### **Download NLTK data**

In [1]:

**import** nltk

nltk.download()

dir(nltk)

### **What can you do with NLTK?**

In [6]:

**from** nltk.corpus **import** stopwords

​

print(stopwords.words('english')[:10])

print(stopwords.words('english')[0:500:25])

Structured Data vs Unstructured Data

**Unstructured Data** - Binary Data, No delimiters, No indication of any rows

80% of data is unstructured data

Messy Data

### **Read in semi-structured text data**

In [1]:

*# Read in the raw text*

rawData **=** open("SMSSpamCollection.tsv").read()

​

*# Print the raw data*

rawData[0:500]

Regular Expressions (regex)

Text String for describing a search pattern

Ex: `[j-q]` can search for all single characters between j and q

`[j-q]+` can search words with letters between j and q

**Why regex?** - when dealing with text data, identifying whitespace btw words/tokens

Identifying/creating delimiters or end-of-line escape characters

Removing punctuation or numbers from your text

Cleaning html tags

Identify text patterns

Use Cases: Passwords meet criteria

Search url

Search files

Document scraping

**Useful Methods for tokenizing:**

**findall(), split()**

**USeful Regexs for tekenizing**

**`\W` & `\w` - words**

**`\S` & `\s` - whitespaces**

Raw Text - model cant distinguish words

Tokenize - tell the model what to look at

Clean Text - remove stopwords, punctuation, stemming etc

Vectorize - convert to numeric form

Machine Learning Algorithm - fit/train model

### **Pre-processing text data**

Cleaning up the text data is necessary to highlight attributes that you're going to want your machine learning system to pick up on. Cleaning (or pre-processing) the data typically consists of a number of steps:

1. **Remove punctuation**
2. **Tokenization**
3. **Remove stopwords**
4. Lemmatize/Stem

The first three steps are covered in this chapter as they're implemented in pretty much any text cleaning pipeline. Lemmatizing and stemming are covered in the next chapter as they're helpful but not critical.

### **Remove punctuation**

*import string*

*def remove\_punct(text):*

*text\_nopunct = "".join([char for char in text if char not in string.punctuation])*

*return text\_nopunct*

*data['body\_text\_clean'] = data['body\_text'].apply(lambda x: remove\_punct(x))*

*data.head()*

### **Tokenization**

*import re*

*def tokenize(text):*

*tokens = re.split("\W+", text)*

*return tokens*

*data['body\_text\_tokenized'] = data['body\_text\_clean'].apply(lambda x : tokenize(x.lower()))*

*data.head()*

### **Remove stopwords**

In [15]:

**import** nltk

​

stopword **=** nltk.corpus.stopwords.words('english')

In [16]:

**def** remove\_stopwords(tokenized\_list):

text **=** [word **for** word **in** tokenized\_list **if** word **not** **in** stopword]

**return** text

​

data['body\_text\_nostop'] **=** data['body\_text\_tokenized'].apply(**lambda** x: remove\_stopwords(x))

data.head()

**Stemming**

Process ofreducing inflected (or sometimes derived) words to their word stem or root

Crudely chopping off teh end of the word to leave only the base

Examples: Stemming/stemmed -> Stem, Electricity?electrical -> Electric, Berries/Berry -> Berri

Connection/connected?connective -> connect

**Why Stemming**

Reduces the corpus of words the model is exposed to

Explicitly correlates words with similar meanings

**Types of Stemmers:**

\***Porter Stemmer**

Snowball Stemmer

Lancester Stemmer

Regex Based Stemmer

### **Test out Porter stemmer**

In [1]:

**import** nltk

​

ps **=** nltk.PorterStemmer()

In [2]:

dir(ps)

### **Stem text**

In [13]:

**def** stemming(tokenized\_text):

text **=** [ps.stem(word) **for** word **in** tokenized\_text]

**return** text

data['body\_text\_stemmed'] **=** data['body\_text\_nostop'].apply(**lambda** x: stemming(x))

​

data.head()

***Lemmatizing***

Process of grouping together the inflected forms of a word so they can be analyzed as a single term, identified by the word’s lemma

Using vocabulary analysis of words aiming to remove inflectional endings to return the dictionary form of word.

