index

May 28, 2022

Link to this lab is here or

`https://github.com/miladshiraniUCB/P4-cumu-NLP-dsc-nlp-lab`

1 Text Classification - Cumulative Lab

1.1 Introduction

In this cumulative lab, we'll use everything we've learned so far to build a model that can classify a text document as one of many possible classes!

1.2 Objectives

You will be able to:

- Practice cleaning and exploring a text dataset with NLTK and base Python
- Practice using scikit-learn vectorizers for text preprocessing
- Tune a modeling process through exploration and model evaluation
- Observe some techniques for feature engineering
- Interpret the result of a final ML model that classifies text data

1.3 Your Task: Complete an End-to-End ML Process with the Newsgroups Dataset

1.3.1 Business Understanding

The ***Newsgroups Dataset*** is a collection of newsgroup posts originally collected around 1995. While the backend code implementation is fairly different, you can think of them as like the Reddit posts of 1995, where a "category" in this dataset is like a subreddit.

The task is to try to identify the category where a post was published, based on the text content of the post.

1.3.2 Data Understanding

Data Source Part of what you are practicing here is using the `sklearn.datasets` submodule, which you have seen before (e.g. the Iris Dataset, the Wine Dataset). You can see a full list of available dataset loaders here.

In this case we will be using the `fetch_20newsgroups` function (documentation here). An important thing to note is that because this is text data, scikit-learn actually downloads a set of documents to the computer you are using to complete this lab, rather than just loading data into memory in Python.

Features Prior to preprocessing, every row in the dataset only contains one feature: a string containing the full text of the newsgroup post. We will perform preprocessing to create additional features.

Target As you might have guessed based on the function name, there are 20 categories in the full dataset. Here is a list of all the possible classes:

This full dataset is quite large. To save us from extremely long runtimes, we'll work with only a subset of the classes. For this lab, we'll work with the following five:

- `'comp.windows.x'`
- `'rec.sport.hockey'`
- `'misc.forsale'`
- `'sci.crypt'`
- `'talk.politics.misc'`

1.3.3 Requirements

1. Load the Data Use pandas and `sklearn.datasets` to load the train and test data into appropriate data structures. Then get a sense of what is in this dataset by visually inspecting some samples.

2. Perform Data Cleaning and Exploratory Data Analysis with `nltk` Standardize the case of the data and use a tokenizer to convert the full posts into lists of individual words. Then compare the raw word frequency distributions of each category.

3. Build and Evaluate a Baseline Model with `TfidfVectorizer` and `MultinomialNB` Ultimately all data must be in numeric form in order to be able to fit a scikit-learn model. So we'll use a tool from `sklearn.feature_extraction.text` to convert all data into a vectorized format.

Initially we'll keep all of the default parameters for both the vectorizer and the model, in order to develop a baseline score.

4. Iteratively Perform and Evaluate Preprocessing and Feature Engineering Techniques Here you will investigate three techniques, to determine whether they should be part of our final modeling process:

1. Removing stopwords
2. Using custom tokens
3. Domain-specific feature engineering
4. Increasing `max_features`

5. Evaluate a Final Model on the Test Set Once you have chosen a final modeling process, fit it on the full training data and evaluate it on the test data.

1.4 1. Load the Data

In the cell below, create the variables `newsgroups_train` and `newsgroups_test` by calling the `fetch_20newsgroups` function twice.

For the train set, specify `subset="train"`. For the test set, specify `subset="test"`.

Additionally, pass in `remove=('headers', 'footers', 'quotes')` in both function calls, in order to automatically remove some metadata that can lead to overfitting.

Recall that we are loading only five categories, out of the full 20. So, pass in `categories=categories` both times.

[3]:
```python
# Replace None with appropriate code
from sklearn.datasets import fetch_20newsgroups

categories = [
    'comp.windows.x',
    'rec.sport.hockey',
    'misc.forsale',
    'sci.crypt',
    'talk.politics.misc'
]

newsgroups_train = fetch_20newsgroups(
    subset="train",
    remove=('headers', 'footers', 'quotes'),
    categories=categories
)

newsgroups_test = fetch_20newsgroups(
    subset="test",
    remove=('headers', 'footers', 'quotes'),
    categories=categories
)
```

Each of the returned objects is a dictionary-like `Bunch` (documentation here):

[4]:
```python
# Run this cell without changes
type(newsgroups_train)
```

[4]:
```
sklearn.utils.Bunch
```

The important thing to know is that the `.data` attribute will extract the feature values, and the `.target` attribute will extract the target values. So, for example, the train features (`X_train`) are located in `newsgroups_train.data`, whereas the train targets (`y_train`) are located in `newsgroups_train.target`.

In the cell below, create `X_train`, `X_test`, `y_train`, `y_test` based on `newsgroups_train` and `newsgroups_test`.

[5]:
```python
# Replace None with appropriate code
import pandas as pd
pd.set_option('max_colwidth', 400)
pd.set_option('use_mathjax', False)

# Extract values from Bunch objects
X_train = pd.DataFrame(newsgroups_train.data, columns=["text"])
X_test = pd.DataFrame(newsgroups_test.data, columns=["text"])
y_train = pd.Series(newsgroups_train.target, name="category")
y_test = pd.Series(newsgroups_test.target, name="category")
```

Double-check that your variables have the correct shape below:

[6]:
```python
# Run this cell without changes

# X_train and X_test both have 1 column (text)
assert X_train.shape[1] == X_test.shape[1] and X_train.shape[1] == 1

# y_train and y_test are 1-dimensional (target value only)
assert len(y_train.shape) == len(y_test.shape) and len(y_train.shape) == 1

# X_train and y_train have the same number of rows
assert X_train.shape[0] == y_train.shape[0] and X_train.shape[0] == 2838

# X_test and y_test have the same number of rows
assert X_test.shape[0] == y_test.shape[0] and X_test.shape[0] == 1890
```

And now let's look at some basic attributes of the dataset.

Distribution of Target We know that there are five categories represented. How many are there of each?

[7]:
```python
# Run this cell without changes

train_target_counts = pd.DataFrame(y_train.value_counts())
train_target_counts["label"] = [newsgroups_train.target_names[val]
                                for val in train_target_counts.index]
train_target_counts.columns = ["count", "target name"]
train_target_counts.index.name = "target value"
train_target_counts
```

[7]:
```
              count         target name
target value
2               600    rec.sport.hockey
3               595          sci.crypt
0               593      comp.windows.x
1               585        misc.forsale
4               465  talk.politics.misc
```

So, for example, the category “comp.windows.x” has the label of `0` in our dataset, and there are 593 text samples in that category within our training data.

We also note that our target distribution looks reasonably balanced. Now let’s look at the features.

Visually Inspecting Features Run the cell below to view some examples of the features:

```
[8]: # Run this cell without changes

     # Sample 5 records and display full text of each
     train_sample = X_train.sample(5, random_state=22)
     train_sample["label"] = [y_train[val] for val in train_sample.index]
     train_sample.style.set_properties(**{'text-align': 'left'})
```

```
[8]: <pandas.io.formats.style.Styler at 0x7fc73cc85c40>
```

In order, we have:

- An example of `comp.windows.x`, talking about “host loading considerations”
- An example of `talk.politics.misc`, talking about government and currency
- An example of `misc.forsale`, talking about a list of comics for sale
- An example of `rec.sport.hockey`, talking about hockey players and the Bruins
- An example of `sci.crypt`, talking about a microprocessor

We appear to have loaded the data correctly, so let’s move on and perform some cleaning and additional exploratory analysis.

1.5 2. Perform Data Cleaning and Exploratory Data Analysis with `nltk`

Prior to any exploratory analysis, we’ll complete two common data cleaning tasks for text data: standardizing case and tokenizing.

1.5.1 Standardizing Case

In an NLP modeling process, sometimes we will want to preserve the original case of words (i.e. to treat `"It"` and `"it"` as different words, and sometimes we will want to standardize case (i.e. to treat `"It"` and `"it"` as the same word).

To figure out what we want to do, let’s look at the first sample from above:

```
[9]: # Run this cell without changes
     windows_sample = train_sample.iloc[0]["text"]
     windows_sample
```

```
[9]: '\n\n\n   Ncd has an excellent document titled "Host Loading Considerations in
     the X \n   environment". I received my copy by emailing support@ncd.com. This
     may\n  help out.'
```

Here we have two references to the company Network Computing Devices, or NCD. At the beginning, the poster refers to it as `"Ncd"`. Then later refers to `"support@ncd.com"`. It seems reasonable to assume that both of these should be treated as references to the same word instead of treating

"Ncd" and "ncd" as two totally separate things. So let's standardize the case of all letters in this dataset.

The typical way to standardize case is to make everything lowercase. While it's possible to do this after tokenizing, it's easier and faster to do it first.

For a single sample, we can just use the built-in Python `.lower()` method:

[10]:
```
# Run this cell without changes
windows_sample.lower()
```

[10]:
```
'\n\n\n   ncd has an excellent document titled "host loading considerations in
the x \n  environment". i received my copy by emailing support@ncd.com. this
may\n  help out.'
```

Standarizing Case in the Full Dataset To access this method in pandas, you use `.str.lower()`:

[11]:
```
# Run this cell without changes

# Transform sample data to lowercase
train_sample["text"] = train_sample["text"].str.lower()
# Display full text
train_sample.style.set_properties(**{'text-align': 'left'})
```

[11]:
```
<pandas.io.formats.style.Styler at 0x7fc73cc85e80>
```

In the cell below, perform the same operation on the full `X_train`:

[12]:
```
# Replace None with appropriate code

# Transform text in X_train to lowercase
X_train["text"] = X_train["text"].str.lower()
```

Double-check your work by looking at an example and making sure the text is lowercase:

[13]:
```
# Run this cell without changes
X_train.iloc[100]["text"]
```

[13]:
```
"i have a problem where an athena strip chart widget is not calling it's\nget
value function.  i am pretty sure this is happening because i am\nnot using
xtappmainloop, but am dealing with events via sockets.  (ya ya).\n\nanyway, i
want to cause a timeout so that the strip chart widget(s) will\ncall their get
value callback.  or if someone knows another fast way around\nthis (or any way
for that matter) let me know.  i cannot (or i don't think)\ncall the xtngetvalue
callback myself because i don't have the value for\nthe third parameter of the
get value proc (xtpointer call_data).  \n\nin other words, i want to force a
strip chart widget to update itself.\n\nany ideas anyone?  \n"
```

1.5.2 Tokenizing

Now that the case is consistent it's time to convert each document from a single long string into a set of tokens.

