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**:
- **Name**:

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.

Import libraries

```
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)
```

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

```
!pip install category_encoders
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: category_encoders in
    /home/ffund/.local/lib/python3.8/site-packages (2.5.0)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.8/dist-packages (from
    category_encoders) (1.19.5)
Requirement already satisfied: patsy>=0.5.1 in /usr/lib/python3/dist-packages (from
    category_encoders) (0.5.1)
Requirement already satisfied: pandas>=1.0.5 in
    /home/ffund/.local/lib/python3.8/site-packages (from category_encoders) (1.4.3)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/lib/python3/dist-packages (from
    category_encoders) (0.22.2.post1)
Requirement already satisfied: scipy>=1.0.0 in /usr/lib/python3/dist-packages (from
    category_encoders) (1.3.3)
```

```
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.8/dist-packages
    (from category_encoders) (0.12.2)
Requirement already satisfied: python-dateutil>=2.8.1 in
    /home/ffund/.local/lib/python3.8/site-packages (from pandas>=1.0.5->category_encoders)
    (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
    /home/ffund/.local/lib/python3.8/site-packages (from pandas>=1.0.5->category_encoders)
    (2022.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from
    python-dateutil>=2.8.1->pandas>=1.0.5->category_encoders) (1.15.0)
WARNING: There was an error checking the latest version of pip.
```

```
import category_encoders as ce
```

Load data

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

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

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

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

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

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

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

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

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.

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.

```
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])))
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-4-292f82be1b7a> in <module>
----> 1 from google.colab import files
      2
      3 uploaded = files.upload()
      4
      5 for fn in uploaded.keys():

ModuleNotFoundError: No module named 'google.colab'
```

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.

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

```
<ipython-input-5-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')
```

```
       ID             PRES                       HOU    WEIGHT @2WAYPRES16  \
0  135355  Hillary Clinton  The Democratic candidate  6.530935
1  135356  Hillary Clinton  The Democratic candidate  6.479016
```

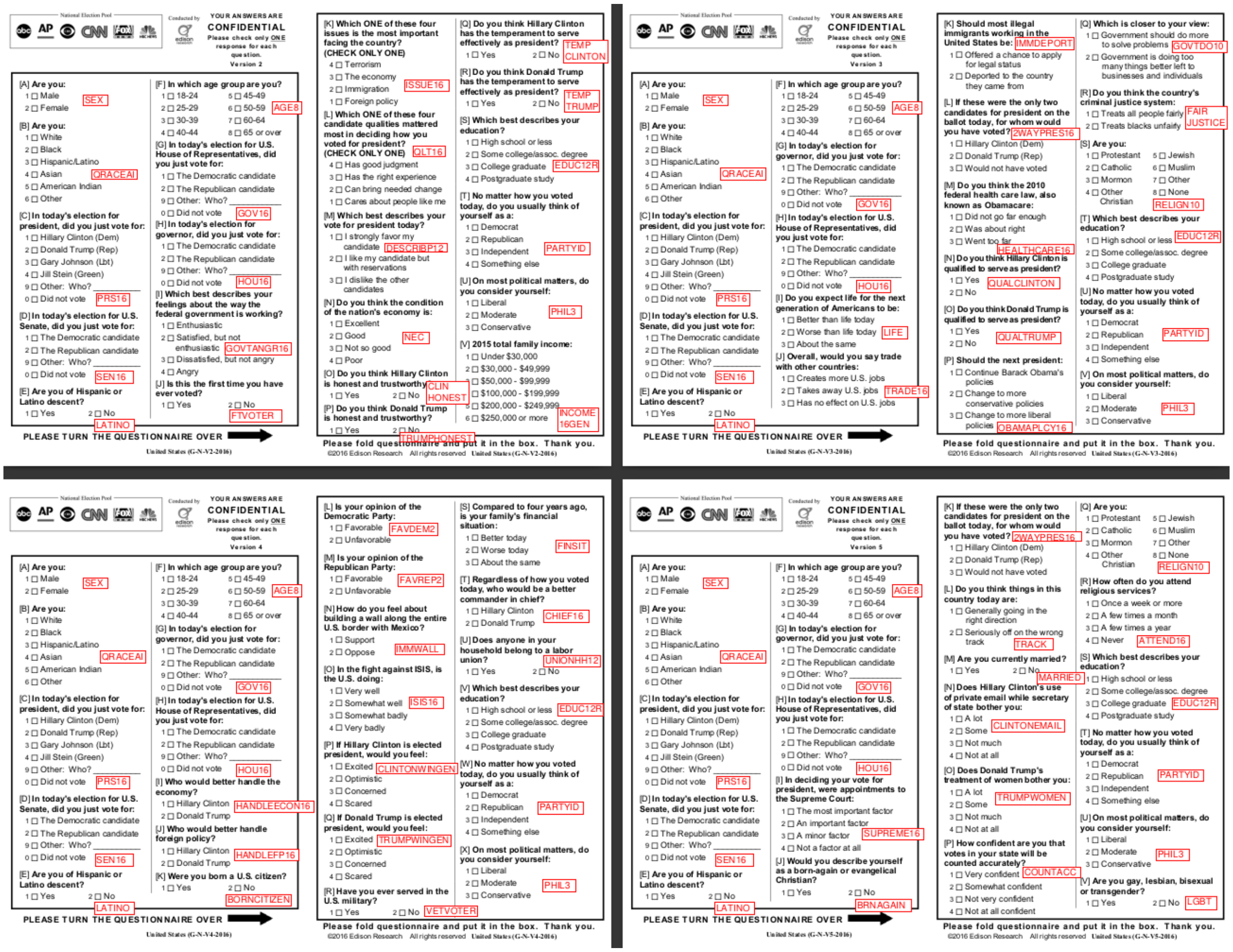
```
2  135357  Hillary Clinton  The Democratic candidate  8.493230
3  135358  Hillary Clinton  The Democratic candidate  3.761814
4  135359  Hillary Clinton  The Democratic candidate  3.470473

     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]
```

