Assignment 1: Text Classification

In this assignment, you'll train a text classifier. You'll prepare data as input into the classifier, construct the classifier itself, and examine its performance. You'll also gain familiarity with the scikit-learn (https://scikit-learn.org/stable/) library, which we'll use to do all of these things.

For most of the assignments in this course, we'll use data from English Wikipedia. Wikipedia articles (obviously) contain lots of text, along with other useful bits of information. In particular, many Wikipedia articles contain infoboxes that summarize their contents. When an article is about a person, the infobox often has a field listing their *nationality*, as determined by the article writers.

We will consider the following task:

Given the text of a person's Wikipedia biography, determine the person's nationality (per what's listed in the infobox).

The texts of biographies contain many hints about peoples' nationalities, so a good starting intuition is that this task is *plausible*. Before starting this assignment, think about what those hints might be.

However, nationality labels are noisy. People move around. What counts as a nation, and what counts a person as a member of a nation, has been disputed throughout history. Our scraping procedure -- i.e., the code we wrote to download Wikipedia articles and extract the relevant infobox fields -- isn't perfect, because real data is messy and accounting for all edge cases is generally impossible.

So, the task, while plausible, probably won't be *easy*, and we should be thoughtful about what our classification errors represent.

In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline

import numpy as np
import re
import pandas as pd
import seaborn as sns
from collections import Counter
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, confusion_matrix
from sklearn.model_selection import train_test_split
from tqdm import tqdm
from nltk.tokenize import word_tokenize
from IPython.display import display
```

A lot of the code you'll write and run in this class has non-deterministic output. To ensure your output matches what our autograder expects, we'll rely on a random seed. Most functions with some randomness provide an optional argument for setting the random seed. Throughout all of the assignments, be sure to set that argument:

In [2]:
```
RANDOM_SEED = 655
```

Part 1: Data processing

We'll start by loading and preprocessing the data. Important point: it takes work to get data into a format that enables further analysis. It's essential that you look at what's in the data to spot messy bits that can throw off later analyses, and then take steps to clean the data up.

We'll load the data for you:

In [3]:
```
nationality_df = pd.read_csv('../../assets/bio_nationality.tsv.gz', sep='\t', compression='gzip', index_col=0)
# skip empty entries:
nationality_df = nationality_df.dropna()
# we'll only use a subset of the data to save on runtime and memory
nationality_df = nationality_df[:75000]
```

The data is stored in a pandas dataframe:

In [4]:
```
nationality_df.sample(5, random_state=RANDOM_SEED)
```

Out[4]:

	bio	nationality
46957	Early life and education\nDiskin was born in G...	israelis, israeli
39447	Biography\nMarjanović was born into a large wo...	serbian
57804	Michel Demaret (18 January 1940 - 9 November 2...	belgium
72739	Life\n'Profile Head with Cap', woodcut on pape...	american
23793	Life\nBeddoes was born in Shifnal on 13 April ...	british

The input data has over 5,000 unique labels:

In [5]:
```
print(nationality_df.nationality.nunique())
assert len(set(nationality_df.nationality)) == 5374
```

```
5374
```

To get a better sense of what the labels are like, we'll print out the most common labels:

In [6]:
```
nationality_df.nationality.value_counts().head(20)
```

Out[6]:
```
american                  14273
british                    5522
united states, american    2968
australian                 2247
united states              1729
indian                     1524
english                    1435
americans, american        1339
canadians, canadian        1298
french                     1289
canadian                   1284
united kingdom, british    1184
german                     1104
japanese                   1038
usa                         996
italian                     927
british people, british     846
polish                      705
irish people, irish         630
irish                       561
Name: nationality, dtype: int64
```

In [ ]:
```

```

Hopefully you've noticed a problem: nationalities appear to be listed in varied formats, and there's quite a bit of redundancy.

Task 1.1: Inspect labels

Set `redundant_entries` to be a two-item list containing two labels that you think are redundant, i.e., probably refer to the same nationality.

In [7]:
```
redundant_entries = ["american", "united states, american"]
# YOUR CODE HERE

#raise NotImplementedError()
```

In [8]:
```
print(redundant_entries)
#hidden tests are within this cell
```

```
['american', 'united states, american']
```

What problems do you think will arise if we keep these redundant labels separate, instead of standardizing them?

Task 1.2: Standardize the nationality labels

We'll implement the following quick fix to standardize the nationality labels and hopefully remove much of the redundancy we just noted:

- split the nationality labels at each comma `,`

- take the last word of the split output
- remove whitespace around the resultant label

The python functions `split` and `strip` should help here.

These steps will be good enough for the purposes of this assignment. They do not 100% address the label format inconsistency issue. For real-world text datasets, data cleaning rarely fixes *everything*, and practitioners frequently must make judgement calls on how much effort to spend, to get the data to a good *enough* state.

In [9]:
```
# YOUR CODE HERE
def standardize_nationality(label):
    # Split the label at each comma
    parts = label.split(',')

    # Take the last part and remove any whitespace
    standardized_label = parts[-1].strip()

    return standardized_label


nationality_df['nationality'] = nationality_df['nationality'].apply(standardize_nationality)

#raise NotImplementedError()
```

Let's count the number of standardized nationalities, after applying our heuristic fix:

In [10]:
```
print(len(set(nationality_df.nationality)))
#hidden tests are within this cell
```

```
3197
```

Let's print out the new list of most common nationalities.

In [11]:
```
nationality_df.nationality.value_counts().head(20)
```

Out[11]:
```
american         19536
british           7991
canadian          2723
australian        2351
french            2066
english           2050
german            1912
indian            1789
united states     1784
italian           1599
japanese          1334
irish             1323
polish            1028
usa               1017
scottish           785
russian            760
dutch              715
spanish            649
swedish            519
norwegian          481
Name: nationality, dtype: int64
```

This hopefully looks better! That said, you might still notice a few problems.

Task 1.3: Inspect standardized labels

Set `redundant_standardized_labels` to be a two-item list containing two labels from the standardized data that you think are (still) redundant.

In [12]:
```
redundant_standardized_labels = ["american", "usa"]
# YOUR CODE HERE
#raise NotImplementedError()
```

In [13]:
```
print(redundant_standardized_labels)
#hidden tests are within this cell
```

```
['american', 'usa']
```

Keep these examples in mind -- they may be relevant later on in the assignment.

When training classifiers, we need a sufficient number of examples per class. If a particular nationality label rarely occurs in the data, there may be not enough training instances for the model to learn what's informative. It's also harder to get an estimate of the model's performance on infrequent labels (consider an extreme case where there is only item of a particular label -- if the model happens to randomly guess that label correctly, is that meaningful?)

Task 1.4: Filter out infrequent labels

- Get the subset of rows in `nationality_df` corresponding to labels which occur **at least 500 times in the data**.
- Call this subset `cleaned_nationality_df` .

In [14]:
```
MIN_NATIONALITY_COUNT = 500
# YOUR CODE HERE



label_counts = nationality_df['nationality'].value_counts()

# Find labels that occur at least MIN_NATIONALITY_COUNT times
frequent_labels = label_counts[label_counts >= MIN_NATIONALITY_COUNT].index.tolist()

# Filter the DataFrame to include only rows with frequent labels
cleaned_nationality_df = nationality_df[nationality_df['nationality'].isin(frequent_labels)]


#raise NotImplementedError()
```

Here's how much data and how many labels we'll work with for the rest of the assignment:

In [15]:
```
print(len(cleaned_nationality_df), cleaned_nationality_df.nationality.nunique())
#hidden tests are within this cell
```

```
51931 19
```

In [82]:
```
cleaned_nationality_df.head(6)
```

Out[82]:

	bio	nationality
0	Alain Connes (born 1 April 1947) is a French m...	french
1	Life\n=== Early life ===\nSchopenhauer's birth...	german
2	Life and career\nAlfred Nobel at a young age i...	swedish
3	Early life\nAlfred Vogt (both "Elton" and "van...	canadian
4	Alfons Maria Jakob (2 July 1884 in Aschaffenbu...	german
5	Biography\nAndrey Markov was born on 14 June 1...	russian

In [81]:
```
print(cleaned_nationality_df.loc[5, 'bio'])
```