Stemming just chops, less accurate, simple rules, fast

Lemmatizing more accurate, uses more informed analysis to create group of words with similar meaning based on context

Lemmatizing can be computationally expensive

We use Word Net Lemmatizer

### **Test out WordNet lemmatizer (read more about WordNet** [**here**](https://wordnet.princeton.edu/)**)**

In [1]:

**import** nltk

​

wn **=** nltk.WordNetLemmatizer()

ps **=** nltk.PorterStemmer()

In [2]:

dir(wn)

### **Lemmatize text**

In [12]:

**def** lemmatizer(tokenized\_text):

text **=** [wn.lemmatize(word) **for** word **in** tokenized\_text]

**return** text

​

data['body\_text\_lemmatized'] **=** data['body\_text\_nostop'].apply(**lambda** x: lemmatizer(x))

​

data.head()

**Vectorizing**

Process of encoding text as integers to create feature vectors

**Feature Vector**

An n-dimenisional vector of numerical features that represent some object

**Document Term Matrix -** we take a dataset that has one line per document(entry) with the cell entry as the actual text message and then we’re converting into a document that still has one line per document, but then you have every word used across all documents as the columns of your matrix and then within each cell it is counting how many times that certain word appeared in that document

**Why Vectorizing**

So that Python and the algorithm used ML can understand

**Types of Vectorization**

Count Vectorization

N-grams

Term frequency - inverse document frequency (TF-IDF)

### **Apply CountVectorizer**

In [8]:

**from** sklearn.feature\_extraction.text **import** CountVectorizer

​

count\_vect **=** CountVectorizer(analyzer **=** clean\_text)

X\_counts **=** count\_vect.fit\_transform(data['body\_text'])

print(X\_counts.shape)

print(count\_vect.get\_feature\_names\_out())

Count Vectorizer has other hyper parameters and are set to default values when not specified

### **Apply CountVectorizer to smaller sample**

In [5]:

data\_sample **=** data[0:20]

​

count\_vect\_sample **=** CountVectorizer(analyzer**=**clean\_text)

X\_counts\_sample **=** count\_vect\_sample.fit\_transform(data\_sample['body\_text'])

print(X\_counts\_sample.shape)

print(count\_vect\_sample.get\_feature\_names())

### **Vectorizers output sparse matrices**

***Sparse Matrix****: A matrix in which most entries are 0. In the interest of efficient storage, a sparse matrix will be stored by only storing the locations of the non-zero elements.*

In [6]:

X\_counts\_sample

Out[6]:

<20x192 sparse matrix of type '<class 'numpy.int64'>'

with 218 stored elements in Compressed Sparse Row format>

In [8]:

X\_counts\_df **=** pd.DataFrame(X\_counts\_sample.toarray())

X\_counts\_df

X\_counts\_df.columns = count\_vect\_sample.get\_feature\_names()

X\_counts\_df #Assigns column names

# **Vectorizing Raw Data: N-Grams**

### **N-Grams**

Creates a document-term matrix where counts still occupy the cell but instead of the columns representing single terms, they represent all combinations of adjacent words of length n in your text.

"NLP is an interesting topic"

| **n** | **Name** | **Tokens** |
| --- | --- | --- |
| 2 | bigram | ["nlp is", "is an", "an interesting", "interesting topic"] |
| 3 | trigram | ["nlp is an", "is an interesting", "an interesting topic"] |
| 4 | four-gram | ["nlp is an interesting", "is an interesting topic"] |

### **Apply CountVectorizer (w/ N-Grams) to smaller sample**

In [6]:

data\_sample **=** data[0:20]

​

ngram\_vect\_sample **=** CountVectorizer(ngram\_range**=**(2,2))