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.



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

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

```
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.

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!

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

```
                   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:

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

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

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

```
                  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):

```
df.dropna()
```

```
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.

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:

```
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
```

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

```
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.

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

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

Encode ordinal features

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

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

For example, consider the `AGE` column, in which users select an option from the following:

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

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

```
enc_ord = ce.OrdinalEncoder(handle_missing='return_nan', handle_unknown='return_nan')
enc_ord.fit(df['AGE'])
```

```
OrdinalEncoder(cols=['AGE'], drop_invariant=False, handle_missing='return_nan',
               handle_unknown='return_nan',
               mapping=[{'col': 'AGE', 'data_type': dtype('O'),
```

```
                          'mapping': 18-29    1
30-44    2
45-65    3
65+      4
NaN     -2
dtype: int64}],
               return_df=True, verbose=0)
```

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

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

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

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

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

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

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

We encode using both `AGE` and `EDUC12R`:

```
enc_ord = ce.OrdinalEncoder(handle_missing='return_nan', handle_unknown='return_nan')
enc_ord.fit(df[['AGE', 'EDUC12R']])
df_enc_ord = enc_ord.transform(df[['AGE', 'EDUC12R']])
```

But, look at the mapping between education values and integer encoding:

```
enc_ord.category_mapping
```

```
[{'col': 'AGE',
  'mapping': 18-29    1
  30-44    2
  45-65    3
  65+      4
  NaN     -2
  dtype: int64,
```

```
  'data_type': dtype('O')},
 {'col': 'EDUC12R',
  'mapping': Some college/assoc. degree    1
  College graduate              2
  Postgraduate study            3
  High school or less           4
  NaN                          -2
  dtype: int64,
  'data_type': dtype('O')}]
```

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

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

(Even if the order that the encoder "guesses" is the order you want, you should pass an explicit mapping so that the result is robust against library version updates that may change the order.)

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

features = ['EDUC12R', 'AGE']

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

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

Now, the mapping should be just what we expect:

```
enc_ord.category_mapping
```

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

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

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

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:

```
df_enc_ord.isna().sum()
```

```
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.)

```
scaler = MinMaxScaler()

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

```
df_enc_ord.describe()
```

```
                AGE       EDUC12R
count  22640.000000  21798.000000
mean       0.542609      0.404120
std        0.323963      0.361286
min        0.000000      0.000000
25%        0.333333      0.000000
```

```
50%        0.666667      0.333333
75%        0.666667      0.666667
max        1.000000      1.000000
```

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

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

```
df_enc_ord.isna().sum()
```

```
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.

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!)