```
Biography
Andrey Markov was born on 14 June 1856 in Russia. He attended Petersburg Grammar, where he was seen as a rebellious student by a select few teachers.
In his academics he performed poorly in most subjects other than mathematics.  Later in life he attended Petersburg University; among his teachers wer
e Yulian Sokhotski (differential calculus, higher algebra), Konstantin Posse (analytic geometry), Yegor Zolotarev (integral calculus), Pafnuty Chebysh
ev (number theory and probability theory), Aleksandr Korkin (ordinary and partial differential equations), Mikhail Okatov (mechanism theory), Osip Som
ov (mechanics), and Nikolai Budaev (descriptive and higher geometry).  He completed his studies at the University and was later asked if he would like
to stay and have a career as a Mathematician. He later taught at high schools and continued his own mathematical studies. In this time he found a prac
tical use for his mathematical skills. He figured out that he could use chains to model the alliteration of vowels and consonants in Russian literatur
e. He also contributed to many other mathematical aspects in his time. He died at age 66 on 20 July 1922.
```

Task 1.5: Create train/dev/test data splits

Now we'll split our data in `cleaned_nationality_df` into train, development and test sets, using the standard proportions of 80%, 10% and 10% respectively.

- Use scikit-learn's `train_test_split` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html).
- Call the subsets `train_df`, `dev_df` and `test_df`

Hint: `train_test_split` will only produce two splits. Your code will call the function twice: first split the data into (train+dev) | (test), and then split (train+dev) into train | dev. Be sure to correctly calculate the proportion to include in each split, for each function call.

Remember the set the random state argument!

In [16]:
```
TRAIN_SIZE = .8
DEV_SIZE = .1
TEST_SIZE = .1

train_dev_df, test_df = train_test_split(cleaned_nationality_df, test_size=TEST_SIZE, random_state=RANDOM_SEED)

dev_proportion_of_train_dev = DEV_SIZE / (TRAIN_SIZE + DEV_SIZE)
train_df, dev_df = train_test_split(train_dev_df, test_size=dev_proportion_of_train_dev, random_state=RANDOM_SEED)

# YOUR CODE HERE
#raise NotImplementedError()
```

Here's how much data ends up in each split:

(the autograder will check that your splits are the same as ours.)

In [17]:
```
print(len(train_df), len(dev_df), len(test_df))
```

```
41544 5193 5194
```

Part 2: Tokenizing the data

To use a text classifier, we have to come up with features that represent the text. To featurize the text, we start by tokenizing it -- i.e., splitting the text into words.

There are many ways to tokenize text which will result in slightly different outputs. We'll explore some of these methods below.

Task 2.1: Implement various tokenizers

We'll compare three tokenization methods, which you'll implement below:

1. `simple_split` : Separate the text by whitespace, using the default string `split()` method.
2. `sklearn_split` : Split the text into tokens that match the regular expression used by scikit-learn's `TfidfVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) class to tokenize text.
    - You can find this regex by reading the documentation for either class.
    - The `re.findall` method will also be useful.

3. `nltk_split` : Use the nltk library's `word_tokenize` (https://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize) function.

By the way -- for the earlier parts of the assignment we'll link you to the documentation of these libraries, for convenience. As you get the hang of things, you should get into the habit of looking up documentation for yourself.

In [18]:
```
def simple_split(text):

    return text.split()
# YOUR CODE HERE

#raise NotImplementedError()

def sklearn_split(text):

    token_pattern = r"(?u)\b\w\w+\b"
    return re.findall(token_pattern, text)
# YOUR CODE HERE
#raise NotImplementedError()

def nltk_split(text):
    return word_tokenize(text)

# YOUR CODE HERE
#raise NotImplementedError()
```

In [19]:
```
split_tokens = Counter()
sklearn_regex_tokens = Counter()
nltk_tokens = Counter()

# to speed things up we'll only tokenize a subset of the data.
wordcount_subset = train_df.head(1000)

for bio in tqdm(wordcount_subset.bio):

    # we'll lowercase the text
    bio = bio.lower()

    split_tokens.update(simple_split(bio))

    sklearn_regex_tokens.update(sklearn_split(bio))

    nltk_tokens.update(nltk_split(bio))
```

```
100%|██████████| 1000/1000 [00:03<00:00, 318.56it/s]
```

How many different types of tokens do we get for each method?

In [20]:
```
print('simple split:', len(split_tokens),
      '\nsklearn:', len(sklearn_regex_tokens),
      '\nnltk:', len(nltk_tokens))
```

```
simple split: 61341
sklearn: 33501
nltk: 39106
```

Let's look more closely at the difference between these methods. In particular, we'll compare the split vs scikit-learn methods, and the scikit-learn vs the nltk methods. The following code will identify and count the number of tokens that one method produces and that the other does not.

In [21]:
```
split_sklearn_diff = Counter({w: c for w,c in split_tokens.items() if w not in sklearn_regex_tokens})
sklearn_split_diff = Counter({w: c for w,c in sklearn_regex_tokens.items() if w not in split_tokens})

nltk_sklearn_diff = Counter({w: c for w,c in nltk_tokens.items() if w not in sklearn_regex_tokens})
sklearn_nltk_diff = Counter({w: c for w,c in sklearn_regex_tokens.items() if w not in nltk_tokens})
print('%d tokens from split that sklearn does not output. most frequent:' % len(split_sklearn_diff))
print(split_sklearn_diff.most_common(20))
print()
print('%d tokens from sklearn that split does not output. most frequent:' % len(sklearn_split_diff))
print(sklearn_split_diff.most_common(20))
print('\n')
print('%d tokens from nltk that sklearn does not output. most frequent:'% len(nltk_sklearn_diff))
print(nltk_sklearn_diff.most_common(20))
print()
print('%d tokens from sklearn that nltk does not output. most frequent:'% len(sklearn_nltk_diff))
print(sklearn_nltk_diff.most_common(20))
```

```
35573 tokens from split that sklearn does not output. most frequent:
[('a', 9979), ("'the", 664), ('i', 417), ('-', 333), ('0', 228), ('*', 204), ('"the', 188), ('however,', 186), ('(born', 174), ('===', 171), ('1', 17
0), ('year,', 154), ('school,', 151), ('4', 148), ('college,', 147), ('u.s.', 144), ('3', 138), ('2', 134), ('st.', 131), ('university,', 123)]

7733 tokens from sklearn that split does not output. most frequent:
[('000', 242), ('anti', 80), ('sts', 58), ('jr', 50), ('hara', 44), ('com', 41), ('didn', 33), ('pre', 31), ('sr', 22), ('wasn', 21), ('fg', 21), ('0
8', 20), ('05', 19), ('09', 19), ('ll', 17), ('multi', 17), ('gaudens', 16), ('07', 16), ('couldn', 15), ('667', 14)]


6838 tokens from nltk that sklearn does not output. most frequent:
[(',', 28697), ('.', 18652), ('a', 10164), ("'", 7698), ('``', 6789), ("'s", 3524), ('(', 3248), (')', 3246), ('"', 1436), (':', 879), (';', 686),
('i', 627), ('*', 440), ('-', 333), ('1', 270), ('===', 245), ('2', 235), ('0', 231), ('4', 204), ('3', 203)]

1233 tokens from sklearn that nltk does not output. most frequent:
[('000', 242), ('anti', 80), ('sts', 58), ('hara', 44), ('com', 41), ('pre', 31), ('sr', 22), ('wasn', 21), ('cannot', 19), ('09', 19), ('multi', 17),
('07', 16), ('667', 14), ('00', 12), ('neill', 12), ('429', 12), ('miou', 12), ('731', 11), ('aquisto', 11), ('connell', 10)]
```

What do you think is the difference between each method? A few strategies to figure out:

- Read the relevant documentation for each function
- Write code to inspect differences in the above output by looking within the various Counters we produced
- Generate some test inputs and run them through each method

(a hint: consider what each method does to contractions like "can't")

Task 2.1.1: Compare the tokenizers

To test your understanding, come up with an input string `input_str` that results in a tokenized output with a **different number of tokens for each method**.

In [22]:
```
input_str = "Dr. Fu's email is dr.fu@gmail.com, and her office number is 123-456-7890. She'll meet at 10 a.m. - Note: Bring the 3-D glasses."


