180297      0.196784     0.730207      0.940651

              LIFE      IMMWALL
count  4562.000000  4365.000000
mean      0.230311     0.754296
std       0.107443     0.288304
min       0.000000     0.000000
25%       0.114440     0.500000
50%       0.228881     1.000000
75%       0.343321     1.000000
max       0.343321     1.000000
```

TODO - describe your approach to feature selection or feature weighting In a text cell, describe **in detail** the approach you used for feature selection or feature weighting. Your answer should include the following parts, in paragraph form:

- **Part 1: Search**: describe the search strategy you use to determine the features or feature subsets to evaluate. Is the approach you chose guaranteed to evaluate the optimal feature subset? How many feature subsets do you need to consider as part of your approach?
- **Part 2: Evaluate**: describe the approach you use to evaluate the "goodness" of a feature or feature subset. Did you use a filter method or a wrapper method? What was the scoring function or model you used to evaluate the "goodness" of a feature or feature subset, and why? And since this dataset has the added complication of missing values, you should also make sure to explain how you handle missing values in your evaluation.
- **Part 3: Number/size**: if you are using feature selection, not feature weighting: Describe the approach you used to select the best **number** of features to include or best **size** of feature

subset.

Also explain: Why is the approach you chose well suited for *this data* and *this model*? And, what are some disadvantages or limitations of the approach you chose?

Part 1: Search

In the method we implemented, we didn't specifically "search" through different subsets of features. Instead, we assigned a weight to each feature based on its relationship with the target variable. The approach is based on Spearman's Rank Correlation, which evaluates the monotonic relationship between each feature and the target. Thus, every feature is evaluated, but they are not necessarily considered in isolation or in subsets. Therefore, the approach is not guaranteed to evaluate the optimal feature subset, as it evaluates each feature independently. The number of feature subsets to consider in this approach is equivalent to the number of features since we're assigning a weight to each feature.

Part 2: Evaluate

The approach we used is a filter method. Filter methods evaluate the relevance of features by their correlation with the dependent variable. The advantage of filter methods is that they are usually faster and less computationally intensive because they do not involve training a model. We used Spearman's Rank Correlation as our scoring function to evaluate the "goodness" of a feature. This method was chosen because it can capture non-linear relationships and is more appropriate for ordinal or discrete data. Regarding missing values, our method inherently handled them since Spearman's Rank Correlation in pandas automatically deals with NaN values by excluding them from the correlation calculation.

Part 3: Number/size

We used feature weighting rather than feature selection. Therefore, we didn't select a specific number of features or subset size. Instead, every feature in the dataset is assigned a weight, which can then be used to scale the feature values. Features with higher weights (indicating higher importance or relevance) have a more significant impact on the model, while those with lower weights have a lesser impact.

1.5.4 Evaluate final classifier

Finally, you'll repeat the process of finding the best number of neighbors using K-fold CV, with your "transformed" data (`X_trans`) and your new custom distance metric.

Then, you'll evaluate the performance of your model on the *test* data, using that optimal number of neighbors.

[114]:
```
# TODO - evaluate - pre-compute 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_trans.iloc[idx_tr[idx]].values,␣
 ↪X_trans.iloc[idx_tr].values)
```

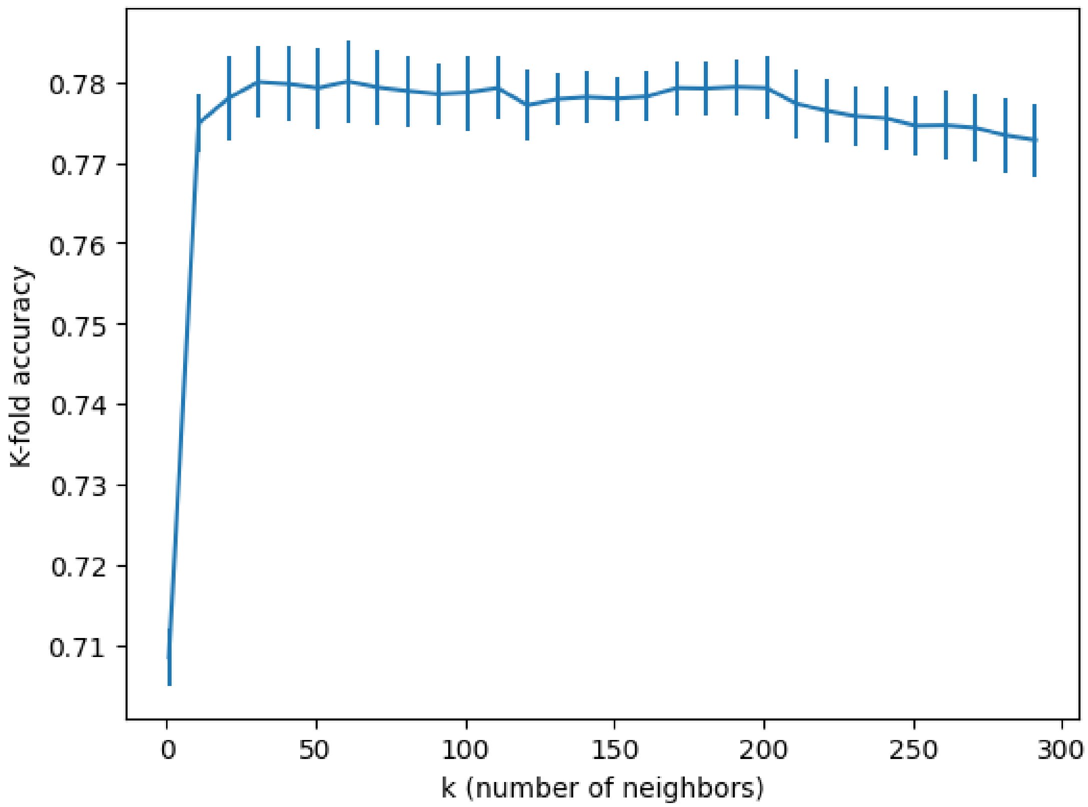
```
Distance matrix: 100%|      | 15958/15958 [02:07<00:00, 125.55it/s]
```

[115]:
```
# TODO - evaluate - use K-fold CV, fill in acc_list