Let's look more closely at the second example from our training data sample:

```
[14]: # Run this cell without changes
      politics_sample = train_sample.iloc[1]["text"]
      politics_sample
```

```
[14]: "\n\n \n           you don't have to.  *it*  believes in you.\n\n\n\n\n
      well, looking at our new government pals, i'm inclined to\n        agree.  i
      don't much believe in our money, either. :)\n\n\n\n\n    oh, ho ho!   if only
      you knew!  :)\n\n    yup, i'm definitely checking out foreign currency, thanks
      to\n    to this newsgroup.  it sure doesn't take much thinking to realize\n
      what direction the u.s. is headed.\n\n\n"
```

If we split this into tokens just by using the built-in Python `.split` string method, we would have a lot of punctuation attached:

```
[15]: # Run this cell without changes
      politics_sample.split()[:10]
```

```
[15]: ['you',
       "don't",
       'have',
       'to.',
       '*it*',
       'believes',
       'in',
       'you.',
       'well,',
       'looking']
```

(Punctuation being attached to words is a problem because we probably want to treat `you` and `you.` as two instances of the same token, not two different tokens.)

Let's use the default token pattern that scikit-learn uses in its vectorizers. The RegEx looks like this:

`(?u)\b\w\w+\b`

That means:

1. `(?u)`: use full unicode string matching
2. `\b`: find a word boundary (a word boundary has length 0, and represents the location between non-word characters and word characters)
3. `\w\w+`: find 2 or more word characters (all letters, numbers, and underscores are word characters)
4. `\b`: find another word boundary

In other words, we are looking for tokens that consist of two or more consecutive word characters, which include letters, numbers, and underscores.

We'll use the `RegexpTokenizer` from NLTK to create these tokens, initially just transforming the politics sample:

```
[16]: # Run this cell without changes

      from nltk.tokenize import RegexpTokenizer

      basic_token_pattern = r"(?u)\b\w\w+\b"

      tokenizer = RegexpTokenizer(basic_token_pattern)
      tokenizer.tokenize(politics_sample)[:10]
```

```
[16]: ['you', 'don', 'have', 'to', 'it', 'believes', 'in', 'you', 'well', 'looking']
```

Tokenizing the Full Dataset The way to tokenize all values in a column of a pandas dataframe is to use `.apply` and pass in `tokenizer.tokenize`.

For example, with the sample dataset:

```
[17]: # Run this cell without changes

      # Create new column with tokenized data
      train_sample["text_tokenized"] = train_sample["text"].apply(tokenizer.tokenize)
      # Display full text
      train_sample.style.set_properties(**{'text-align': 'left'})
```

```
[17]: <pandas.io.formats.style.Styler at 0x7fc73dd3e3a0>
```

In the cell below, apply the same operation on `X_train`:

```
[18]: # Replace None with appropriate code

      # Create column text_tokenized on X_train
      X_train["text_tokenized"] = X_train["text"].apply(tokenizer.tokenize)
```

Visually inspect your work below:

```
[19]: # Run this cell without changes
      X_train.iloc[100]["text_tokenized"][:20]
```

```
[19]: ['have',
       'problem',
       'where',
       'an',
       'athena',
       'strip',
       'chart',
       'widget',
       'is',
       'not',
       'calling',
       'it',
       'get',
       'value',
       'function',
       'am',
       'pretty',
       'sure',
       'this',
       'is']
```

(Note that we have removed all single-letter words, so instead of `"have", "a", "problem"`, the sample now shows just `"have", "problem"`. If we wanted to include single-letter words, we could use the token pattern `(?u)\b\w+\b` instead.)

Now that our data is cleaned up (case standardized and tokenized), we can perform some EDA.

1.5.3 Exploratory Data Analysis: Frequency Distributions

Recall that a frequency distribution is a data structure that contains pieces of data as well as the count of how frequently they appear. In this case, the pieces of data we'll be looking at are tokens (words).

In the past we have built a frequency distribution "by hand" using built-in Python data structures. Here we'll use another handy tool from NLTK called `FreqDist` (documentation here). `FreqDist` allows us to pass in a single list of words, and it produces a dictionary-like output of those words and their frequencies.

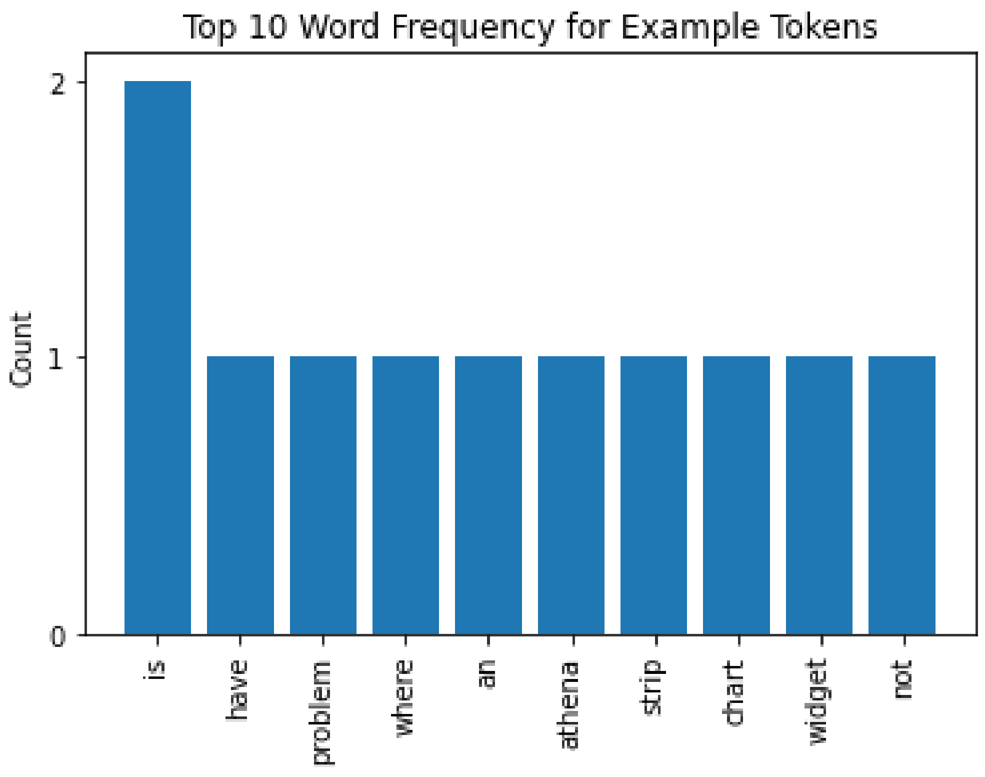
For example, this creates a frequency distribution of the example shown above:

[20]:
```
# Run this cell without changes
from nltk import FreqDist

example_freq_dist = FreqDist(X_train.iloc[100]["text_tokenized"][:20])
example_freq_dist
```

[20]:
```
FreqDist({'is': 2, 'have': 1, 'problem': 1, 'where': 1, 'an': 1, 'athena': 1,
'strip': 1, 'chart': 1, 'widget': 1, 'not': 1, …})
```

Then can use Matplotlib to visualize the most common words:

[21]:
```
# Run this cell without changes
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

def visualize_top_10(freq_dist, title):
    # Extract data for plotting
    top_10 = list(zip(*freq_dist.most_common(10)))
    tokens = top_10[0]
    counts = top_10[1]

    # Set up plot and plot data
    fig, ax = plt.subplots()
    ax.bar(tokens, counts)

    # Customize plot appearance
    ax.set_title(title)
    ax.set_ylabel("Count")
    ax.yaxis.set_major_locator(MaxNLocator(integer=True))
    ax.tick_params(axis="x", rotation=90)

visualize_top_10(example_freq_dist, "Top 10 Word Frequency for Example Tokens")
```



Interpreting the chart above is a bit artificial, since this sample only included 20 tokens. But essentially this is saying that the token with the highest frequency in our example is "is", which occurred twice.

Visualizing the Frequency Distribution for the Full Dataset Let's do that for the full X_train.

First, we need a list of all of the words in the text_tokenized column. We could do this manually by looping over the rows, but fortunately pandas has a handy method called .explode() (documentation here) that does exactly this.