# set the value of input_str below:
# YOUR CODE HERE

#raise NotImplementedError()
```

In [23]:
```
simple_output = simple_split(input_str)
sklearn_output = sklearn_split(input_str)
nltk_output = nltk_split(input_str)

print('split output: ', simple_output)
print('sklearn output: ', sklearn_output)
print('nltk output: ', nltk_output)
print()
print('number of tokens outputted: (should be different numbers)')
print(len(simple_output), len(sklearn_output), len(nltk_output))
```

```
split output:  ['Dr.', "Fu's", 'email', 'is', 'dr.fu@gmail.com,', 'and', 'her', 'office', 'number', 'is', '123-456-7890.', "She'll", 'meet', 'at', '1
0', 'a.m.', '-', 'Note:', 'Bring', 'the', '3-D', 'glasses.']
sklearn output:  ['Dr', 'Fu', 'email', 'is', 'dr', 'fu', 'gmail', 'com', 'and', 'her', 'office', 'number', 'is', '123', '456', '7890', 'She', 'll', 'm
eet', 'at', '10', 'Note', 'Bring', 'the', 'glasses']
nltk output:  ['Dr.', 'Fu', "'s", 'email', 'is', 'dr.fu', '@', 'gmail.com', ',', 'and', 'her', 'office', 'number', 'is', '123-456-7890', '.', 'She',
"'ll", 'meet', 'at', '10', 'a.m.', '-', 'Note', ':', 'Bring', 'the', '3-D', 'glasses', '.']

number of tokens outputted: (should be different numbers)
22 25 30
```

When testing and understanding text-processing code, it's useful to try to come up with such "minimal test cases"!

A few practical notes.

- `nltk` implements a slightly fancier tokenizer than the other two methods, at the price of a slower runtime. In assignment 3 you'll encounter a similar tokenizer from the `spacy` library. (Also, different `nltk` versions might produce slightly different output from each other, so beware of that.)
- you might find that scikit-learn output is slightly cleaner than the simple split approach.
- in a pinch (i.e., if you want to quickly run/test something), the simple approach might work just fine -- you'll need to make a call based on your particular use-case.

For the rest of the assignment, we'll stick with scikit-learn's tokenizer:

In [24]:
```
token_counts = sklearn_regex_tokens
```

Task 2.2: Examine token frequencies

How often does each word (per scikit-learn) occur in the data sample we considered? More precisely, we'll examine the *frequency* of each token, i.e., the fraction of tokens that each word accounts for.

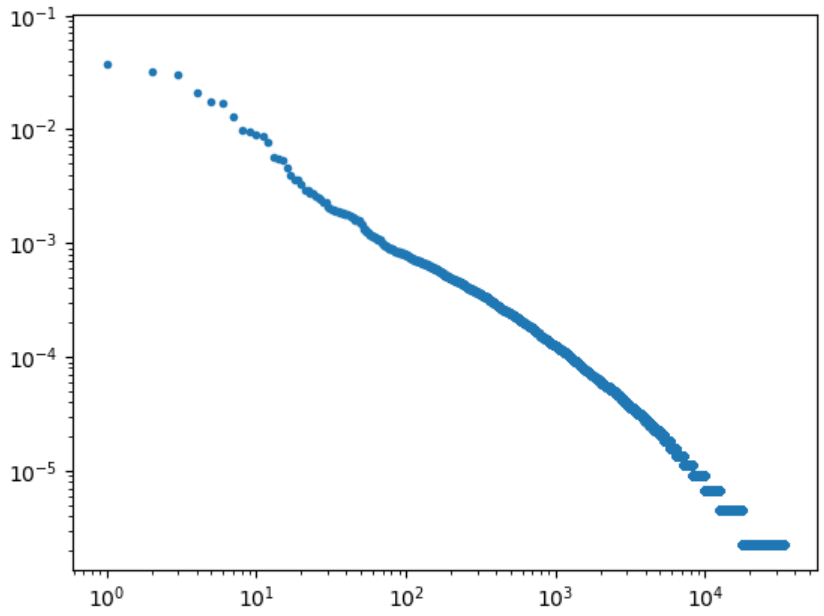
- create a list, `freqs`, where each entry contains the frequency of each token. You'll compute this from `token_counts`.
- sort `freqs` in descending order, such that its first entry contains the highest frequency, corresponding to the most frequently-occurring word

In [25]:
```
total_tokens = sum(token_counts.values())

# Calculate the frequency of each token (as a fraction of the total number of tokens)
freqs = [count / total_tokens for token, count in token_counts.items()]

# Sort the frequencies in descending order
freqs.sort(reverse=True)

# YOUR CODE HERE
#raise NotImplementedError()
```

In many text corpora (including ours), word frequencies follow a power-law like distribution, a phenomenon known as Zipf's Law (https://en.wikipedia.org/wiki/Zipf%27s_law). In essence, a few words are *very* common and account for most of the tokens we have in the data; most other words are relatively rare.

To see Zipf's law in action, the code below plots the frequency of each word, ordered from most-to-least frequent words. We'll log-scale both the x and y axes. What "power-law distribution" means is that once we log-scale the axes, the resultant graph should resemble a straight line.

In [26]:
```
plt.plot(freqs, '.')
plt.yscale('log')
plt.xscale('log')
```



In [ ]:
```

```

Part 3: Train and evaluate classifiers

Now let's build our nationality classifier.

For later use, we'll store the train/dev/test labels in the following lists:

In [27]:
```
y_train = list(train_df.nationality)
y_dev = list(dev_df.nationality)
y_test = list(test_df.nationality)
```

Task 3.1: Generate tf-idf reweighted bag-of-words features

First we need to convert each biography into features. A straightforward starting point is as follows:

- split the biographies into words (perhaps with one of the tokenizers we just implemented)
- for each biography, count the number of times each word occurs in it
- construct a feature matrix, where rows represent biographies, columns represent words, and each entry of the matrix stores the number of times each word occurs in each biography.

A standard improvement on this starting point, which we'll use, is to slightly adjust the values of this feature matrix using *tf-idf* reweighting, which assumes that words which occur in fewer documents are more distinctive and or informative (quantified via the *idf*), and upweights them.

Fortunately, the `TfidfVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) class from scikit-learn does all of this for you, with optimizations for speed and memory efficiency. `TfidfVectorizer` comes with a lot of parameters, and two will be relevant here:

- we have a lot of words, and accounting for all of them would result in gigantic feature vectors. That puts a strain on our computer's memory usage and performance. However, as we saw in the graph above, *most* words are relatively rare, and our classifier's performance shouldn't suffer too much if we ignore these words (instead of accounting for them as features). To filter out rare words, we'll use the `min_df` parameter and only consider words that occur at least 500 times in the training data.
- there are a few very common words that appear in most biographies, such as "the" and "a". In NLP these are known as *stop words*. It's often the case that these features don't add much information, so for this assignment we'll filter them out. In particular, we'll use the `stop_words` parameter and specify scikit-learn's default `english` list of stop words.

Below, you will:

- create a `TfidfVectorizer` object called `vectorizer` with those parameters.
- "fit" `vectorizer` on the biographies in the training data, `train_df.bio` . (this will figure out what words are in the training data, and hence, among other things, how big the feature matrix will be. It will also compute some summary statistics that are required to calculate idf scores -- think about what those may be.)
- "transform" the biographies in `train_df` using `vectorizer` -- i.e., convert them to the matrix of wordcount features. Call this matrix `X_train` .

In [28]:
```
MIN_DF = 500

vectorizer = TfidfVectorizer(min_df=MIN_DF, stop_words='english')

vectorizer.fit(train_df.bio)

X_train = vectorizer.transform(train_df.bio)