X\_counts\_sample **=** ngram\_vect\_sample.fit\_transform(data\_sample['cleaned\_text'])

print(X\_counts\_sample.shape)

print(ngram\_vect\_sample.get\_feature\_names\_out())

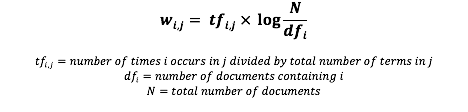
X\_counts\_df = pd.DataFrame(X\_counts\_sample.toarray())

X\_counts\_df.columns = ngram\_vect\_sample.get\_feature\_names\_out()

# **Vectorizing Raw Data: TF-IDF**

### **TF-IDF**

Creates a document-term matrix where the columns represent single unique terms (unigrams) but the cell represents a weighting meant to represent how important a word is to a document.



### **Apply TfidfVectorizer to smaller sample**

In [7]:

data\_sample **=** data[:20]

​

tfidf\_vect\_sample **=** TfidfVectorizer(analyzer **=** clean\_text)

X\_tfidf\_sample **=** tfidf\_vect\_sample.fit\_transform(data\_sample['body\_text'])

print(X\_tfidf\_sample.shape)

print(tfidf\_vect\_sample.get\_feature\_names\_out())

### **Vectorizers output sparse matrices**

***Sparse Matrix****: A matrix in which most entries are 0. In the interest of efficient storage, a sparse matrix will be stored by only storing the locations of the non-zero elements.*

In [8]:

X\_tfidf\_df **=** pd.DataFrame(X\_tfidf\_sample.toarray())

In [9]:

X\_tfidf\_df.columns **=** tfidf\_vect\_sample.get\_feature\_names\_out()

X\_tfidf\_df.head()

***FEATURE ENGINEERING***

Creating new features or transforming your existing features to get the most out of your data

Creating New Features

-Length of text field

-Percentage of characters that are punctuation in the text

-Percentage of characters that are capitalized

Transformation Exisiting Features

-Power transformation (square, square root)

-Standardizing data

### **Create feature for text message length**

In [2]:

data['body\_len'] **=** data['body\_text'].apply(**lambda** x: len(x)**-**x.count(" "))

​

data.head()

Out[2]:

### **Create feature for % of text that is punctuation**

In [5]:

**import** string

​

**def** count\_punct(text):

count **=** sum([1 **for** char **in** text **if** char **in** string.punctuation])

**return** round(count**/**(len(text) **-** text.count(" ")), 3)**\***100

​

data['punct%'] **=** data['body\_text'].apply(**lambda** x: count\_punct(x))

### **Evaluate created features**

In [7]:

**from** matplotlib **import** pyplot

**import** numpy **as** np

**%**matplotlib inline

In [11]:

bins **=** np.linspace(0, 200, 40)

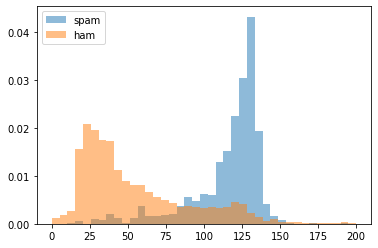
​

pyplot.hist(data[data['label']**==**'spam']['body\_len'], bins, alpha **=** 0.5, density **=** **True**, label **=** 'spam')

pyplot.hist(data[data['label']**==**'ham']['body\_len'], bins, alpha **=** 0.5, density **=** **True**, label **=** 'ham')

pyplot.legend(loc**=**'upper left')

pyplot.show()



In [12]:

bins **=** np.linspace(0, 50, 40)

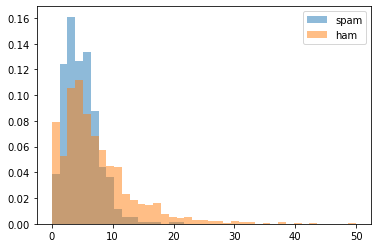
​

pyplot.hist(data[data['label']**==**'spam']['punct%'], bins, alpha **=** 0.5, density **=** **True**, label **=** 'spam')

pyplot.hist(data[data['label']**==**'ham']['punct%'], bins, alpha **=** 0.5, density **=** **True**, label **=** 'ham')

pyplot.legend(loc**=**'upper right')

pyplot.show()