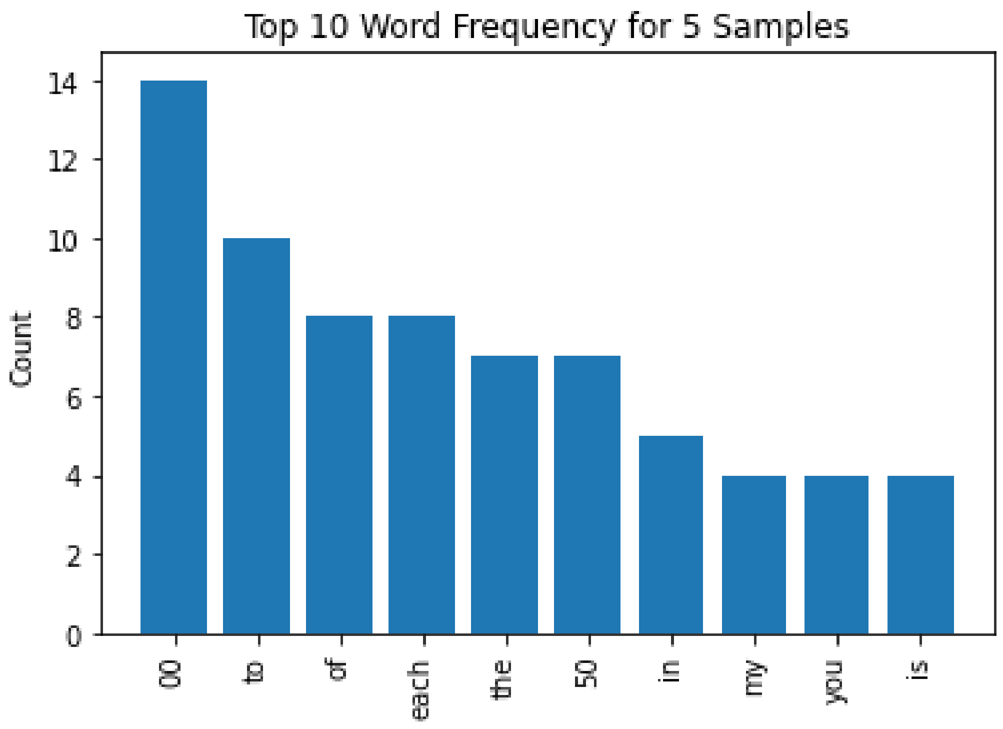
Here is an example applying that to the sample dataframe:

[22]:
```
# Run this cell without changes
train_sample["text_tokenized"].explode()
```

[22]:
```
1300            ncd
1300            has
1300             an
1300      excellent
1300       document
              …
1043   infringement
1043             on
1043          thier
1043           name
1043           sake
Name: text_tokenized, Length: 289, dtype: object
```

And we can visualize the top 10 words from the sample dataframe like this:

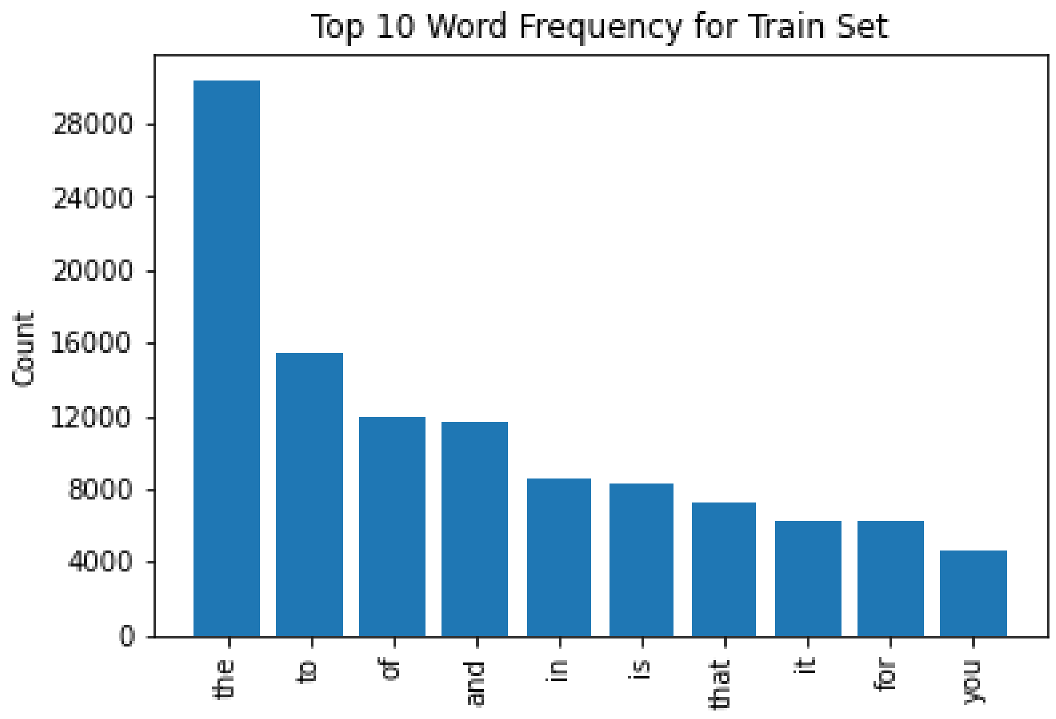
[23]:
```
# Run this cell without changes
sample_freq_dist = FreqDist(train_sample["text_tokenized"].explode())
visualize_top_10(sample_freq_dist, "Top 10 Word Frequency for 5 Samples")
```



Note that "00" and "50" are both in the top 10 tokens, due to many prices appearing in the `misc.forsale` example.

In the cell below, complete the same process for the full `X_train`:

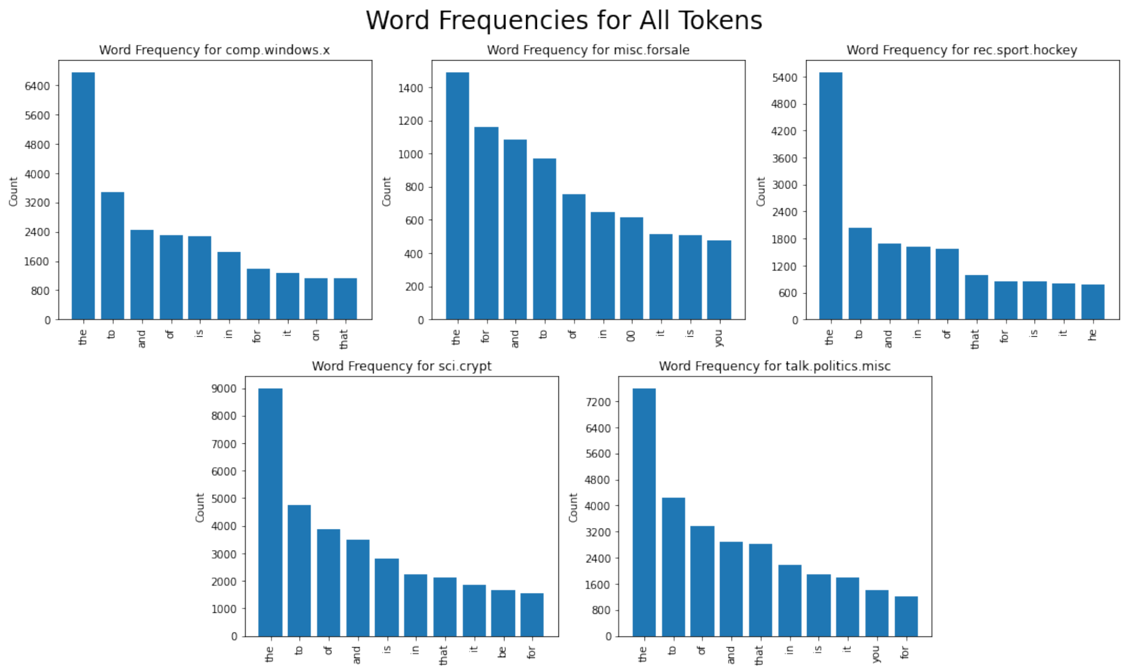
[24]:
```
# Replace None with appropriate code

# Create a frequency distribution for X_train
train_freq_dist = FreqDist(X_train["text_tokenized"].explode())

# Plot the top 10 tokens
visualize_top_10(train_freq_dist, "Top 10 Word Frequency for Train Set")
```



Ok great, we have a general sense of the word frequencies in our dataset!

We can also subdivide this by category, to see if it makes a difference:

```
[25]: # Run this cell without changes

# Add in labels for filtering (we won't pass them in to the model)
X_train["label"] = [y_train[val] for val in X_train.index]

def setup_five_subplots():
    """
    It's hard to make an odd number of graphs pretty with just nrows
    and ncols, so we make a custom grid. See example for more details:
    https://matplotlib.org/stable/gallery/subplots_axes_and_figures/
 ↪gridspec_multicolumn.html

    We want the graphs to look like this:
     [ ] [ ] [ ]
       [ ] [ ]

    So we make a 2x6 grid with 5 graphs arranged on it. 3 in the
    top row, 2 in the second row

      0 1 2 3 4 5
    0|[|]|[|]|[|]|
    1| |[|]|[|]| |
    """
    fig = plt.figure(figsize=(15,9))
    fig.set_tight_layout(True)
    gs = fig.add_gridspec(2, 6)
    ax1 = fig.add_subplot(gs[0, :2]) # row 0, cols 0-1
    ax2 = fig.add_subplot(gs[0, 2:4])# row 0, cols 2-3
    ax3 = fig.add_subplot(gs[0, 4:]) # row 0, cols 4-5
    ax4 = fig.add_subplot(gs[1, 1:3])# row 1, cols 1-2
    ax5 = fig.add_subplot(gs[1, 3:5])# row 1, cols 3-4
    return fig, [ax1, ax2, ax3, ax4, ax5]

def plot_distribution_of_column_by_category(column, axes, title="Word Frequency␣
 ↪for"):
    for index, category in enumerate(newsgroups_train.target_names):
        # Calculate frequency distribution for this subset
        all_words = X_train[X_train["label"] == index][column].explode()
        freq_dist = FreqDist(all_words)
        top_10 = list(zip(*freq_dist.most_common(10)))
        tokens = top_10[0]
        counts = top_10[1]

        # Set up plot
        ax = axes[index]
        ax.bar(tokens, counts)

        # Customize plot appearance
        ax.set_title(f"{title} {category}")
        ax.set_ylabel("Count")
        ax.yaxis.set_major_locator(MaxNLocator(integer=True))
        ax.tick_params(axis="x", rotation=90)


fig, axes = setup_five_subplots()
plot_distribution_of_column_by_category("text_tokenized", axes)
fig.suptitle("Word Frequencies for All Tokens", fontsize=24);
```



If these were unlabeled, would you be able to figure out which one matched with which category?