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

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

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

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

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

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

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

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

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:

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

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.

```
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`:

```
y.iloc[idx_tr]
```

```
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`:

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

```
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:

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

```
       RACE_Hispanic/Latino    RACE_Asian    RACE_Other    RACE_Black  \
count          15744.000000  15744.000000  15744.000000  15744.000000
mean               0.097561      0.030043      0.031885      0.133067
std                0.296730      0.170712      0.175700      0.339657
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      0.404124
std        0.454951      0.324832      0.360903
min        0.000000      0.000000      0.000000
25%        0.000000      0.333333      0.000000
50%        1.000000      0.666667      0.333333
75%        1.000000      0.666667      0.666667
max        1.000000      1.000000      1.000000
```

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.

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:

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

```
0.5321637426900585
```

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

`KNeighborsClassifier` does not support data with NaNs

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:

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

```
---------------------------------------------------------------------------
ValueError                        Traceback (most recent call last)
<ipython-input-38-e5f2d31b0001> in <module>
      1 clf = KNeighborsClassifier(n_neighbors=3)
----> 2 clf.fit(X.iloc[idx_tr], y.iloc[idx_tr])

/usr/lib/python3/dist-packages/sklearn/neighbors/_base.py in fit(self, X, y)
   1124         """
   1125         if not isinstance(X, (KDTree, BallTree)):
-> 1126             X, y = check_X_y(X, y, "csr", multi_output=True)
   1127
   1128         if y.ndim == 1 or y.ndim == 2 and y.shape[1] == 1:

/usr/lib/python3/dist-packages/sklearn/utils/validation.py in check_X_y(X, y, accept_sparse,
    accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd,
    multi_output, ensure_min_samples, ensure_min_features, y_numeric, warn_on_dtype,
    estimator)
    745         raise ValueError("y cannot be None")
    746
--> 747     X = check_array(X, accept_sparse=accept_sparse,
    748                     accept_large_sparse=accept_large_sparse,
    749                     dtype=dtype, order=order, copy=copy,

/usr/lib/python3/dist-packages/sklearn/utils/validation.py in check_array(array,
    accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d,
    allow_nd, ensure_min_samples, ensure_min_features, warn_on_dtype, estimator)
    575
    576         if force_all_finite:
--> 577             _assert_all_finite(array,
    578                                allow_nan=force_all_finite == 'allow-nan')
    579

/usr/lib/python3/dist-packages/sklearn/utils/validation.py in _assert_all_finite(X,
    allow_nan, msg_dtype)
     55                 not allow_nan and not np.isfinite(X).all()):
     56             type_err = 'infinity' if allow_nan else 'NaN, infinity'
---> 57             raise ValueError(
     58                     msg_err.format
```

```
     59                      (type_err,

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

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!

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:

```python
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):

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

```
       RACE_Hispanic/Latino  RACE_Asian  RACE_Other  RACE_Black  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         0.0         0.0         1.0         0.0
22501                   NaN         NaN         NaN         NaN         NaN

            AGE   EDUC12R
10296  0.666667  0.333333
510    1.000000  0.333333
4827   0.666667  0.000000
20937  0.333333  0.000000
22501  0.666667  0.666667
```

and this set of training data b:

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

```
       RACE_Hispanic/Latino  RACE_Asian  RACE_Other  RACE_Black  RACE_White  \
10379                   NaN         NaN         NaN         NaN         NaN
4343                    0.0         1.0         0.0         0.0         0.0
7359                    0.0         0.0         0.0         0.0         1.0
1028                    0.0         0.0         0.0         1.0         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                    1.0         0.0         0.0         0.0         0.0
11758                   0.0         0.0         0.0         1.0         0.0
13163                   0.0         0.0         0.0         1.0         0.0

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

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.

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

```
(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.

```
# 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, 1246.23it/s]
```

Let's look at those distances now:

```
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 1.67 3.33 0.67 0.67 1.33 1.   1.33 0.67 1.   2.67 2.33 2.33]
 [0.   2.67 3.33 1.67 3.   1.   3.67 2.67 3.67 2.33 3.33 2.33 0.67 0.67]
 [0.   0.33 1.   0.67 0.67 0.67 0.67 1.   0.67 0.67 0.33 0.67 0.33 0.33]]
```

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.