# YOUR CODE HERE
#raise NotImplementedError()
```

Let's look at the size of the matrix, and the vocabulary of words represented.

In [29]:
```
print('size:' ,X_train.shape)
print('vocab:' ,vectorizer.get_feature_names_out())
```

```
size: (41544, 2760)
vocab: ['000' '08' '10' ... 'younger' 'youngest' 'youth']
```

Technical note.

The `X_train` matrix has a few special properties:

- It's very big (there are lots of documents and words within the document).
- It's mostly zeroes (most words do not appear in most documents, cf Zipf's law) -- so it's a *sparse matrix*.

This means that:

- if we were to simply store `X_train` as a regular matrix, we'd use up a *lot* of memory, since it's very big.
- it would be more efficient to simply keep track of where its non-zero entries are, and the values of these entries.

Accordingly, `TfidfVectorizer` outputs *sparse matrices* (https://docs.scipy.org/doc/scipy/reference/sparse.html), which are optimized to store big matrices with lots of zeros.

In [30]:
```
print("here's how many values we'd need to store if we represented X_train as a regular matrix")
print('i.e., the number of squares in the matrix, or # rows times # columns')
print(X_train.shape[0] * X_train.shape[1])
print()
print("here's how many values we'd need to store if we represented X_train as a sparse matrix")
print(len(X_train.nonzero()[0]))
print("(with a bit of overhead for the sparse matrix data structure...don't worry about that though)")
print("note this number is much smaller")
```

```
here's how many values we'd need to store if we represented X_train as a regular matrix
i.e., the number of squares in the matrix, or # rows times # columns
114661440

here's how many values we'd need to store if we represented X_train as a sparse matrix
4446231
(with a bit of overhead for the sparse matrix data structure...don't worry about that though)
note this number is much smaller
```

In [ ]:
```

```

Task 3.2: Featurize dev data

Now let's generate the feature representations for the dev data. Use `vectorizer` to compute feature representations for the biographies from `dev_df`; call the output `X_dev`.

Note: one of the methods mentioned above, `fit`, `transform` and `fit_transform` will be helpful here. Choose wisely. Importantly, we do *not* "re-fit" `vectorizer` on the dev data.

In [31]:
```
# YOUR CODE HERE
X_dev = vectorizer.transform(dev_df.bio)
#raise NotImplementedError()
```

In [32]:
```
print(X_dev.shape)
```

```
(5193, 2760)
```

Task 3.3: Train classifier

Now let's train a logistic regression classifier, using the scikit-learn class `LogisticRegression`.

- create a `LogisticRegression` object called `clf`. Don't forget to set the random state so your results are consistent from run to run, and (importantly) with the autograder.
- fit this classifier on the training data: the feature matrix `X_train`, and the labels `y_train`.

In [33]:
```
# YOUR CODE HERE



clf = LogisticRegression(random_state=RANDOM_SEED)


clf.fit(X_train, y_train)

#raise NotImplementedError()
```

Out[33]:
```
LogisticRegression(random_state=655)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

Here's what the model has learned -- more precisely, the feature weights. They won't make much sense right now, but we'll come back to them later.

In [34]:
```
print('labels:', clf.classes_)
print(clf.intercept_)
print(clf.coef_)
```

```
labels: ['american' 'australian' 'british' 'canadian' 'dutch' 'english' 'french'
 'german' 'indian' 'irish' 'italian' 'japanese' 'polish' 'russian'
 'scottish' 'spanish' 'swedish' 'united states' 'usa']
[ 2.22073082e+00  5.16113232e-03  1.19515782e+00  3.92137722e-01
 -6.07328112e-01  1.37917038e-01  2.50274686e-01  4.13356398e-01
  3.28063058e-03 -6.73008625e-01  1.70035556e-01  2.55698752e-01
 -6.42392014e-01 -6.37337274e-01 -9.24112636e-01 -7.38189763e-01
 -1.01581167e+00  1.95246266e-01 -8.16717886e-04]
[[ 0.51831913  0.48983629  0.017632   ... -1.08613744  0.86943819
  -0.95471675]
 [-0.49198722  0.14285784  0.01195855 ...  0.37032932  0.42430038
  -0.29727517]
 [-0.29855912 -0.29331917 -0.43682612 ...  0.64282624 -0.38625675
  -0.33945469]
 ...
 [ 0.0859758  -0.06127969 -0.54019108 ...  0.18363471  0.18868035
   0.07423562]
 [-0.88161826 -0.13396403 -0.76672775 ... -0.32053532 -0.048031
   0.07207912]
 [-0.35402418  0.08291726 -0.14750007 ...  0.18995987 -0.38702355
   0.17016547]]
```

In [ ]:
```

```

Task 3.4: Train baseline classifiers

It's always important to contextualize the performance of your model by comparing it to simpler baselines. If the baselines already do pretty well, your task is probably very easy, and you might want to consider a simpler approach, or a different choice of metric that better quantifies performance on the hard parts.

We'll consider two baselines, which we implement using the DummyClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html) class:

- randomly guess nationalities, with equal probability for each label. Call this classifier `uniform_clf`, and fit it to `X_train` and `y_train`.
- randomly guess nationalities, outputting guesses corresponding to the frequency that each label occurs in the training data (so more frequent labels are guessed more often). Call this classifier `stratified_clf`, and fit it to `X_train` and `y_train`.

These are *very simple* baselines: they don't even require that you look at the biographies!

Note that these two baselines correspond to two different settings of the `strategy` parameter of `DummyClassifier`, which you should determine by consulting the documentation.

Be sure to set the `random_state` parameter of each `DummyClassifer` to be `RANDOM_SEED`.

In [35]:
```
# YOUR CODE HERE
uniform_clf = DummyClassifier(strategy='uniform', random_state=RANDOM_SEED)
uniform_clf.fit(X_train, y_train)


stratified_clf = DummyClassifier(strategy='stratified', random_state=RANDOM_SEED)
stratified_clf.fit(X_train, y_train)
#raise NotImplementedError()
```

Out[35]:
```
DummyClassifier(random_state=655, strategy='stratified')
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

Here's a few examples of what the model guesses, per each baseline:

In [36]:
```
print(uniform_clf.predict(X_dev[:25]))
print(stratified_clf.predict(X_dev[:25]))
```

```
['german' 'united states' 'irish' 'russian' 'irish' 'french' 'russian'
 'japanese' 'canadian' 'american' 'japanese' 'indian' 'indian' 'russian'
 'usa' 'irish' 'french' 'russian' 'united states' 'swedish' 'english'
 'usa' 'japanese' 'indian' 'italian']
['british' 'british' 'american' 'american' 'dutch' 'german' 'british'
 'american' 'italian' 'indian' 'american' 'german' 'american' 'american'
 'american' 'american' 'italian' 'american' 'british' 'usa' 'american'
 'american' 'american' 'united states' 'american']
```

Comparing the above outputs might help you get a more concrete sense of what the uniform and stratified baselines are doing.

Task 3.4.1: Interpret baseline classifiers

Set `most_freq_guess` as the most frequent label you expect will be predicted by `stratified_clf`, and `most_freq_pr` as the proportion of predicted labels you'd expect are equal to `most_freq_guess`. (Note: the autograder has a fairly generous tolerance here. That said, you should *not* need to call `stratified_clf.predict(X_dev)`, i.e., run the classifier on the entire dev set, to estimate these two values. In fact, if you understand what the baseline classifier is doing, and what the input data looks like, you shouldn't even need to manually write down an actual number for `most_freq_pr`.)

In [37]:
```
# Count the frequency of each label in y_train
label_counts = Counter(y_train)

# Find the most frequent label
most_freq_guess = label_counts.most_common(1)[0][0]

# Calculate the proportion of the most frequent label
most_freq_pr = label_counts[most_freq_guess] / len(y_train)




# YOUR CODE HERE
#raise NotImplementedError()
```

In [38]:
```
print(most_freq_guess, most_freq_pr)
```

```
american 0.37750336992104755
```

Task 3.5: Generate model predictions

Let's generate our predictions. We have three models: our `LogisticRegression` model and the two `DummyClassifier` baselines.

On the dev data, predict the nationality label for each person and store the predictions as:

- `lr_dev_preds` (logistic regression)
- `uniform_dev_preds` (uniform random guessing)
- `stratified_dev_preds` (stratified random guessing

In [39]:
```
# YOUR CODE HERE
lr_dev_preds = clf.predict(X_dev)

uniform_dev_preds = uniform_clf.predict(X_dev)


stratified_dev_preds = stratified_clf.predict(X_dev)
#raise NotImplementedError()
```

Some prediction output:

In [40]:
```
print(lr_dev_preds[:25])
```