Well, `misc.forsale` still has a number (`"00"`) as one of its top tokens, so you might be able to figure out that one, but it seems very difficult to distinguish the others; every single category has `"the"` as the most common token, and every category except for `misc.forsale` has `"to"` as the second most common token.

After building our baseline model, we'll use this information to inform our next preprocessing steps.

1.6 3. Build and Evaluate a Baseline Model with `TfidfVectorizer` and `MultinomialNB`

Let's start modeling by building a model that basically only has access to the information in the plots above. So, using the default token pattern to split the full text into tokens, and using a limited vocabulary.

To give the model a little bit more information with those same features, we'll use a `TfidfVectorizer` (documentation here) so that it counts not only the term frequency (`tf`) within a single document, it also includes the inverse document frequency (`idf`) — how rare the term is.

In the cell below, import the vectorizer, instantiate a vectorizer object, and fit it on `X_train["text"]`.

[29]:
```
# Replace None with appropriate code

# Import the relevant vectorizer class
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
# Instantiate a vectorizer with max_features=10
# (we are using the default token pattern)
tfidf = TfidfVectorizer(max_features=10)

# Fit the vectorizer on X_train["text"] and transform it
X_train_vectorized = tfidf.fit_transform(X_train["text"])

# Visually inspect the 10 most common words
pd.DataFrame.sparse.from_spmatrix(X_train_vectorized, columns=tfidf.
 ↪get_feature_names())
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

[29]:
```
           and       for        in        is        it        of      that  \
0     0.322609  0.077590  0.304553  0.238740  0.203477  0.331334  0.290966
1     0.090518  0.097966  0.096133  0.100479  0.000000  0.092966  0.104965
2     0.173200  0.187451  0.367889  0.192259  0.196634  0.355768  0.401688
3     0.468758  0.000000  0.355598  0.520342  0.152052  0.206330  0.077654
4     0.000000  0.328237  0.322097  0.000000  0.000000  0.000000  0.000000
…          …         …         …         …         …         …         …
2833  0.135376  0.097676  0.095849  0.450819  0.256154  0.370765  0.418620
2834  0.296277  0.192393  0.755176  0.328880  0.000000  0.121716  0.274852
2835  0.489400  0.794502  0.000000  0.000000  0.277808  0.000000  0.000000
2836  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
2837  0.000000  0.105889  0.103908  0.325815  0.000000  0.502423  0.567271

           the        to       you
0     0.278467  0.334292  0.561259
1     0.937591  0.253249  0.000000
2     0.448504  0.484575  0.000000
3     0.462422  0.312257  0.000000
4     0.261785  0.848518  0.000000
…          …         …         …
2833  0.350558  0.505001  0.058880
2834  0.306886  0.110522  0.000000
2835  0.000000  0.228205  0.000000
2836  0.000000  0.000000  0.000000
2837  0.168903  0.091243  0.510644

[2838 rows x 10 columns]
```

Check the shape of your vectorized data:

[30]:
```python
# Run this cell without changes

# We should still have the same number of rows
assert X_train_vectorized.shape[0] == X_train.shape[0]

# The vectorized version should have 10 columns, since we set
# max_features=10
assert X_train_vectorized.shape[1] == 10
```

Now that we have preprocessed data, fit and evaluate a multinomial Naive Bayes classifier (documentation here) using `cross_val_score` (documentation here).

[31]:
```python
# Replace None with appropriate code

# Import relevant class and function
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import cross_val_score

# Instantiate a MultinomialNB classifier
baseline_model = MultinomialNB()

# Evaluate the classifier on X_train_vectorized and y_train
baseline_cv = cross_val_score(baseline_model, X_train_vectorized, y_train)
baseline_cv
```

[31]:
```
array([0.39964789, 0.41725352, 0.3943662 , 0.42151675, 0.37389771])
```

How well is this model performing? Well, recall the class balance:

[32]:
```python
# Run this cell without changes
y_train.value_counts(normalize=True)
```

[32]:
```
2    0.211416
3    0.209655
0    0.208950
1    0.206131
4    0.163848
Name: category, dtype: float64
```

If we guessed the plurality class every time (class `2`), we would expect about 21% accuracy. So when this model is getting 37-42% accuracy, that is a clear improvement over just guessing. But with an accuracy below 50%, we still expect the model to guess the wrong class the majority of the time. Let's see if we can improve that with more sophisticated preprocessing.

1.7 4. Iteratively Perform and Evaluate Preprocessing and Feature Engineering Techniques

Now that we have our baseline, the fun part begins. As you've seen throughout this section, preprocessing text data is a bit more challenging than working with more traditional data types because there's no clear-cut answer for exactly what sort of preprocessing we need to do. As we are preprocessing our text data, we need to make some decisions about things such as:

- Do we remove stop words or not?
- What should be counted as a token? Do we stem or lemmatize our text data, or leave the words as is? Do we want to include non-"words" in our tokens?
- Do we engineer other features, such as bigrams, or POS tags, or Mutual Information Scores?
- Do we use the entire vocabulary, or just limit the model to a subset of the most frequently used words? If so, how many?
- What sort of vectorization should we use in our model? Boolean Vectorization? Count Vectorization? TF-IDF? More advanced vectorization strategies such as Word2Vec?

In this lab, we will work through the first four of these.

1.7.1 Removing Stopwords

Let's begin with the first question: ***do we remove stopwords or not?*** In general we assume that stopwords do not contain useful information, but that is not always the case. Let's empirically investigate the top word frequencies of each category to see whether removing stopwords helps us to distinguish between the catogories.

As-is, recall that the raw word frequency distributions of 4 out of 5 categories look very similar. They start with `the` as the word with by far the highest frequency, then there is a downward slope of other common words, starting with `to`. The `misc.forsale` category looks a little different, but it still has `the` as the top token.

If we remove stopwords, how does this change the frequency distributions for each category?

Stopwords List Once again, NLTK has a useful tool for this task. You can just import a list of standard stopwords:

[33]:
```python
# Run this cell without changes
from nltk.corpus import stopwords

stopwords_list = stopwords.words('english')
stopwords_list[:20]
```

[33]:
```
['i',
 'me',
 'my',
 'myself',
 'we',
 'our',
 'ours',
 'ourselves',
 'you',
 "you're",
 "you've",
 "you'll",
 "you'd",
 'your',
 'yours',
 'yourself',
 'yourselves',
 'he',
 'him',
 'his']
```

We can customize that list as well.

Let's say that we want to keep the word `"for"` in our final vocabulary, since it appears disproportionately often in the `misc.forsale` category. The code below removes that from the stopwords:

[35]:
```python
# Run this cell without changes
print("Original list length:", len(stopwords_list))
stopwords_list.pop(stopwords_list.index("for"))
print("List length after removing 'for':", len(stopwords_list))
```

```
Original list length: 179
List length after removing 'for': 178
```

In the cell below, write a function `remove_stopwords` that takes in a list-like collection of strings (tokens) and returns only those that are not in the list of stopwords. (Use the `stopwords_list` in the global scope, so that we can later use `.apply` with this function.)

[36]:
```python
# Replace None with appropriate code
def remove_stopwords(token_list):
    """
    Given a list of tokens, return a list where the tokens
    that are also present in stopwords_list have been
    removed
    """
    return [item for item in token_list if item not in stopwords_list]
```

Test it out on one example:

[37]:
```python
# Run this cell without changes
tokens_example = X_train.iloc[100]["text_tokenized"]
print("Length with stopwords:", len(tokens_example))
assert len(tokens_example) == 110

tokens_example_without_stopwords = remove_stopwords(tokens_example)
print("Length without stopwords:", len(tokens_example_without_stopwords))
assert len(tokens_example_without_stopwords) == 65
```

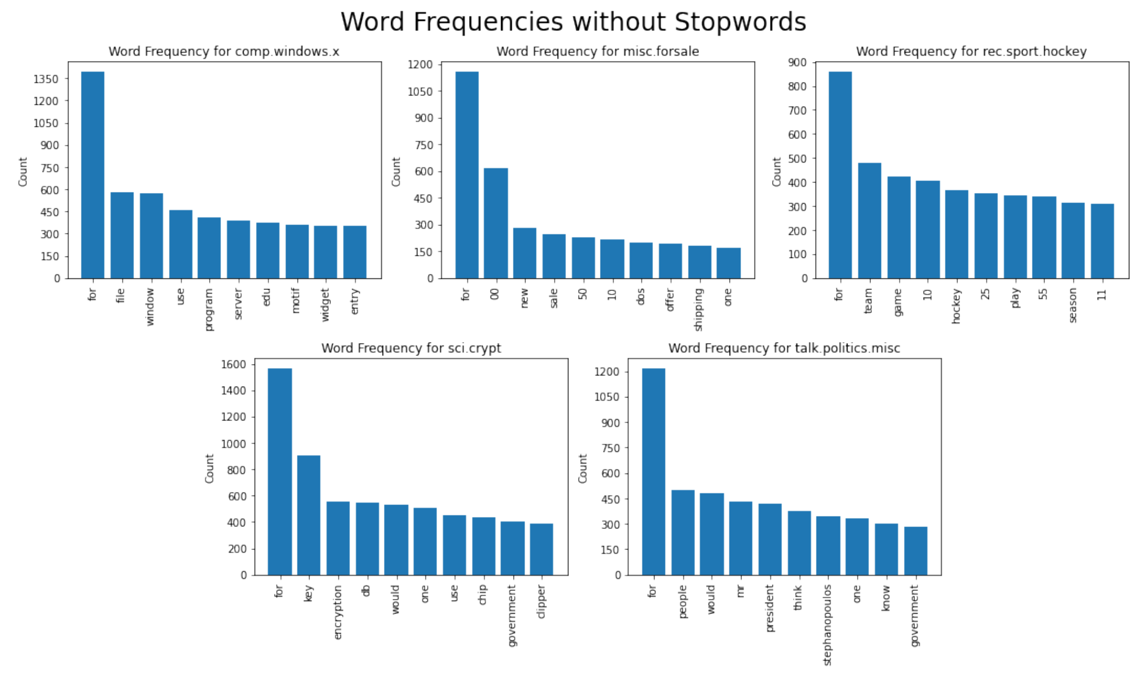
```
Length with stopwords: 110
Length without stopwords: 65
```

If that ran successfully, go ahead and apply it to the full `X_train`.

[38]:
```
# Run this cell without changes
X_train["text_without_stopwords"] = X_train["text_tokenized"].
 ↪apply(remove_stopwords)
```

Now we can compare frequency distributions without stopwords:

[39]:
```
# Run this cell without changes
fig, axes = setup_five_subplots()
plot_distribution_of_column_by_category("text_without_stopwords", axes)
fig.suptitle("Word Frequencies without Stopwords", fontsize=24);
```



Ok, this seems to answer our question. The most common words differ significantly between categories now, meaning that hopefully our model will have an easier time distinguishing between them.

Let's redo our modeling process, using `stopwords_list` when instantiating the vectorizer:

[40]:
```
# Run this cell without changes