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, shuffle=True)

for isplit, idx_k in enumerate(kf.split(idx_tr)):
  print("Iteration %d" % isplit)
  idx_tr_k, idx_val_k = idx_k
  y_val_kfold = y.iloc[idx_tr[idx_val_k]]
  distances_val_kfold = distances_kfold[idx_val_k[:, None], idx_tr_k]
  r_matrix = np.random.random(size=(distances_val_kfold.shape))
  distances_sorted = np.array([np.lexsort((r, row)) for r, row in zip(r_matrix,␣
 ↪distances_val_kfold)])
  for idx_k, k in enumerate(k_list):
    nn_lists_idx = idx_tr[idx_tr_k[distances_sorted[:,: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)
```

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

See how the validation accuracy changes with number of neighbors:

[116]:
```
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");
```



Find the best choice for k (number of neighbors) using the "highest validation accuracy" rule:

[117]:
```
# TODO - evaluate - find best k
best_k = k_list[np.argmax(acc_list.mean(axis=1))]
print(best_k)
```

```
61
```

Finally, re-run our KNN algorithm using the entire training set and this `best_k` number of neighbors. Check its accuracy on the test data.

[118]:
```
# TODO - evaluate - find accuracy
# compute distance matrix for test vs. training data
distances_test = np.zeros(shape=(len(idx_ts), len(idx_tr)))

for idx in range(len(idx_ts)):
    distances_test[idx] = custom_distance(X_trans.iloc[idx_ts[idx]].values,␣
 ↪X_trans.iloc[idx_tr].values)

r_matrix_test = np.random.random(size=distances_test.shape)
distances_sorted_test = np.array([np.lexsort((r, row)) for r, row in␣
 ↪zip(r_matrix_test, distances_test)])
```

```python
nn_lists_test = distances_sorted_test[:, :best_k]
nn_lists_idx_test = idx_tr[nn_lists_test]
y_pred = [y.iloc[nn].mode()[0] for nn in nn_lists_idx_test]
acc = accuracy_score(y.iloc[idx_ts], y_pred)
```

[119]:
```python
print(acc)
```

```
0.783187134502924
```