```
['british' 'british' 'indian' 'american' 'spanish' 'american' 'russian'
 'american' 'irish' 'indian' 'american' 'american' 'american' 'american'
 'american' 'italian' 'american' 'american' 'british' 'american'
 'american' 'american' 'american' 'american' 'american']
```

In [ ]:

Task 3.6: Evaluate model predictions

How well did we do? To quantify, for each model's output, we'll compute a *macro-averaged* F1 score, which we'll refer to as `lr_f1`, `rand_f1`, and `mf_f1`:

In [41]:
```
# YOUR CODE HERE

lr_f1 = f1_score(y_dev, lr_dev_preds, average='macro')

uniform_f1 = f1_score(y_dev, uniform_dev_preds, average='macro')

stratified_f1 = f1_score(y_dev, stratified_dev_preds, average='macro')

#raise NotImplementedError()
```

In [42]:
```
print('logistic regression f1:' , lr_f1)
print('uniform random guessing f1:' , uniform_f1)
print('stratified random guessing f1:' , stratified_f1)
```

```
logistic regression f1: 0.7617058014979912
uniform random guessing f1: 0.04107889723595883
stratified random guessing f1: 0.0516538950424169
```

Macro-averaging means that rather than simply computing a single F1 score across all of the model predictions, we'll separately compute F1 scores *within each label*, and then take the average F1 score over labels. That means that if our model is bad at predicting some of the labels, then the F1 score will be lower, even if the overall performance is decent. That's useful to know in very "imbalanced" settings like ours -- where some labels are much less common than others, but we're interested in good performance even for the rarer labels.

Task 3.6.1: Compare macro- with micro-averaging

You can see this for yourself by computing F1 scores across all the model predictions without aggregating by label. Compute the *micro-averaged* F1 scores, which we'll refer to as `lr_f1_micro`, `rand_f1_micro` and `mf_f1_micro`.

In [43]:
```
# YOUR CODE HERE
lr_f1_micro = f1_score(y_dev, lr_dev_preds, average='micro')

uniform_f1_micro = f1_score(y_dev, uniform_dev_preds, average='micro')

stratified_f1_micro = f1_score(y_dev, stratified_dev_preds, average='micro')

#raise NotImplementedError()
```

In [44]:
```
print('microaveraged logistic regression f1:' , lr_f1_micro)
print('microaveraged uniform random guessing f1:' , uniform_f1_micro)
print('microaveraged stratified random guessing f1:' , stratified_f1_micro)
```

```
microaveraged logistic regression f1: 0.8178316965145388
microaveraged uniform random guessing f1: 0.052378201424995184
microaveraged stratified random guessing f1: 0.18024263431542462
```

You should have gotten higher numbers across the board. For the sake of comparing our logistic regression to the baseline, micro-averaging doesn't lose a lot of information. But especially for one of the baselines (which, and why?) you might note that the micro-averaged F1 is deceptively high.

Part 4: Examine classifier performance

It looks like, at least relative to the baselines, our tf-idf + logistic regression model does pretty well! Now we'll examine how the model uses the input features to come up with output labels, and dig a little deeper into how well we're doing. Each of the following tasks comprises a useful way of making sense of what your model is doing.

Task 4.1: Examine model weights for each feature

The feature weights learned by our logistic regression model roughly correspond to how informative each feature (i.e., word) is of each class (i.e., nationality label). A higher feature weight means that this feature is more informative of this class -- i.e., roughly speaking, this word provides a strong hint to the classifier that a biography containing the world has a particular nationality label.

(Roughly in the sense that it takes some extra work to interpret them in a statistically rigorous way -- for instance, we don't mean "informative" in an information-theoretic sense [without a few more qualifications that are outside the scope of this class]. However, as we'll see, for the purposes of a cursory inspection this will do just fine.)

Assign the following variables the following values:

- `coefs` : a matrix consisting of the feature weights as learned by the classifier. (short for *coefficients*)
- `features` : an array consisting of the feature names -- i.e., the words considered, corresponding to each column of `coefs`
- `classes` : an array consisting of the classes -- i.e., the nationality labels, corresponding to each row of `coefs`

Note that `features` and `classes` need to occur in a particular order, which the classifier has learned (this is what is meant by "corresponding to").

Hint: you can figure out where to find these values by looking at the documentation for `TfidfVectorizer` and `LogisticRegression` . Also, we've already printed out each of these variables earlier on in the notebook.

In [45]:
```
coefs = None
features = None
classes = None

# YOUR CODE HERE

coefs = clf.coef_

features = vectorizer.get_feature_names_out()

classes = clf.classes_

#raise NotImplementedError()
```

We'll store the model weights, with corresponding features and class names, in a dataframe:

In [46]:
```
coef_df = pd.DataFrame(data=coefs.T, index=features, columns=classes)
display(coef_df)
```

	american	australian	british	canadian	dutch	english	french	german	indian	irish	italian	japanese	polish	russian	scottish	spanish	swed
000	0.518319	-0.491987	-0.298559	0.508226	0.830811	0.484417	-0.123374	0.024988	-0.488017	-0.294451	-0.058392	0.428152	-0.034472	0.028960	0.428169	-0.313122	0.085
08	0.489836	0.142858	-0.293319	-0.020717	0.072312	0.080704	0.013214	-0.152921	-0.128715	-0.102207	0.334434	-0.137517	0.258942	-0.082906	-0.119552	-0.242120	-0.061
10	0.017632	0.011959	-0.436826	0.571091	-0.292841	0.165264	0.367481	0.386875	0.220127	0.450940	0.093750	0.056321	-0.219795	0.016526	-0.300937	0.346852	-0.540
100	0.175876	-0.046997	-0.108253	0.086444	-0.021516	-0.283867	-0.293748	-0.091089	0.131450	-0.352609	0.130449	0.715888	0.379453	-0.568853	0.044731	-0.289617	0.073
10th	-0.567833	-0.349522	-0.231236	-0.058474	-0.089719	-0.051206	0.058599	-0.209089	0.250353	0.254890	0.318672	1.029486	-0.054907	0.142472	0.107138	0.083566	-0.123
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
yorkshire	-1.861420	-0.431546	3.290354	-0.405314	-0.170139	3.330026	-0.356180	-0.401695	-0.428060	-0.329052	-0.391586	-0.293259	-0.061430	-0.147890	-0.364892	-0.179698	-0.142
young	0.581852	0.475812	0.256559	-0.079970	0.046781	0.262733	0.053633	-0.820662	-0.116473	0.119176	0.110670	0.529735	-0.312783	-0.250234	-0.326176	-0.362705	0.253
younger	-1.086137	0.370329	0.642826	-0.450749	0.775437	-0.259633	-0.130427	-0.550578	0.515963	-0.029117	0.532215	0.283838	0.030269	-0.251419	-0.275705	-0.170172	0.183
youngest	0.869438	0.424300	-0.386257	-0.214906	-0.065204	-0.278804	0.067888	-0.143955	0.028181	0.083298	-0.237813	-0.118626	0.024020	-0.134896	0.620873	-0.291165	0.188
youth	-0.954717	-0.297275	-0.339455	0.021296	0.064419	0.178303	-0.056971	0.371497	0.098710	-0.192885	0.142567	-0.199291	0.284040	0.269213	0.117784	0.176285	0.074

2760 rows × 19 columns

As an example, the following code will print out the words that are most informative of the classes 'american' and 'canadian', respectively, along with their feature weights for those classes.

In [47]:
```
print('most predictive words for american:')
print(coef_df['american'].sort_values(ascending=False).head(10))
print()
print('most predictive words for canadian:')
print(coef_df['canadian'].sort_values(ascending=False).head(10))
```