# Instantiate the vectorizer
tfidf = TfidfVectorizer(
    max_features=10,
    stop_words=stopwords_list
)

# Fit the vectorizer on X_train["text"] and transform it
X_train_vectorized = tfidf.fit_transform(X_train["text"])

# Visually inspect the vectorized data
pd.DataFrame.sparse.from_spmatrix(X_train_vectorized, columns=tfidf.
 ↪get_feature_names())
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

[40]:
```
           edu       for       get  key      like       new       one  \
0     0.000000  0.599601  0.000000  0.0  0.000000  0.561043  0.000000
1     0.000000  0.502172  0.864768  0.0  0.000000  0.000000  0.000000
2     0.000000  1.000000  0.000000  0.0  0.000000  0.000000  0.000000
3     0.000000  0.000000  0.000000  0.0  0.000000  0.000000  0.000000
4     0.908329  0.418257  0.000000  0.0  0.000000  0.000000  0.000000
…          …         …         …  …         …         …         …
2833  0.000000  0.575451  0.495478  0.0  0.476688  0.000000  0.000000
2834  0.000000  0.867274  0.497831  0.0  0.000000  0.000000  0.000000
2835  0.000000  0.695901  0.000000  0.0  0.000000  0.000000  0.718138
2836  0.000000  0.000000  0.000000  0.0  0.000000  0.000000  0.000000
2837  0.000000  0.312260  0.537729  0.0  0.517336  0.000000  0.000000

        people       use     would
0     0.570709  0.000000  0.000000
1     0.000000  0.000000  0.000000
2     0.000000  0.000000  0.000000
3     0.525951  0.000000  0.850515
4     0.000000  0.000000  0.000000
…          …         …         …
2833  0.000000  0.000000  0.442862
2834  0.000000  0.000000  0.000000
2835  0.000000  0.000000  0.000000
2836  0.000000  0.000000  0.000000
2837  0.000000  0.587966  0.000000

[2838 rows x 10 columns]
```

[41]:
```
# Run this cell without changes

# Evaluate the classifier on X_train_vectorized and y_train
```

```
stopwords_removed_cv = cross_val_score(baseline_model, X_train_vectorized,␣
  ↪y_train)
stopwords_removed_cv
```

```
[41]: array([0.40669014, 0.42077465, 0.37676056, 0.45502646, 0.42857143])
```

How does this compare to our baseline?

```
[42]: # Run this cell without changes
print("Baseline:         ", baseline_cv.mean())
print("Stopwords removed:", stopwords_removed_cv.mean())
```

```
Baseline:          0.4013364135429863
Stopwords removed: 0.41756464714211183
```

Looks like we have a marginal improvement, but still an improvement. So, to answer ***do we remove stopwords or not:*** yes, let's remove stopwords.

1.7.2 Using Custom Tokens

Our next question is ***what should be counted as a token?***

Recall that currently we are using the default token pattern, which finds words of two or more characters. What happens if we also *stem* those words, so that `swims` and `swimming` would count as the same token?

Here we have provided a custom tokenizing function:

```
[43]: # Run this cell without changes
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer(language="english")

def stem_and_tokenize(document):
    tokens = tokenizer.tokenize(document)
    return [stemmer.stem(token) for token in tokens]
```

This uses `tokenizer` that we created earlier, as well as a new `stemmer` object. See an example below:

```
[44]: # Run this cell without changes
print("Original sample:", X_train.iloc[100]["text_tokenized"][20:30])
print("Stemmed sample: ", stem_and_tokenize(X_train.iloc[100]["text"])[20:30])
```

```
Original sample: ['happening', 'because', 'am', 'not', 'using', 'xtappmainloop',
'but', 'am', 'dealing', 'with']
Stemmed sample:  ['happen', 'becaus', 'am', 'not', 'use', 'xtappmainloop',
'but', 'am', 'deal', 'with']
```

We also need to stem our stopwords:

```
[45]: # Run this cell without changes
      stemmed_stopwords = [stemmer.stem(word) for word in stopwords_list]
```

In the cells below, repeat the modeling process from earlier. This time when instantiating the `TfidfVectorizer`, specify:

- `max_features=10` (same as previous)
- `stop_words=stemmed_stopwords` (modified)
- `tokenizer=stem_and_tokenize` (new)

```
[46]: # Replace None with appropriate code

      # Instantiate the vectorizer
      tfidf = TfidfVectorizer(max_features = 10,
                              stop_words = stemmed_stopwords,
                              tokenizer = stem_and_tokenize)

      # Fit the vectorizer on X_train["text"] and transform it
      X_train_vectorized = tfidf.fit_transform(X_train["text"])

      # Visually inspect the vectorized data
      pd.DataFrame.sparse.from_spmatrix(X_train_vectorized, columns=tfidf.
       ↪get_feature_names())
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

```
[46]:      file       for       get       key      like       new       one  \
      0     0.0  0.501934  0.000000  0.000000  0.400150  0.469658  0.000000
      1     0.0  0.524938  0.851140  0.000000  0.000000  0.000000  0.000000
      2     0.0  0.556285  0.000000  0.000000  0.000000  0.000000  0.000000
      3     0.0  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
      4     0.0  1.000000  0.000000  0.000000  0.000000  0.000000  0.000000
      …     …         …         …         …         …         …         …
      2833  0.0  0.588738  0.477293  0.000000  0.469351  0.000000  0.000000
      2834  0.0  0.879732  0.475469  0.000000  0.000000  0.000000  0.000000
      2835  0.0  0.700743  0.000000  0.000000  0.000000  0.000000  0.713414
      2836  0.0  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
      2837  0.0  0.180021  0.291888  0.386198  0.287031  0.000000  0.000000

               peopl       use     would
      0     0.476249  0.374901  0.000000
      1     0.000000  0.000000  0.000000
      2     0.000000  0.830992  0.000000
      3     0.524754  0.000000  0.851254
```

```
4     0.000000  0.000000  0.000000
…        …         …         …
2833  0.000000  0.000000  0.453088
2834  0.000000  0.000000  0.000000
2835  0.000000  0.000000  0.000000
2836  0.000000  0.000000  0.000000
2837  0.000000  0.806758  0.000000

[2838 rows x 10 columns]
```

[47]:
```
# Run this cell without changes

# Evaluate the classifier on X_train_vectorized and y_train
stemmed_cv = cross_val_score(baseline_model, X_train_vectorized, y_train)
stemmed_cv
```

[47]:
```
array([0.45246479, 0.44542254, 0.41373239, 0.50440917, 0.46737213])
```

How does this compare to our previous best modeling process?

[48]:
```
# Run this cell without changes
print("Stopwords removed:", stopwords_removed_cv.mean())
print("Stemmed:          ", stemmed_cv.mean())
```

```
Stopwords removed: 0.41756464714211183
Stemmed:           0.4566802046848995
```

Great! Another improvement, a slightly bigger one than we got when just removing stopwords. So, our best modeling process for now is one where we remove stopwords, use the default token pattern, and stem our tokens with a snowball stemmer.

1.7.3 Domain-Specific Feature Engineering

The way to really get the most information out of text data is by adding features beyond just vectorizing the tokens. This code will be completed for you, and it's okay if you don't fully understand everything that is happening, but we hope it helps you brainstorm for future projects!

Number of Sentences Does the number of sentences in a post differ by category? Let's investigate.

Once again, there is a tool from NLTK that helps with this task.

[49]:
```
# Run this cell without changes
from nltk.tokenize import sent_tokenize