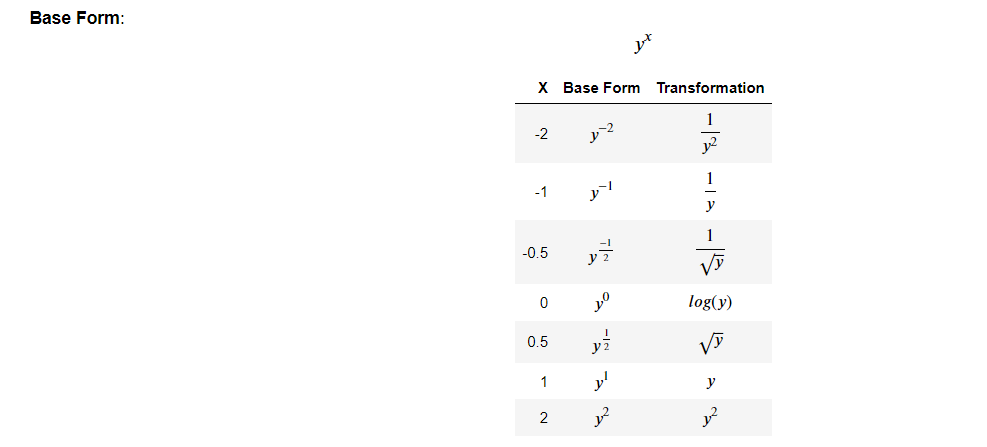


# **Feature Engineering: Transformations**

Process that alters each data point in a certain column in a systematic way (eg -x^2, sqrt(x))

### **Transform the punctuation % feature**

### **Box-Cox Power Transformation**

****

**Process**

1. Determine what range of exponents to test
2. Apply each transformation to each value of your chosen feature
3. Use some criteria to determine which of the transformations yield the best distribution

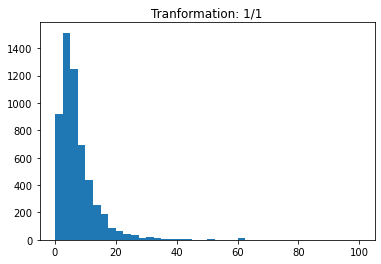
In [11]:

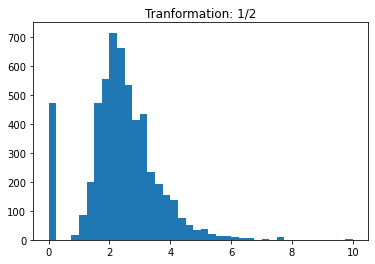
**for** i **in** [1,2,3,4,5]:

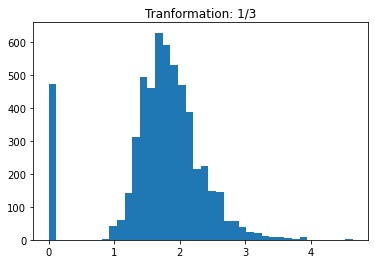
pyplot.hist(data['punct%']**\*\***(1**/**i), bins **=** 40)

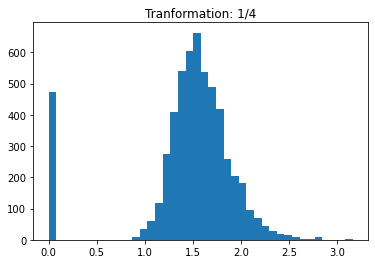
pyplot.title("Tranformation: 1/{}".format(str(i)))