```
most predictive words for american:
american         10.393521
york              5.606974
california        5.545281
basketball        5.373368
massachusetts     4.296384
washington        4.116208
boston            3.992902
carolina          3.982723
virginia          3.905295
pennsylvania      3.821710
Name: american, dtype: float64

most predictive words for canadian:
canadian      14.664749
canada        11.756756
toronto        8.559270
ontario        8.488920
montreal       7.300539
columbia       5.348878
premier        2.602114
edward         2.432212
prince         2.136724
lieutenant     2.053234
Name: canadian, dtype: float64
```

Task 4.1.1: Find most informative features

Set `top_french_feats` to be a list containing the five words that are most informative of the "french" nationality label, per the logistic regression coefficients.

In [48]:
```
top_french_feats = []
# YOUR CODE HERE
top_french_feats = coef_df['french'].sort_values(ascending=False).head(5).index.tolist()

#raise NotImplementedError()
```

In [49]:
```
print(top_french_feats)
```

```
['french', 'paris', 'france', 'jean', 'des']
```

In [ ]:
```

```

We can also look at what exactly the model predicts, and hence start to get a better sense of where it does well or poorly. In particular, we'll inspect the model's predictions on the dev data, and compare them with the actual labels.

In [50]:
```
pred_df = pd.DataFrame({'pred': lr_dev_preds, 'label': y_dev})
display(pred_df.sample(5, random_state=RANDOM_SEED))
```

	pred	label
459	british	english
1524	british	british
1844	american	american
1621	american	canadian
3020	american	american

Task 4.2: Examine model performance per label

One strategy for examining model predictions, especially for multi-class tasks, is to examine its performance per class. Here, we'll compute *accuracy* scores within each nationality label.

Create two dictionaries whose keys are the nationality labels:

- `class_size` stores the number of items in the dev data with each nationality label.
- `class_accuracy` stores the accuracy of predictions per class. i.e., `class_accuracy['american']` contains the percent of items in the dev data which have the label 'american', and are also predicted to be 'american' by the classifier.

(technically, setting each variable to be pandas series whose indexes are the nationality labels will also work, so do that if it's more straightforward to you.)

In [51]:
```
# here's some code that might help
pred_df['correct'] = pred_df.label == pred_df.pred

# YOUR CODE HERE
class_size = {}
class_accuracy = {}

class_size = pred_df['label'].value_counts()

correct_predictions_per_class = pred_df[pred_df['correct']].groupby('label').size()

class_accuracy = (correct_predictions_per_class / class_size).fillna(0)

#raise NotImplementedError()
```

For convenience we'll put these values in a table, and then inspect each label and its respective accuracy.

In [52]:
```
pred_stats = pd.DataFrame({'class_accuracy': class_accuracy,
              'class_size': class_size})
display(pred_stats.sort_values('class_accuracy', ascending=False))
```

	class_accuracy	class_size
american	0.974093	1930
polish	0.906542	107
australian	0.896714	213
indian	0.880682	176
british	0.854756	778
french	0.847222	216
german	0.834197	193
canadian	0.789286	280
japanese	0.789062	128
dutch	0.781609	87
italian	0.779310	145
russian	0.731707	82
spanish	0.728395	81
swedish	0.725490	51
irish	0.701493	134
usa	0.504587	109
scottish	0.500000	82
english	0.279817	218
united states	0.027322	183

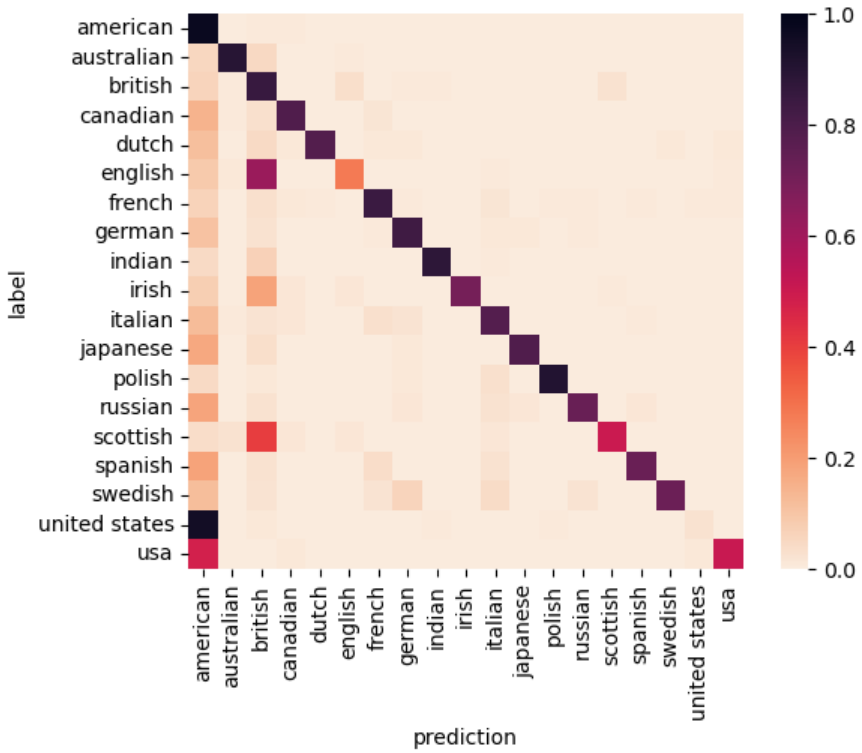
What do you notice about which labels the model does better on?

Another strategy for examining model performance is to look at common "confusions": where the true label is one thing, but the model consistently outputs another label. scikit-learn has a function, `confusion_matrix`, that will compute, for each label, the proportion of times that the model predicts a different label.

In [53]:
```
conf_df = pd.DataFrame(data=confusion_matrix(y_dev, lr_dev_preds, labels=classes, normalize='true'),
                       index=classes, columns=classes)
```

A good way to get a handle on model confusions is to plot a heatmap of the confusion matrix. Here, darker values correspond to more frequent (label, prediction) pairs.

In [54]:
```
ax = sns.heatmap(conf_df, square=True, cmap = sns.cm.rocket_r, vmin=0, vmax=1)
ax.set_ylabel('label')
ax.set_xlabel('prediction')
None
```



How to read this?

1. look at the (upper left to lower right) diagonal. These squares visualize how often your model outputs the *correct* prediction per label. Hopefully the diagonal is way darker than the other squares, i.e., your model is generally right.
2. look at the off-diagonal values -- i.e., what the model gets wrong. Which of those squares are particularly dark?

Now, some questions for you to think about:

- which of the on-diagonal squares are particularly light? meaning, the model did *not* generally output the correct prediction for that label.
- which of the off-diagonal squares are particularly dark? meaning, the model confused one nationality label for another.
- what do these errors tell us? could our model do a better job of capturing the distinctions between biographies of different nationality labels? could we have done a better job of cleaning the data? is there real-world nuance in who counts as what nationality, that we should think more carefully about?

Task 4.2.1: Examining the confusion matrix

Set `top_preds_for_canadian` to be a list containing the five predictions most frequently outputted by the model, for dev data items whose label is 'canadian'.

In [55]:
```
top_preds_for_canadian = None

# YOUR CODE HERE

canadian_pred_df = pred_df[pred_df['label'] == 'canadian']
top_preds_for_canadian_counts = canadian_pred_df['pred'].value_counts()

top_preds_for_canadian = top_preds_for_canadian_counts.head(5).index.tolist()
#raise NotImplementedError()
```

In [56]:
```
print(top_preds_for_canadian)
```