```
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]
```

Example: one test sample

For example, this was the first test sample:

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

```
       RACE_Hispanic/Latino  RACE_Asian  RACE_Other  RACE_Black  RACE_White  \
10296                   0.0         0.0         0.0         0.0         1.0

            AGE   EDUC12R
10296  0.666667  0.333333
```

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

```
distances_custom[0]
```

```
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.

```
distances_sorted[0]
```

```
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:

```
nn_lists[0]
```

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

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

```
nn_lists_idx[0]
```

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

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

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

```
       RACE_Hispanic/Latino  RACE_Asian  RACE_Other  RACE_Black  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  0.333333
131    1.0  0.333333
```

and their corresponding values in `y` are:

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

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

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

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

```
array([1])
```

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`.

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

```
(6840, 15958)
```

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

```
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 [00:12<00:00, 527.76it/s]
```

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

```
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]
```

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

```
0.5497076023391813
```

This classifier seems to improve over the "prediction by mode" classifier… but only *barely* does so.

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:

```
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]
```

```
       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.

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

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

On the other hand, consider these two samples:

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

```
        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.

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

```
array([1.])
```

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?

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

```
[[18099  1527  2412]
 [22393 11973   890]
 [13031 19214 10538]
 [13603 20092 18486]
 [20589 15024 17519]
 [16839  3866  8114]
 [11347  1364 18464]
 [ 6587  2243  9885]
 [10417 10395 19581]
 [ 5209  8446  7693]
 [ 8049 15730 21228]
 [12554  6545   409]
 [20589 15024 17519]
 [10709  3222 18733]
 [12554 17678  1527]
 [11347  1364 18464]
 [12554 17678  1527]
 [11347  1364 18464]
 [ 4808  8509 15235]
 [20429 19473 16645]
 [ 3115 12766  7529]
 [13754 13953 20597]
 [ 1349  1474  8437]
 [11347  1364 18464]
 [ 1349  1474  8437]
 [21240 18388 19637]
 [ 8049 15730 21228]
 [10709  3222 18733]
 [19070  5627 16931]
 [13754 13953 20597]
 [13603 20092 18486]
 [ 1349  1474  8437]
 [11347  1364 18464]
 [16839  3866  8114]
 [ 7149 11931 21321]
 [20429 19473 16645]
 [11347  1364 18464]
 [18278 17012 10432]
 [11347  1364 18464]
 [ 1349  1474  8437]
 [ 1349  1474  8437]
 [10709  3222 18733]
 [ 1349 10056 17430]
 [ 8049 15730 21228]
 [ 4808  8509 15235]
 [11347  1364 18464]
 [13098  5582   286]
 [ 2240 18834  4185]
 [16839  3866  8114]
 [17667  6889  7996]]
```

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

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

```
       RACE_Hispanic/Latino  RACE_Asian  RACE_Other  RACE_Black  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      0.0
10379       NaN      NaN
1883   0.666667      0.0
```

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

```
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'])  ]
```

```
       RACE_Hispanic/Latino  RACE_Asian  RACE_Other  RACE_Black  RACE_White  \
536                     0.0         0.0         0.0         0.0         1.0
1021                    0.0         0.0         0.0         0.0         1.0
1362                    0.0         0.0         0.0         0.0         1.0
2240                    0.0         0.0         0.0         0.0         1.0
2565                    0.0         0.0         0.0         0.0         1.0
2778                    0.0         0.0         0.0         0.0         1.0
3017                    0.0         0.0         0.0         0.0         1.0
3505                    0.0         0.0         0.0         0.0         1.0
3747                    0.0         0.0         0.0         0.0         1.0
5495                    0.0         0.0         0.0         0.0         1.0
7621                    0.0         0.0         0.0         0.0         1.0
10608                   0.0         0.0         0.0         0.0         1.0
11306                   0.0         0.0         0.0         0.0         1.0
18044                   0.0         0.0         0.0         0.0         1.0
20324                   0.0         0.0         0.0         0.0         1.0
20730                   0.0         0.0         0.0         0.0         1.0
20835                   0.0         0.0         0.0         0.0         1.0
21363                   0.0         0.0         0.0         0.0         1.0

       AGE   EDUC12R
536    NaN  0.333333
1021   NaN  0.333333
1362   NaN  0.333333
2240   NaN  0.333333
2565   NaN  0.333333
2778   NaN  0.333333
3017   NaN  0.333333
3505   NaN  0.333333
3747   NaN  0.333333
5495   NaN  0.333333
7621   NaN  0.333333
10608  NaN  0.333333
11306  NaN  0.333333
18044  NaN  0.333333
```

```
20324  NaN  0.333333
20730  NaN  0.333333
20835  NaN  0.333333
21363  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:

```python
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:

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

```
[[19654  5101  4473]
 [22360 18980  8015]
 [21119 16979 13845]
 [17509  8120  2099]
 [19613 10232 16304]
 [15519  6579  8740]
 [18857 10543  9697]
 [ 4730  1022   669]
 [ 2179 13820 17662]
 [ 1858 18769  2766]
 [ 7695  2073  1413]
 [ 3949  4072  5588]
 [18090 15024 16773]
 [ 3791 22248  5572]
 [10453 16192  5549]
 [14834 20881  4740]
 [ 2464 10550 10709]
 [  257 16758 21559]
 [13144 13486 19675]
 [ 5850  5971 21302]
 [17193 16652   613]
 [ 7956 22248 20394]
 [ 2679 14626  2953]
 [ 4513  5211  8442]
 [ 5204  7552  6180]
 [13114 13460  6287]
 [ 9720  9100 11273]
 [ 5402 10670 18200]
```

```
 [17657 10493 21282]
 [14192   744 12992]
 [16299 19313 20644]
 [ 8715  7174  7974]
 [11679 12826  6669]
 [10543 11395 10699]
 [18355 15220  5790]
 [19473 12847 20226]
 [15513 20171 14103]
 [ 5094 19639  5896]
 [10752  6564  8727]
 [ 6284 21646  1494]
 [ 4944  8982 16661]
 [ 8943  5537 18338]
 [ 5644 12617  3818]
 [10282   781 19274]
 [ 4457 10463  6749]
 [ 8237 14517  5106]
 [12955 16838 18792]
 [ 2718 11492  9812]
 [ 8169 11661  9721]
 [14887  1861  7829]]
```

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

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

```
0.5940058479532164
```

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.)

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:

```
# 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 [00:22<00:00, 709.31it/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.

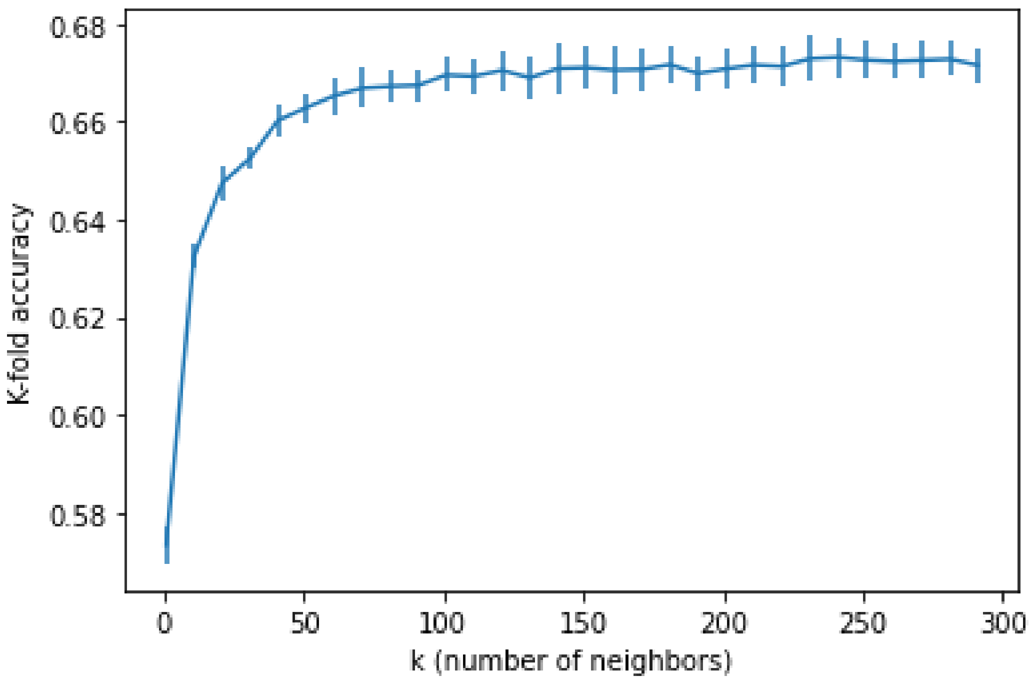
```
n_fold = 5
k_list = np.arange(1, 301, 10)
n_k = len(k_list)
acc_list = np.zeros((n_k, n_fold))

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:

```
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):

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

```
241
```

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

```
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]
```

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

```
0.6779239766081872
```

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.

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.

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

```
# TODO - implement distance metric

def custom_distance(a, b):
  # fill in your solution here!
  # you are encouraged to use efficient numpy functions where possible
  # refer to numpy documentation

  # this is just a placeholder - your function shouldn't actually return
  # all zeros ;)
```

```python
  return np.zeros(b.shape[0])
```

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.

```python
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
```

```python
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:

```python
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
```

```python
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. 0. 0. 0.]]
```

These should have *decreasing* distance:

```python
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
```

```python
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. 0.]]
```

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

These should have *increasing* distance:

```
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
```

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

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:

```
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
```

```
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.]]
```

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:

```
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: use `handle_unknown='return_nan'` in your `OrdinalEncoder` (as in the example), and don't include "Omit" in your `mapping_ord` dictionary. 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.

```
# TODO - encode ordinal features

# set up mapping dictionary and list of features to encode with ordinal encoding
mapping_ord = ...
features_ord = ...

# create an OrdinalEncoder and fit it
...

# use transform to get the encoded columns, save in df_enc_ord
df_enc_ord = ...

# scale each column to the range 0-1
df_enc_ord =
```

```
  File "<ipython-input-86-abcfaf8f9f7f>", line 14
    df_enc_ord =
                ^
```

```
SyntaxError: invalid syntax
```

Look at the encoded data to check your work:

```
df_enc_ord.describe()
```

```
                AGE       EDUC12R
count  22640.000000  21798.000000
mean       0.542609      0.404120
std        0.323963      0.361286
min        0.000000      0.000000
25%        0.333333      0.000000
50%        0.666667      0.333333
75%        0.666667      0.666667
max        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, and will drop the column corresponding to the "Omit" value from the data frame (as shown below).

```
# TODO - encode categorical features

# set up list of features to encode using one-hot encoding
features_oh = ...

# create a OneHotEncoder and fit it
...

# use transform to get the encoded columns, save in df_enc_oh
df_enc_oh = ...

# 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')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-88-2d0d403392be> in <module>
     11
     12 # drop the Omit columns, if any of these are in the data frame
---> 13 df_enc_oh.drop(['ISSUE16_Omit', 'QLT16_Omit',
    'TRACK_Omit','IMMWALL_Omit','GOVTDO10_Omit'],
     14                 axis=1, inplace=True, errors='ignore')
```

```
AttributeError: 'ellipsis' object has no attribute 'drop'
```

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-89-08b0322b924f> in <module>
----> 1 X = pd.concat([df_enc_oh, df_enc_ord], axis=1)

~/.local/lib/python3.8/site-packages/pandas/util/_decorators.py in wrapper(*args, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312 
    313         return wrapper

~/.local/lib/python3.8/site-packages/pandas/core/reshape/concat.py in concat(objs, axis,
    join, ignore_index, keys, levels, names, verify_integrity, sort, copy)
    345     ValueError: Indexes have overlapping values: ['a']
    346     """
--> 347     op = _Concatenator(
    348         objs,
    349         axis=axis,

~/.local/lib/python3.8/site-packages/pandas/core/reshape/concat.py in __init__(self, objs,
    axis, join, keys, levels, names, ignore_index, verify_integrity, copy, sort)
    435                     "only Series and DataFrame objs are valid"
    436                 )
--> 437                 raise TypeError(msg)
    438 
    439             ndims.add(obj.ndim)

TypeError: cannot concatenate object of type '<class 'ellipsis'>'; only Series and DataFrame
    objs are valid
```

```
X.describe()
```

```
       RACE_Hispanic/Latino    RACE_Asian    RACE_Other    RACE_Black  \
count          22488.000000  22488.000000  22488.000000  22488.000000
mean               0.098275      0.030505      0.030283      0.133093
std                0.297692      0.171976      0.171368      0.339683
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  22488.000000  22640.000000  21798.000000
```

```
mean        0.707844        0.542609        0.404120
std         0.454764        0.323963        0.361286
min         0.000000        0.000000        0.000000
25%         0.000000        0.333333        0.000000
50%         1.000000        0.666667        0.333333
75%         1.000000        0.666667        0.666667
max         1.000000        1.000000        1.000000
```

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.