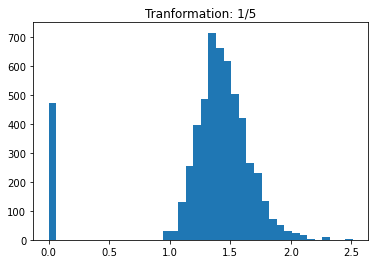
pyplot.show()











***MACHINE LEARNING***

**Tools used to evaluate our models**

1. Holdout Test Set

Sample of data not used in fitting a model for the purpose of evaluating the model’s ability to generalize unseen data

1. K-Fold Cross Validation

The full data set is divided into k-subsets and the holdout method is repeated k times. Each time, one of the k-1 subsets are put together to be used to train the model

**Evaluation Metrics**

1. **Classification**: Accuracy = #predicted correctly/Total obs

Precision = #predicted as spam that are actually spam/

total # predicted as spam

Recall = #predicted as spam that are actually spam/

Total # that are actually spam

**Ensemble Method**: Technique that creates multiple models and then combines them to produce better results than any of the single models individually

Benefits: can be used for regression and classification, easily handles outliers, missing values etc, accepts various types of input, less likely to overfit, outputs feature importance

**Random Forest**: Ensemble learning method that constructs a collection of decision trees and then aggregates the predictions of each tree to determine the final prediction

# **Building Machine Learning Classifiers: Building a basic Random Forest model**

### **Read in & clean text**

In [1]:

**import** nltk

**import** pandas **as** pd

**import** re

**from** sklearn.feature\_extraction.text **import** TfidfVectorizer

**import** string

​

stopwords **=** nltk.corpus.stopwords.words('english')

ps **=** nltk.PorterStemmer()

​

data **=** pd.read\_csv("SMSSpamCollection.tsv", sep**=**'\t')

data.columns **=** ['label', 'body\_text']

​

**def** count\_punct(text):

count **=** sum([1 **for** char **in** text **if** char **in** string.punctuation])

**return** round(count**/**(len(text) **-** text.count(" ")), 3)**\***100

​

data['body\_len'] **=** data['body\_text'].apply(**lambda** x: len(x) **-** x.count(" "))

data['punct%'] **=** data['body\_text'].apply(**lambda** x: count\_punct(x))

​

**def** clean\_text(text):

text **=** "".join([word.lower() **for** word **in** text **if** word **not** **in** string.punctuation])

tokens **=** re.split('\W+', text)

text **=** [ps.stem(word) **for** word **in** tokens **if** word **not** **in** stopwords]

**return** text

​

tfidf\_vect **=** TfidfVectorizer(analyzer**=**clean\_text)

X\_tfidf **=** tfidf\_vect.fit\_transform(data['body\_text'])

​

X\_features **=** pd.concat([data['body\_len'], data['punct%'], pd.DataFrame(X\_tfidf.toarray())], axis**=**1)

X\_features.head()

Out[1]:

|  | **body\_len** | **punct%** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **...** | **8094** | **8095** | **8096** | **8097** | **8098** | **8099** | **8100** | **8101** | **8102** | **8103** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 128 | 4.7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **1** | 49 | 4.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **2** | 62 | 3.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **3** | 28 | 7.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **4** | 135 | 4.4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

5 rows × 8106 columns

### **Explore RandomForestClassifier Attributes & Hyperparameters**

In [2]:

**from** sklearn.ensemble **import** RandomForestClassifier

In [3]:

print(dir(RandomForestClassifier))

print(RandomForestClassifier())

['\_\_abstractmethods\_\_', '\_\_annotations\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setstate\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_abc\_impl', '\_check\_feature\_names', '\_check\_n\_features', '\_compute\_oob\_predictions', '\_estimator\_type', '\_get\_oob\_predictions', '\_get\_param\_names', '\_get\_tags', '\_make\_estimator', '\_more\_tags', '\_parameter\_constraints', '\_repr\_html\_', '\_repr\_html\_inner', '\_repr\_mimebundle\_', '\_required\_parameters', '\_set\_oob\_score\_and\_attributes', '\_validate\_X\_predict', '\_validate\_data', '\_validate\_estimator', '\_validate\_params', '\_validate\_y\_class\_weight', 'apply', 'base\_estimator\_', 'decision\_path', 'estimator\_', 'feature\_importances\_', 'fit', 'get\_params', 'predict', 'predict\_log\_proba', 'predict\_proba', 'score', 'set\_params']