sent_tokenize(X_train.iloc[100]["text"])
```

[49]:
```
["i have a problem where an athena strip chart widget is not calling it's\nget
value function.",
 'i am pretty sure this is happening because i am\nnot using xtappmainloop, but
am dealing with events via sockets.',
 '(ya ya).',
 'anyway, i want to cause a timeout so that the strip chart widget(s) will\ncall
their get value callback.',
 'or if someone knows another fast way around\nthis (or any way for that matter)
let me know.',
 "i cannot (or i don't think)\ncall the xtngetvalue callback myself because i
don't have the value for\nthe third parameter of the get value proc (xtpointer
call_data).",
 'in other words, i want to force a strip chart widget to update itself.',
 'any ideas anyone?']
```

We can just take the length of this list to find the number of sentences:

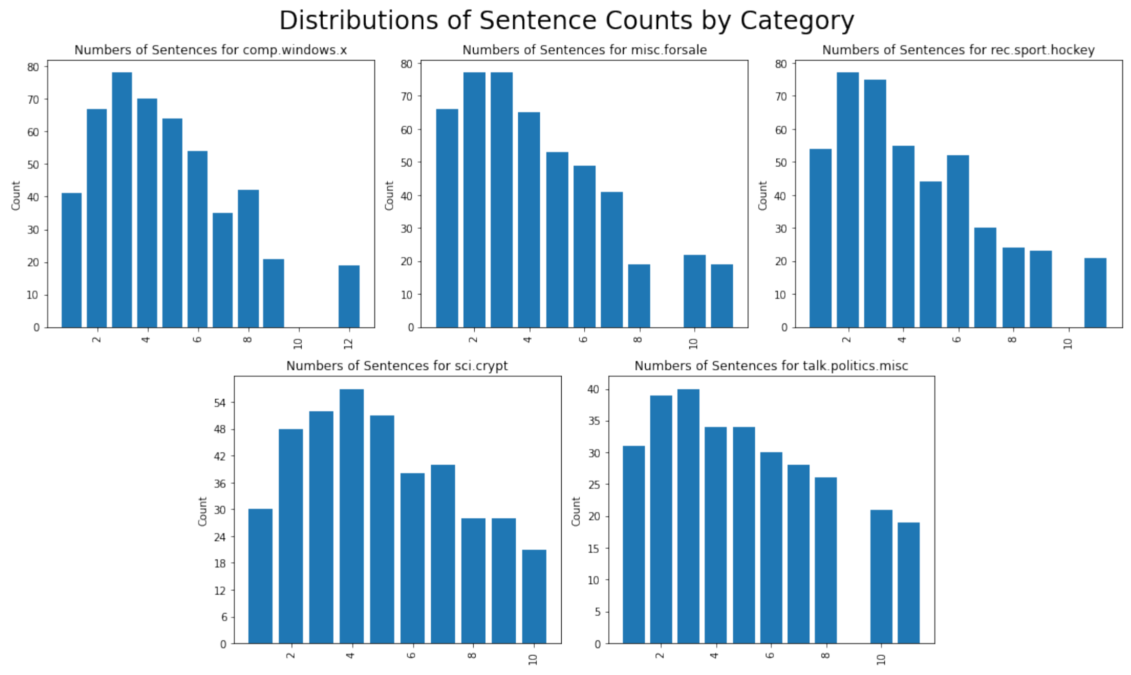
```
[50]: # Run this cell without changes
      len(sent_tokenize(X_train.iloc[100]["text"]))
```

```
[50]: 8
```

The following code adds a feature `num_sentences` to `X_train`:

```
[51]: # Run this cell without changes
      X_train["num_sentences"] = X_train["text"].apply(lambda x:␣
       ↪len(sent_tokenize(x)))
```

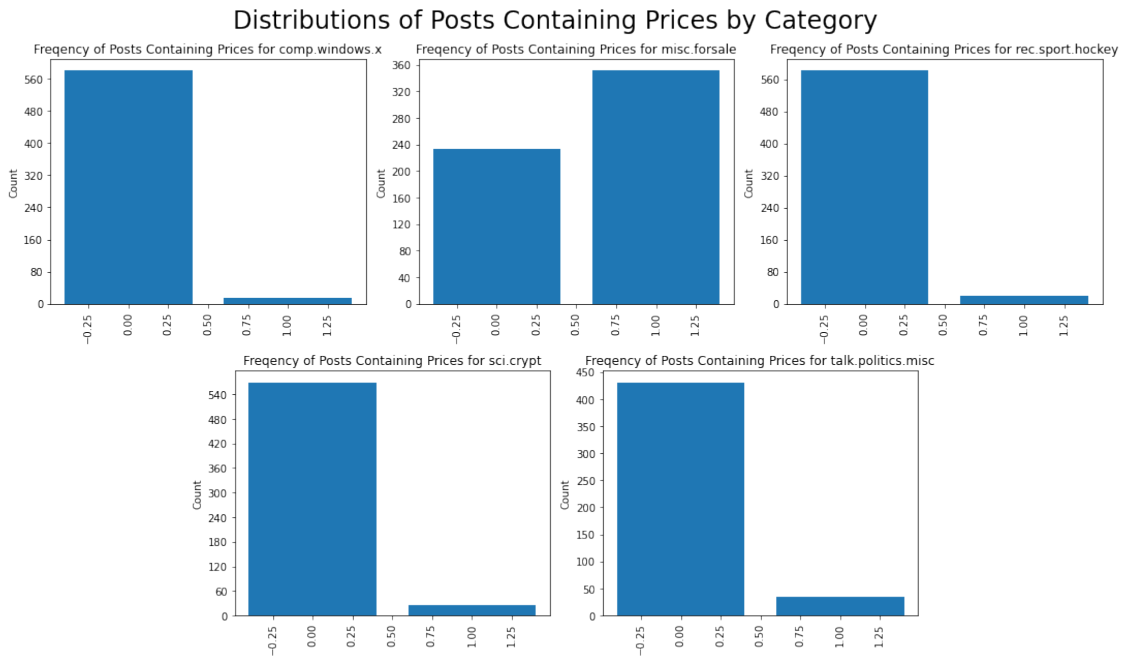
```
[52]: # Run this cell without changes
      fig, axes = setup_five_subplots()
      plot_distribution_of_column_by_category("num_sentences", axes, "Numbers of␣
       ↪Sentences for")
      fig.suptitle("Distributions of Sentence Counts by Category", fontsize=24);
```



Does this seem like a useful feature? Maybe. The distributions differ a bit, but it's hard to know if our model will pick up on this information. Let's go ahead and keep it.

Contains a Price The idea here is particularly to be able to distinguish the `misc.forsale` category, but it might also help with identifying the others. Let's use RegEx to check if the text contains a price:

[53]:
```
# Run this cell without changes

# Define a price as a dollar sign followed by 1-3 numbers,
# optional commas or decimals, 1-2 numbers after the decimal
# (we're not too worried about accidentally matching malformed prices)
price_query = r'\$(?:\d{1,3}[,.]?)+(?:\\d{1,2})?'

X_train["contains_price"] = X_train["text"].str.contains(price_query)

fig, axes = setup_five_subplots()
plot_distribution_of_column_by_category("contains_price", axes, "Freqency of␣
  ↪Posts Containing Prices for")
fig.suptitle("Distributions of Posts Containing Prices by Category",␣
  ↪fontsize=24);
```

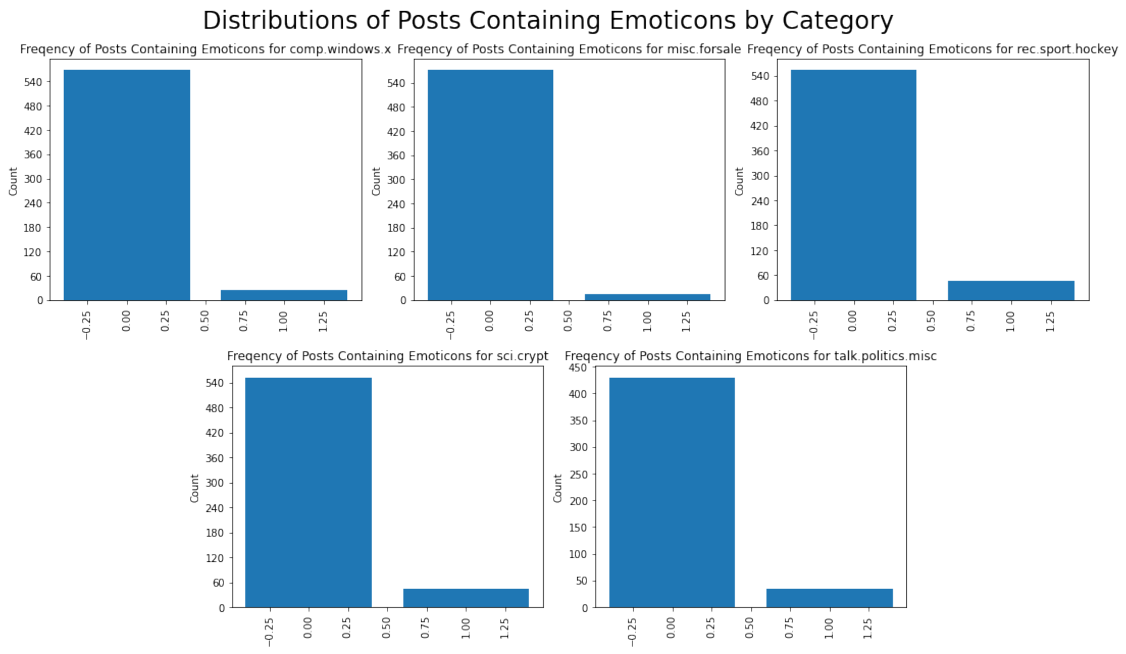


As we expected, the `misc.forsale` category looks pretty different from the others. More than half of those posts contain prices, whereas the overwhelming majority of posts in other categories do not contain prices. Let's include this in our final model.

Contains an Emoticon This is a bit silly, but we were wondering whether different categories feature different numbers of emoticons.

Here we define an emoticon as an ASCII character representing eyes, an optional ASCII character representing a nose, and an ASCII character representing a mouth.