There are many options for feature selection or feature weighting, and you can choose anything that seems reasonable to you - there isn't one right answer here! But, you will have to explain and justify your choice.

For full credit, 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.

```
# TODO - feature selection OR feature weighting

# if you choose feature selection
# feat_inc = ...
# X_trans = X[feat_inc]

# if you choose feature weighting
# feat_wt =
# X_trans = X.multiply(feat_wt)
```

Check your work:

```
X_trans.describe()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-92-d83cba9a6b42> in <module>
----> 1 X_trans.describe()

NameError: name 'X_trans' is not defined
```

TODO - describe your approach to feature selection or feature weighting In a text cell, answer the following questions:

- Describe **in detail** the approach you used for feature selection or feature weighting.
- Consider your approach in the context of our lecture discussion on feature selection/weighting. Is the one you used a wrapper method, a filter method, or an embedded method? Does your take into account redundancy between features, or does it consider each feature to be independent? Explain.
- Why is the approach you chose well suited for *this data* and *this model*?

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.

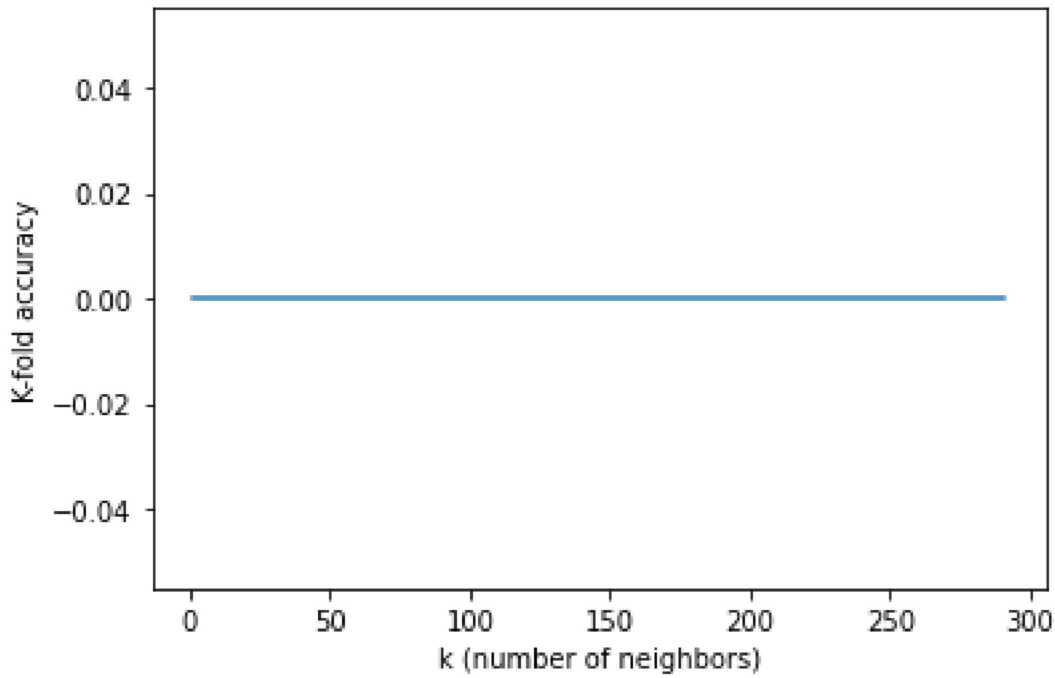
```
# TODO - evaluate - pre-compute distance matrix of training vs. training data

distances_kfold = ...
```

```
# 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))

# use this random state so your results will match the auto-graders'
kf = KFold(n_splits=5, shuffle=True, random_state=3)

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

  # Outer loop

  for idx_k, k in enumerate(k_list):

    # Inner loop

    acc_list[idx_k, isplit] = ...
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-94-138220f04d9d> in <module>
     17     # Inner loop
     18
---> 19     acc_list[idx_k, isplit] = ...

TypeError: float() argument must be a string or a number, not 'ellipsis'
```

See how the validation accuracy changes with number of neighbors:

```
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:

```
# TODO - evaluate - find best k
best_k = ...
```

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.

```
# TODO - evaluate - find accuracy
# compute distance matrix for test vs. training data
# use KNN with best_k to find y_pred for test data
y_pred = ...
# compute accuracy
acc = ...
```

```python
print(acc)
```

```
Ellipsis
```