```
['canadian', 'american', 'british', 'french', 'australian']
```

Task 4.2.2: Diagnosing some confusions

As you may have gleaned from `top_preds_for_canadian`, one reason the model might produce incorrect predictions comes from properties of the problem domain: articles for people of certain nationalities might be too linguistically similar to each other to be distinguishable, or, people might inherently move around different countries, in which case the single nationality label is too coarse-grained to account for their biography.

Another reason why the model might be outputting incorrect predictions comes out of data cleanliness. Recall that earlier in the assignment, we processed out some redundancy among the labels. But our heuristics were not perfect.

Set `most_accurate_class` to be the nationality with the highest within-class accuracy. Then set `confused_classes` to be a *two-item* list (in any order) containing two labels that were in the training data, where data points with those labels were likely to be confused for `most_accurate_class` because they were redundant.

In [57]:
```
# YOUR CODE HERE
most_accurate_class = 'american'

confused_classes = ['united states', 'usa']


#raise NotImplementedError()
```

In [58]:
```
print(most_accurate_class)
print('redundant labels:', confused_classes)
```

```
american
redundant labels: ['united states', 'usa']
```

Task 4.3: Inspect data points with incorrect predictions

Finally, we'll look at particular biographies in the dev data where the model made an error. Note that the model makes *lots* of errors -- much more than we've got the time to read -- so we need to be strategic about picking erroneously-predicted items that will give us insights.

A nice feature of logistic regression classifiers is that in the course of outputting a predicted label, the model will also estimate the *probability* that an item is of a particular label (so in that sense, it is sort of like a regressor...). The prediction the model ultimately outputs corresponds to the class with the highest estimated probability. In scikit-learn's `LogisticRegression` class, we can get these probabilities via the `predict_proba` method.

The below code will set the 'prob' column of `pred_df` to be the estimated probability that each dev set item is of the label that the model predicted. So,

- higher values mean that the model is more sure that the item is of that label
- lower values mean the model is less sure

We colloquially say that the model has higher or lower "confidence" in its prediction, respectively.

In [59]:
```
probs = clf.predict_proba(X_dev).max(axis=1)
pred_df['prob'] = probs
```

Write code to get the dev data items where the model makes an incorrect prediction, and where the model is particularly confident or particularly unconfident about its output.

In particular, construct two dataframes which are *subsets* of `pred_df`:

- `most_confident_incorrect` is a dataframe of the 10 most confident incorrect predictions, ordered from more to less confident;
- `least_confident_incorrect` is a dataframe of the 10 least confident incorrect predictions, ordered from less to more confident.

In [60]:
```
# for convenience, we'll add the biography text to pred_df.
pred_df['bio'] = dev_df.bio.values

incorrect_preds = pred_df[pred_df['label'] != pred_df['pred']]
most_confident_incorrect = incorrect_preds.sort_values(by='prob', ascending=False).head(10)

least_confident_incorrect = incorrect_preds.sort_values(by='prob', ascending=True).head(10)
# YOUR CODE HERE
#raise NotImplementedError()
```

As you can (hopefully) see, the probability estimates are quite different for these two subsets:

In [61]:
```
print('most confident incorrect predictions')
display(most_confident_incorrect)
print()
print('least confident incorrect predictions')
display(least_confident_incorrect)
```

```
most confident incorrect predictions
```

	pred	label	correct	prob	bio
5100	italian	french	False	0.999862	Early life\nMilza was a second-generation Ital...
654	swedish	dutch	False	0.999020	Biography\nDe Geer was born in Liège, Belgium...
4179	irish	canadian	False	0.998145	Career\nNí Chofaigh is a journalist, presenter...
4049	canadian	british	False	0.993532	John Roger Spottiswoode (born 5 January 1945) ...
3692	canadian	american	False	0.992715	Early life\nBeard was born in Scarborough, Ont...
4266	canadian	irish	False	0.989996	Early years\nBorn in Mooncoin, Co. Kilkenny, I...
3369	american	french	False	0.987551	Early life and college years\nOlivier Saint-Je...
1161	american	united states	False	0.985592	Biography\nHughes joined the Army after high s...
127	canadian	italian	False	0.984926	Early life\nCaruana was born into the mob in C...
2197	polish	american	False	0.983407	Life and career\nSchally was born in Wilno, Se...

```
least confident incorrect predictions
```

	pred	label	correct	prob	bio
4480	german	italian	False	0.114869	Life\nBorn in Milano, Merz started drawing dur...
754	united states	french	False	0.143374	Last years and death\nIn the meantime, he had ...
3045	british	japanese	False	0.147103	Biography\nŌta Sukemoto was the third son of H...
1806	american	polish	False	0.147210	Personal bests\nOutdoor\n* 100 metres – 10.76 ...
3057	american	german	False	0.151718	Stefan Lucks is a researcher in the fields of ...
2482	spanish	british	False	0.153726	Career\n===Early career===\nDavies started rac...
3704	british	french	False	0.161031	Biography\nGiscard d'Estaing studied at the P...
1542	american	dutch	False	0.168582	Early life and career\nBorn Georg Henri Anton ...
1832	british	english	False	0.176893	Life and career\nNina was born in Brasília, Di...
755	french	spanish	False	0.177196	Career\nLeón travelled to Madrid to start a ca...

Inspect the most confident incorrect predictions. You can modify the below code to toggle how much of the biography is printed out, if that helps.

Can you spot signals in the text of the biographies that misled the model? Or, are you also confused by what the actual label in the dataset is? (it may be informative to Google some of these people and their biographies.)

In [62]:
```
for _, row in most_confident_incorrect.iterrows():
    print('actual: %s; predicted: %s; probability estimate: %.3f'
         % (row.label, row.pred, row.prob))
    print('bio:')
    print(row.bio[:500])
    print()
```

```
actual: french; predicted: italian; probability estimate: 1.000
bio:
Early life
Milza was a second-generation Italian immigrant, born in Paris, France to Italian parents. His father, Olivier Milza, was born near Parma in Italy. Hi
s status as an immigrant motivated his studies in Italian history and Italian-French immigration. Milza first visited Italy at 16, learned Italian and
began to study history.


actual: dutch; predicted: swedish; probability estimate: 0.999
bio:
Biography
De Geer was born in  Liège, Belgium. He was the son of the iron industrialist and merchant Louis de Geer de Gaillarmont (1535-1602), and Jeanne de Nei
lle (1557-1641). His father had previously (1563) been married to Maria de Jalhéa (died 1578). In 1592, one of De Geer's half-sisters, Marie de Geer
(1574-1609) married  (1569-1636), a Dutch merchant and a director of the Dutch East India Company who lived in Dordrecht. Presumably due to ongoing tu
rmoil in the Prince-Bishopric of Liège as

actual: canadian; predicted: irish; probability estimate: 0.998
bio:
Career
Ní Chofaigh is a journalist, presenter and broadcaster on Irish television. Former presenter of ''Sin É'', and former reporter for RTÉ News and Curren
t Affairs, she covered stories as diverse as the Clinton visit; 2004 Summer Olympics in Athens, 2003 Special Olympics World Summer Games; tall ships a
nd female genital mutilation.

Ní Chofaigh has presented shows such as ''The Afternoon Show'', ''Echo Island'' and The RTÉ People in Need Telethon on RTÉ Television. Her first role
as a TV pres

actual: british; predicted: canadian; probability estimate: 0.994
bio:
John Roger Spottiswoode (born 5 January 1945) is a Canadian-British director, editor and writer of film and television. He was born in Ottawa, Ontari
o, Canada, and was raised in Britain. His father Raymond Spottiswoode was a British film theoretician who worked at the National Film Board of Canada
during the 1940s, directing such short films such as ''Wings of a Continent''.

actual: american; predicted: canadian; probability estimate: 0.993
bio:
Early life
Beard was born in Scarborough, Ontario, Canada. Her Suga BayBee was also a character on the ''Mad Dog and Billie'' morning show formerly aired on Toro
nto radio station KISS 92.5. Beard's squeaky-voiced character began appearing on KISS when Beard was 19 years old and otherwise working as a sales cle
rk.
Career
Her voice in 2000 was most known as the voice Sailor Mini Moon in the Cloverway adaptation of ''Sailor Moon''.
From 2001 to 2007, Beard hosted and produced ''The Zone'', a series

actual: irish; predicted: canadian; probability estimate: 0.990
bio:
Early years
Born in Mooncoin, Co. Kilkenny, Ireland, Walsh and came to Canada in 1852 to complete his studies at the Saint-Sulpice Seminary in Montreal, Lower Can
ada. He was ordained a priest of Toronto, Upper Canada in 1854 by then Bishop of Toronto Armand-François-Marie de Charbonnel, at St. Michael's Cathedr
al. On his way to Toronto, in the summer of 1854, Walsh came down cholera, which permanently undermined his health.