```
[54]: # Run this cell without changes

emoticon_query = r'(?:[\:;X=B][-^]?[)\]3D([OP/\\|])(?:(?=\s))'

X_train["contains_emoticon"] = X_train["text"].str.contains(emoticon_query)

fig, axes = setup_five_subplots()
plot_distribution_of_column_by_category("contains_emoticon", axes, "Freqency of␣
↪Posts Containing Emoticons for")
fig.suptitle("Distributions of Posts Containing Emoticons by Category",␣
↪fontsize=24);
```



```
[56]: X_train.head()
```

```
[56]: text  \
0  # ## #i remain pro-choice, but when pro-choicers compare abortion in a\n# ##
#clinic to a religious ritual in a church, you have to start wondering\n# ## #a
bit if the pro-life criticism of abortion as modern human sacrifice\n# ##
#doesn't have a grain of truth to it.\n# \n# ## \n# ## ah, clayton, so i see
that you have found someone new to bash.  tell\n# ## me, how many pro-choicers
have comp…
1  usually when i start up an application, i first get the window outline\non my
display. i then have to click on the mouse button to actually\nplace the window
on the screen. yet when i specify the -geometry \noption the window appears
right away, the properties specified by\nthe -geometry argument. the question
now is:\n\nhow can i override the intermediary step of the user having to
specify\nw…
2                                                      \ntry reading between the
lines david - there are *strong* hints in there\nthat they're angling for nren
next, and the only conceivable meaning of\napplying this particular technology
to a computer network is that they\nintend it to be used in exclusion to any
other means of encryption.\n\ndon't be lulled by the wedge because its end looks
so thin.
3  \nagreed.  \n\n\nit is a failure of libertarianism if the ideology does not
provide any\nreasonable way to restrain such actions other than utopian dreams.
just\nas marxism "fails" to specify how pure communism is to be achieved
and\nthe state is to "wither away," libertarians frequently fail to show
how\nweakening the power of the state will result in improvement in the
```

```
human\ncondition.\n\n…
4
followup-to:kedz@wpi.wpi.edu \ndistribution: ne\norganization: worcester
polytechnic institute\nkeywords: \n\ni am looking for an inexpensive motorcycle,
nothing fancy, have to be able to do all maintinence my self. looking in the
<$400 range.

text_tokenized  \
0  [remain, pro, choice, but, when, pro, choicers, compare, abortion, in,
clinic, to, religious, ritual, in, church, you, have, to, start, wondering, bit,
if, the, pro, life, criticism, of, abortion, as, modern, human, sacrifice,
doesn, have, grain, of, truth, to, it, ah, clayton, so, see, that, you, have,
found, someone, new, to, bash, tell, me, how, many, pro, choicers, have,
compared, abortion…
1  [usually, when, start, up, an, application, first, get, the, window, outline,
on, my, display, then, have, to, click, on, the, mouse, button, to, actually,
place, the, window, on, the, screen, yet, when, specify, the, geometry, option,
the, window, appears, right, away, the, properties, specified, by, the,
geometry, argument, the, question, now, is, how, can, override, the,
intermediary, step,…
2          [try, reading, between, the, lines, david, there, are, strong, hints,
in, there, that, they, re, angling, for, nren, next, and, the, only,
conceivable, meaning, of, applying, this, particular, technology, to, computer,
network, is, that, they, intend, it, to, be, used, in, exclusion, to, any,
other, means, of, encryption, don, be, lulled, by, the, wedge, because, its,
end, looks, so, thin]
3  [agreed, it, is, failure, of, libertarianism, if, the, ideology, does, not,
provide, any, reasonable, way, to, restrain, such, actions, other, than,
utopian, dreams, just, as, marxism, fails, to, specify, how, pure, communism,
is, to, be, achieved, and, the, state, is, to, wither, away, libertarians,
frequently, fail, to, show, how, weakening, the, power, of, the, state, will,
result, in, impr…
4
[followup, to, kedz, wpi, wpi, edu, distribution, ne, organization, worcester,
polytechnic, institute, keywords, am, looking, for, an, inexpensive, motorcycle,
nothing, fancy, have, to, be, able, to, do, all, maintinence, my, self, looking,
in, the, 400, range]

   label  \
0      4
1      0
2      3
3      4
4      1

text_without_stopwords  \
0  [remain, pro, choice, pro, choicers, compare, abortion, clinic, religious,
ritual, church, start, wondering, bit, pro, life, criticism, abortion, modern,
human, sacrifice, grain, truth, ah, clayton, see, found, someone, new, bash,
tell, many, pro, choicers, compared, abortion, clinic, religious, ritual,
church, bet, seen, overwhelming, support, for, opinion, newsgroup, another,
seen, compariso…
1  [usually, start, application, first, get, window, outline, display, click,
mouse, button, actually, place, window, screen, yet, specify, geometry, option,
window, appears, right, away, properties, specified, geometry, argument,
question, override, intermediary, step, user, specify, window, position,
mouseclick, tried, explicitly, setting, window, size, position, alter, normal,
program, behavio…
2
[try, reading, lines, david, strong, hints, angling, for, nren, next,
conceivable, meaning, applying, particular, technology, computer, network,
intend, used, exclusion, means, encryption, lulled, wedge, end, looks, thin]
3  [agreed, failure, libertarianism, ideology, provide, reasonable, way,
restrain, actions, utopian, dreams, marxism, fails, specify, pure, communism,
achieved, state, wither, away, libertarians, frequently, fail, show, weakening,
power, state, result, improvement, human, condition, strawman, argument, fails,
several, grounds, case, limited, big, government, defined, would, point,
lebanon, somali…
4
[followup, kedz, wpi, wpi, edu, distribution, ne, organization, worcester,
polytechnic, institute, keywords, looking, for, inexpensive, motorcycle,
nothing, fancy, able, maintinence, self, looking, 400, range]

   num_sentences  contains_price  contains_emoticon
0             21           False              False
1              6           False              False
2              2           False              False
3              9           False              False
4              2            True              False
```

Well, that was a lot less definitive. Emoticons are fairly rare across categories. But, there are some small differences so let's go ahead and keep it.

Modeling with Vectorized Features + Engineered Features Let's combine our best vectorizer with these new features:

[57]:
```python
# Run this cell without changes

# Instantiate the vectorizer
tfidf = TfidfVectorizer(
    max_features=10,
    stop_words=stemmed_stopwords,
    tokenizer=stem_and_tokenize
)
```

```
# Fit the vectorizer on X_train["text"] and transform it
X_train_vectorized = tfidf.fit_transform(X_train["text"])

# Create a full df of vectorized + engineered features
X_train_vectorized_df = pd.DataFrame(X_train_vectorized.toarray(),␣
 ↪columns=tfidf.get_feature_names())
preprocessed_X_train = pd.concat([
    X_train_vectorized_df, X_train[["num_sentences", "contains_price",␣
 ↪"contains_emoticon"]]
], axis=1)
preprocessed_X_train
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

[57]:
```
      file       for       get       key      like       new       one  \
0      0.0  0.501934  0.000000  0.000000  0.400150  0.469658  0.000000
1      0.0  0.524938  0.851140  0.000000  0.000000  0.000000  0.000000
2      0.0  0.556285  0.000000  0.000000  0.000000  0.000000  0.000000
3      0.0  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
4      0.0  1.000000  0.000000  0.000000  0.000000  0.000000  0.000000
…     …        …         …         …         …         …         …
2833   0.0  0.588738  0.477293  0.000000  0.469351  0.000000  0.000000
2834   0.0  0.879732  0.475469  0.000000  0.000000  0.000000  0.000000
2835   0.0  0.700743  0.000000  0.000000  0.000000  0.000000  0.713414
2836   0.0  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
2837   0.0  0.180021  0.291888  0.386198  0.287031  0.000000  0.000000

         peopl       use     would  num_sentences  contains_price  \
0     0.476249  0.374901  0.000000             21           False
1     0.000000  0.000000  0.000000              6           False
2     0.000000  0.830992  0.000000              2           False
3     0.524754  0.000000  0.851254              9           False
4     0.000000  0.000000  0.000000              2            True
…        …         …         …            …              …
2833  0.000000  0.000000  0.453088             11           False
2834  0.000000  0.000000  0.000000             14           False
2835  0.000000  0.000000  0.000000              5            True
2836  0.000000  0.000000  0.000000              1           False
2837  0.000000  0.806758  0.000000              7           False

      contains_emoticon
0                 False
```

```
1                 False
2                 False
3                 False
4                 False
…                 …
2833              False
2834              False
2835              False
2836              False
2837              False

[2838 rows x 13 columns]
```

[58]:
```
# Run this cell without changes
preprocessed_cv = cross_val_score(baseline_model, preprocessed_X_train, y_train)
preprocessed_cv
```

[58]:
```
array([0.47535211, 0.46302817, 0.45422535, 0.49206349, 0.48148148])
```

[59]:
```
# Run this cell without changes
print("Stemmed:           ", stemmed_cv.mean())
print("Fully preprocessed:", preprocessed_cv.mean())
```

```
Stemmed:            0.4566802046848995
Fully preprocessed: 0.4732301214695581
```

Ok, another small improvement! We're still a bit below 50% accuracy, but we're getting improvements every time.

1.7.4 Increasing `max_features`

Right now we are only allowing the model to look at the tf-idf of the top 10 most frequent tokens. If we allow it to look at all possible tokens, that could lead to high dimensionality issues (especially if we have more rows than columns), but there is a lot of room between 10 and `len(X_train)` features:

[60]:
```
# Run this cell without changes
len(X_train)
```

[60]:
```
2838
```

(In other words, setting `max_features` to 2838 would mean an equal number of rows and columns, something that can cause problems for many model algorithms.)

Let's try increasing `max_features` from 10 to 200:

[64]:
```
# Replace None with appropriate code

# Instantiate the vectorizer
```

```
tfidf = TfidfVectorizer(
    max_features=200,
    stop_words=stemmed_stopwords,
    tokenizer=stem_and_tokenize
)

# Fit the vectorizer on X_train["text"] and transform it
X_train_vectorized = tfidf.fit_transform(X_train["text"])

# Create a full df of vectorized + engineered features
X_train_vectorized_df = pd.DataFrame(X_train_vectorized.toarray(),␣
 ↪columns=tfidf.get_feature_names())
final_X_train = pd.concat([
    X_train_vectorized_df, X_train[["num_sentences", "contains_price",␣
 ↪"contains_emoticon"]]
], axis=1)
final_X_train
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

[64]:
```
       00   10   11   12   13   14   15   16        17   18  …  widget  win  \
0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
1     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
2     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
3     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
4     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
…     …   …   …   …   …   …   …   …       …   …  …     …   …
2833  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
2834  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
2835  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.352469  0.0  …     0.0  0.0
2836  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0
2837  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.000000  0.0  …     0.0  0.0

        window  without      work     would  year  num_sentences  \
0     0.000000      0.0  0.000000  0.000000   0.0             21
1     0.800693      0.0  0.134936  0.000000   0.0              6
2     0.000000      0.0  0.000000  0.000000   0.0              2
3     0.000000      0.0  0.000000  0.167189   0.0              9
4     0.000000      0.0  0.000000  0.000000   0.0              2
…          …      …       …       …   …            …
2833  0.000000      0.0  0.000000  0.040110   0.0             11
2834  0.000000      0.0  0.000000  0.000000   0.0             14
2835  0.000000      0.0  0.237706  0.000000   0.0              5
```

```
2836  0.000000      0.0  0.000000  0.000000   0.0                1
2837  0.000000      0.0  0.000000  0.000000   0.0                7

      contains_price  contains_emoticon
0              False              False
1              False              False
2              False              False
3              False              False
4               True              False
…              …                 …
2833           False              False
2834           False              False
2835            True              False
2836           False              False
2837           False              False

[2838 rows x 203 columns]
```

[65]:
```
# Run this cell without changes

final_cv = cross_val_score(baseline_model, final_X_train, y_train)
final_cv
```

[65]:
```
array([0.75704225, 0.77464789, 0.77288732, 0.77954145, 0.75837743])
```

Nice! Our model was able to learn a lot more with these added features. Let's say this is our final modeling process and move on to a final evaluation.

1.8 5. Evaluate a Final Model on the Test Set

Instantiate the model, fit it on the full training set and check the score:

[66]:
```
# Run this cell without changes
final_model = MultinomialNB()

final_model.fit(final_X_train, y_train)
final_model.score(final_X_train, y_train)
```

[66]:
```
0.7914023960535589
```

Create a vectorized version of `X_test`'s text:

[67]:
```
# Run this cell without changes

# Note that we just transform, don't fit_transform
X_test_vectorized = tfidf.transform(X_test["text"])
```

Feature engineering for `X_test`:

[68]:
```
# Run this cell without changes
X_test["num_sentences"] = X_test["text"].apply(lambda x: len(sent_tokenize(x)))
X_test["contains_price"] = X_test["text"].str.contains(price_query)
X_test["contains_emoticon"] = X_test["text"].str.contains(emoticon_query)
```

Putting it all together:

[69]:
```
# Run this cell without changes
X_test_vectorized_df = pd.DataFrame(X_test_vectorized.toarray(), columns=tfidf.
 ↪get_feature_names())
final_X_test = pd.concat([
    X_test_vectorized_df, X_test[["num_sentences", "contains_price",␣
 ↪"contains_emoticon"]]
], axis=1)
final_X_test
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will
be removed in 1.2. Please use get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

[69]:
```
       00        10        11        12   13   14   15        16   17   18  \
0     0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
1     0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
2     0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
3     0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
4     0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
…    …        …         …         …      …    …    …        …      …    …
1885  0.0  0.000000  0.000000  0.364446  0.0  0.0  0.0  0.000000  0.0  0.0
1886  0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
1887  0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
1888  0.0  0.000000  0.000000  0.000000  0.0  0.0  0.0  0.000000  0.0  0.0
1889  0.0  0.122374  0.145127  0.000000  0.0  0.0  0.0  0.140736  0.0  0.0

      …  widget      win  window  without     work     would      year  \
0     …     0.0  0.00000     0.0      0.0  0.00000  0.000000  0.297128
1     …     0.0  0.00000     0.0      0.0  0.00000  0.260175  0.000000
2     …     0.0  0.00000     0.0      0.0  0.06002  0.096676  0.000000
3     …     0.0  0.29106     0.0      0.0  0.00000  0.000000  0.107175
4     …     0.0  0.00000     0.0      0.0  0.00000  0.000000  0.000000
…    …      …       …       …       …        …        …         …
1885  …     0.0  0.00000     0.0      0.0  0.00000  0.000000  0.000000
1886  …     0.0  0.00000     0.0      0.0  0.00000  0.000000  0.000000
1887  …     0.0  0.00000     0.0      0.0  0.00000  0.189733  0.000000
1888  …     0.0  0.00000     0.0      0.0  0.00000  0.000000  0.000000
1889  …     0.0  0.00000     0.0      0.0  0.00000  0.000000  0.208741
```

```
      num_sentences  contains_price  contains_emoticon
0                 4           False              False
1                 5           False              False
2                14           False              False
3                10           False              False
4                 5           False              False
…               …             …                …
1885             17            True              False
1886              3            True              False
1887              7           False              False
1888              5           False              False
1889             21           False              False

[1890 rows x 203 columns]
```

Scoring on the test set:

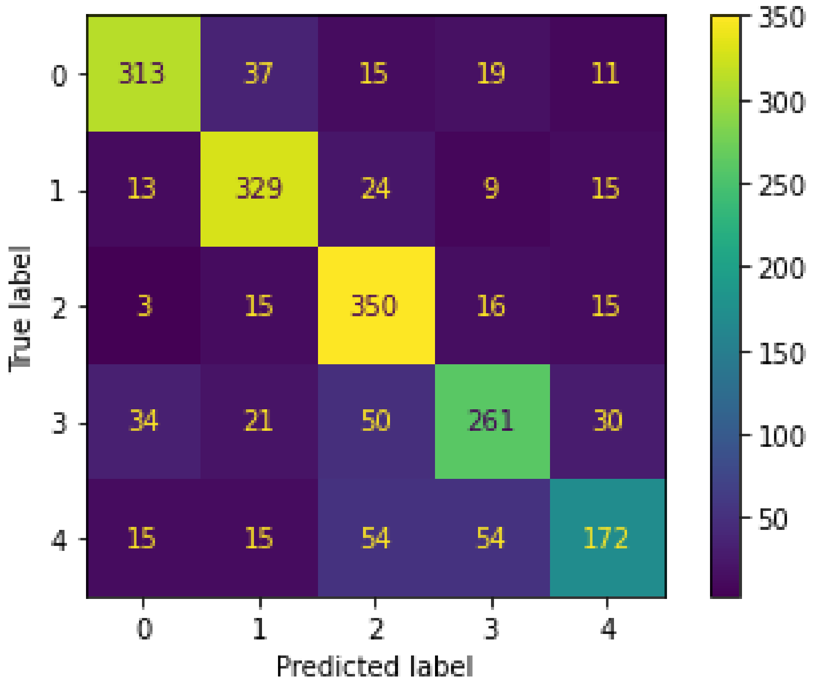
```
[70]: # Run this cell without changes
      final_model.score(final_X_test, y_test)
```

```
[70]: 0.753968253968254
```

Plotting a confusion matrix:

```
[71]: # Run this cell without changes
      from sklearn.metrics import plot_confusion_matrix
      plot_confusion_matrix(final_model, final_X_test, y_test);
```

```
/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function
plot_confusion_matrix is deprecated; Function `plot_confusion_matrix` is
deprecated in 1.0 and will be removed in 1.2. Use one of the class methods:
ConfusionMatrixDisplay.from_predictions or
ConfusionMatrixDisplay.from_estimator.
  warnings.warn(msg, category=FutureWarning)
```



Recall that these are the names associated with the labels:

[72]:
```
# Run this cell without changes
target_values_and_names = train_target_counts.drop("count", axis=1)
target_values_and_names
```

[72]:
```
                           target name
target value
2                     rec.sport.hockey
3                            sci.crypt
0                       comp.windows.x
1                         misc.forsale
4                   talk.politics.misc
```

1.8.1 Interpreting Results

Interpret the results seen above. How well did the model do? How does it compare to random guessing? What can you say about the cases that the model was most likely to mislabel? If this were a project and you were describing next steps, what might those be?

[73]:
```
# Replace None with appropriate text
"""
The model did fairly well. The naive accuracy rate (random guessing)
was around 20%, and we are getting about 75% of the labels right.
```

```
Looking at the confusion matrix, the largest areas of mislabeled posts
were 50 (predicted rec.sport.hockey, actually sci.crypt), 54 (predicted
rec.sport.hockey, actually talk.politics.misc), and 54 (predicted
sci.crypt, actually talk.politics.misc).

Next steps for this process might be to examine some of these mislabeled
examples to see if we can understand what went wrong, and potentially
perform additional feature engineering to help the model label these
instances correctly. It's also possible that there is just some
irreducible error in this dataset, where those posts "look like" they
should be in a different category than they really are.
"""
```

[73]:
```
'\nThe model did fairly well. The naive accuracy rate (random guessing)\nwas
around 20%, and we are getting about 75% of the labels right.\n\nLooking at the
confusion matrix, the largest areas of mislabeled posts\nwere 50 (predicted
rec.sport.hockey, actually sci.crypt), 54 (predicted\nrec.sport.hockey, actually
talk.politics.misc), and 54 (predicted\nsci.crypt, actually
talk.politics.misc).\n\nNext steps for this process might be to examine some of
these mislabeled\nexamples to see if we can understand what went wrong, and
potentially\nperform additional feature engineering to help the model label
these\ninstances correctly. It\'s also possible that there is just
some\nirreducible error in this dataset, where those posts "look like"
they\nshould be in a different category than they really are.\n'
```

1.9 Summary

In this lab, we used our NLP skills to clean, preprocess, explore, and fit models to text data for classification. This wasn't easy — great job!!