RandomForestClassifier()

### **Explore RandomForestClassifier through Cross-Validation**

In [4]:

**from** sklearn.model\_selection **import** KFold, cross\_val\_score

In [6]:

rf **=** RandomForestClassifier(n\_jobs**=-**1)

k\_fold **=** KFold(n\_splits**=**5)

cross\_val\_score(rf, X\_features.values, data['label'].values, cv**=**k\_fold, scoring**=**'accuracy', n\_jobs**=-**1)

Out[6]:

array([0.97576302, 0.98114901, 0.97663971, 0.96585804, 0.97124888])

# **Building Machine Learning Classifiers: Random Forest on a holdout test set**

### **Read in & clean text**

In [1]:

**import** nltk

**import** pandas **as** pd

**import** re

**from** sklearn.feature\_extraction.text **import** TfidfVectorizer

**import** string

​

stopwords **=** nltk.corpus.stopwords.words('english')

ps **=** nltk.PorterStemmer()

​

data **=** pd.read\_csv("SMSSpamCollection.tsv", sep**=**'\t')

data.columns **=** ['label', 'body\_text']

​

**def** count\_punct(text):

count **=** sum([1 **for** char **in** text **if** char **in** string.punctuation])

**return** round(count**/**(len(text) **-** text.count(" ")), 3)**\***100

​

data['body\_len'] **=** data['body\_text'].apply(**lambda** x: len(x) **-** x.count(" "))

data['punct%'] **=** data['body\_text'].apply(**lambda** x: count\_punct(x))

​

**def** clean\_text(text):

text **=** "".join([word.lower() **for** word **in** text **if** word **not** **in** string.punctuation])

tokens **=** re.split('\W+', text)

text **=** [ps.stem(word) **for** word **in** tokens **if** word **not** **in** stopwords]

**return** text

​

tfidf\_vect **=** TfidfVectorizer(analyzer**=**clean\_text)

X\_tfidf **=** tfidf\_vect.fit\_transform(data['body\_text'])

​

X\_features **=** pd.concat([data['body\_len'], data['punct%'], pd.DataFrame(X\_tfidf.toarray())], axis**=**1)

X\_features.head()

Out[1]:

|  | **body\_len** | **punct%** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **...** | **8094** | **8095** | **8096** | **8097** | **8098** | **8099** | **8100** | **8101** | **8102** | **8103** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 128 | 4.7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **1** | 49 | 4.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **2** | 62 | 3.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **3** | 28 | 7.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **4** | 135 | 4.4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

5 rows × 8106 columns

### **Explore RandomForestClassifier through Holdout Set**

In [2]:

**from** sklearn.metrics **import** precision\_recall\_fscore\_support **as** score

**from** sklearn.model\_selection **import** train\_test\_split

In [5]:

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X\_features.values, data['label'].values, test\_size **=** 0.2)

In [6]:

**from** sklearn.ensemble **import** RandomForestClassifier

​

rf **=** RandomForestClassifier(n\_estimators **=** 50, max\_depth **=** 20, n\_jobs **=** **-**1)

rf\_model **=** rf.fit(X\_train, y\_train)

In [9]:

y\_pred **=** rf\_model.predict(X\_test)

precision,recall,fscore,support **=** score(y\_test, y\_pred, pos\_label **=** 'spam', average **=** 'binary')

In [10]:

print('Precision: {} / Recall: {} / Accuracy: {}'.format(round(precision, 3),

round(recall, 3),

round((y\_pred**==**y\_test).sum() **/** len(y\_pred),3)))

Precision: 1.0 / Recall: 0.601 / Accuracy: 0.949