Father Walsh served as pastor at Brock. In April 1857 he was made paris

actual: french; predicted: american; probability estimate: 0.988
bio:
Early life and college years
Olivier Saint-Jean was born in Maisons-Alfort near Paris from parents who were natives of French Guiana. His mother George Goudet was a professional b
asketball player.  After graduating from Lycee Aristide Briand in 1993, Abdul-Wahad first played college basketball for two years at Michigan and tran
sferred to San Jose State in 1995. Abdul-Wahad was part of the San Jose State team that won the 1996 Big West Conference Men's Basketball Tournament a
```

```
nd made the NCAA tour

actual: united states; predicted: american; probability estimate: 0.986
bio:
Biography
Hughes joined the Army after high school. At 6'6" he was recruited for a special unit that just played basketball; it was the first integrated team Hu
ghes ever played on. When he left the Army, Texas Southern University offered him a basketball scholarship. Playing for coach Edward H. Adams, Hughes
was an All-American at Southern. In 1955 Hughes met his wife, Jacquelyne Johnson, while playing in a tournament in Memphis, then was drafted by the Bo
ston Celtics, but he did not make the te

actual: italian; predicted: canadian; probability estimate: 0.985
bio:
Early life
Caruana was born into the mob in Castelvetrano in the province of Trapani, Sicily. His family originated from Siculiana in the province of Agrigento,
and is closely related to the Cuntrera-family through several marriages. Alfonso Caruana married his niece Giuseppina Caruana and has a son and two da
ughters.
Mafia career
In 1968, Alfonso Caruana arrived in Montreal, Quebec, Canada, with $100 in his pocket pretending to be an electrician. Ten years later he was stopped
at the internatio

actual: american; predicted: polish; probability estimate: 0.983
bio:
Life and career
Schally was born in Wilno, Second Polish Republic (since 1945 Vilnius, Lithuania), as the son of Gen. Brigadier Kazimierz Schally, who was Chief of th
e Cabinet of President Ignacy Mościcki of Poland, and Maria (Łącka).

In September 1939, when Poland was attacked by Nazi Germany and the Soviet Union, Schally escaped with the Poland's President Ignacy Mościcki, Prime M
inister and the whole cabinet to the neutral Romania, where they were interned.
I was fortunate to survive the hol
```

Now inspect the least confident errors. These are often cases where the model did not have enough information to go off of, to make a prediction. For instance, maybe the words used in the biography were relatively obscure, and did not register as features when you constructed the `TfidfVectorizer` earlier on. Or, maybe the various words conflict in which label they signal.

In [63]:
```
for _, row in least_confident_incorrect.iterrows():
    print('actual: %s; predicted: %s; probability estimate: %.3f'
         % (row.label, row.pred, row.prob))
    print('bio:')
    print(row.bio[:500])
    print()
```

```
actual: italian; predicted: german; probability estimate: 0.115
bio:
Life
Born in Milano, Merz started drawing during World War II, when he was imprisoned for his activities with the ''Giustizia e Libertà'' antifascist grou
p. He experimented with a continuous graphic stroke–not removing his pencil point from the paper. He explored the relationship between nature and the
subject, until he had his first exhibitions in the intellectually incendiary context of Turin in the 1950s, a cultural climate fed by such writers as
Cesare Pavese, Elio Vittorini, and Ezra Pound.

actual: french; predicted: united states; probability estimate: 0.143
bio:
Last years and death
In the meantime, he had fired most of the League's staff, including all of the British employees. By the time when World War II started on September
1, 1939, Avenol decided to leave Geneva and the League of Nations for good on August 31, 1940. His services were not accepted by the Vichy government,
and he was forced to flee back into Switzerland on New Year's Eve 1943 to avoid getting arrested or killed by the Germans. He died at his home in Duil
lier, Switzerland in 1952, ag

actual: japanese; predicted: british; probability estimate: 0.147
bio:
Biography
Ōta Sukemoto was the third son of Hotta Masazane, ''daimyō'' of Omi-Miyagawa Domain. He was selected as posthumous heir on Ōta Suketoki's sudden death
in 1810 and married to one of Suketoki's daughters. At the time, he was only eleven years old. He was received in formal audience by ''Shōgun'' Tokuga
wa Ienari in 1812 and was appointed a ''sōshaban'' in 1818.

Sukemoto was appointed ''Jisha-bugyō'' on July 17, 1822, and ''Osaka jōdai'' on November 22, 1828, followed by the post of ''Kyo

actual: polish; predicted: american; probability estimate: 0.147
bio:
Personal bests
Outdoor
* 100 metres – 10.76 s (2012)
* 200 metres – 21.07 s (2011)
* 400 metres – 45.27 s (2011)

'''Indoor'''
* 400 metres – 46.64 (2006)


actual: german; predicted: american; probability estimate: 0.152
bio:
Stefan Lucks is a researcher in the fields of communications security and cryptography. Lucks is known for his attack on Triple DES, and for extending
Lars Knudsen's Square attack to Twofish, a cipher outside the Square family, thus generalising the attack  into integral cryptanalysis.  He has also c
o-authored attacks on AES, LEVIATHAN, and the E0 cipher used in Bluetooth devices, as well as publishing strong password-based key agreement schemes.

actual: british; predicted: spanish; probability estimate: 0.154
bio:
Career
===Early career===
Davies started racing in the British Mini Moto championship in 1995, moving up to Junior Road racing in 1999. In the same season he was given special
dispensation at the age of 12 to take part in the Aprilia Challenge 125cc Championship. Davies finished 6th overall, and was the only rider to finish
every race in a points scoring position. He set a new lap record at Donington Park National circuit, and was awarded 'Superteen of the Year'. He staye
d with the series for a

actual: french; predicted: british; probability estimate: 0.161
bio:
Biography
Giscard d'Estaing studied at the  Paris Institute of Political Studies and has a masters in economics.
```

```
He began his career with Cofremca where he served as associate director from 1982 to 1987, helping research changes in patterns of food consumption an
d its impacts on marketing and strategy.

In 1987 he joined the Danone Group and held various executive positions with subsidiaries such as HP Foods and Evian-Badoit.

Giscard d’Estaing joined the resort company Club Med in 1997 as

actual: dutch; predicted: american; probability estimate: 0.169
bio:
Early life and career
Born Georg Henri Anton Ivens into a wealthy family, Ivens went to work in one of his father's photo supply shops and from there developed an interest
in film. Under the direction of his father, he completed his first film at 13; in college he studied economics with the goal of continuing his fathe
r's business, but an interest in class issues distracted him from that path. He met photographer Germaine Krull in Berlin in 1923, and entered into a
marriage of convenience with h

actual: english; predicted: british; probability estimate: 0.177
bio:
Life and career
Nina was born in Brasília, Distrito Federal. Her mother is the English Liz Thompson-Miranda; Her father is the Brazilian artist Luiz Aquila, who is al
so a visual artist. The family subsequently lived in both England and France. Miranda said  that she wanted to be an opera singer when she was younge
r, but later studied art instead.

Smoke City played major venues and festivals in the UK and Europe.  Their first album, ''Flying Away'', was released in 1997, followed in 2001 by ''He

actual: spanish; predicted: french; probability estimate: 0.177
bio:
Career
León travelled to Madrid to start a career as an actress, where she met Pedro Almodóvar during La Movida Madrileña. She has played in ''Women on the V
erge of a Nervous Breakdown'' (1987), ''Tie Me Up! Tie Me Down!'' (1989), ''The Bilingual Lover'' (1993), and ''Libertarias'' (1996). In 1997, she had
her first leading role in ''Amor de hombre'', and played Paloma in ''Aquí no hay quien viva'' (2003). In 2016, she played Menchu in ''La que se avecin
a'' as Yoli's mother.
```

In [ ]:

In [ ]: