In [1]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

from IPython.display import display, JSON
from pprint import pprint

from random import shuffle
from statistics import mean
```

In [2]:

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, SnowballStemmer, WordNetLemmatizer
from nltk.book import *
```

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

In [3]:

```
from nltk.sentiment import SentimentIntensityAnalyzer

from sklearn.naive_bayes import BernoulliNB, ComplementNB, MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

In [ ]:

```

```

Natural Language Processing With Python's NLTK Package

Natural Language Processing With Python's NLTK Package

Tokenizing

In [ ]:

```
example_string = """
Muad'Dib learned rapidly because his first training was in how to learn.
And the first lesson of all was the basic trust that he could learn.
It's shocking to find how many people do not believe they can learn,
and how many more believe learning to be difficult.
"""

example_string = example_string.strip("\n").replace("\n", " ")
example_string
```

In [ ]:

```
sent_tokenize(example_string)
```

In [ ]:

```
print(word_tokenize(example_string))
```

Filtering Stop Words

In [4]:

```
stop_words = set(stopwords.words("english"))
```

In [ ]:

```
print(stop_words)
```

In [ ]:

```
worf_quote = "Sir, I protest. I am not a merry man!"
words_in_quote = word_tokenize(worf_quote)
```

In [ ]:

```
filtered_list = [word for word in words_in_quote if not word.casefold() in stop
filtered_list
```

Stemming

PorterStemmer

In [ ]:

```
stemmer = PorterStemmer()
```

In [ ]:

```
string_for_stemming = """
The crew of the USS Discovery discovered many discoveries.
Discovering is what explorers do.
"""

string_for_stemming = string_for_stemming.strip("\n").replace("\n", " ")
string_for_stemming
```

In [ ]:

```
words = word_tokenize(string_for_stemming)
print(words)
```

In [ ]:

```
stemmed_words = [stemmer.stem(word) for word in words]
print(stemmed_words)
```

SnowballStemmer

In [ ]:

```
stemmer = SnowballStemmer("english")
```

In [ ]:

```
stemmed_words = [stemmer.stem(word) for word in words]
print(stemmed_words)
```

Tagging

In [ ]:

```
sagan_quote = """
If you wish to make an apple pie from scratch,
you must first invent the universe.
"""

sagan_quote = sagan_quote.strip("\n").replace("\n", " ")
sagan_quote
```

In [ ]:

```
words_in_sagan_quote = word_tokenize(sagan_quote)
print(words_in_sagan_quote)
```

In [ ]:

```
nltk.pos_tag(words_in_sagan_quote)
```

In [ ]:

```
# [lemmatizer.lemmatize(word, pos) for word, pos in nltk.pos_tag(words_in_sagan
```

In [ ]:

```
nltk.pos_tag(words_in_sagan_quote)
```

In [ ]:

```
nltk.help.upenn_tagset()
```

In [ ]:

```
>>> jabberwocky_excerpt = """
'Twas brillig, and the slithy toves did gyre and gimble in the wabe:
all mimsy were the borogoves, and the mome raths outgrabe.
"""

jabberwocky_excerpt = jabberwocky_excerpt.strip("\n").replace("\n", " ")
jabberwocky_excerpt
```

In [ ]:

```
words_in_excerpt = word_tokenize(jabberwocky_excerpt)
```

In [ ]:

```
nltk.pos_tag(words_in_excerpt)
```

Lemmatizing

Note: A *lemma* is a word that represents a whole group of words, and that group of words is called a lexeme. For example, if you were to look up the word **“blending”** in a dictionary, then you’d need to look at the entry for **“blend,”** but you would find “blending” listed in that entry. In this example, “blend” is the *lemma*, and “blending” is part of the *lexeme*. So when you *lemmatize* a word, you are reducing it to its *lemma*.

In [ ]:

```
lemmatizer = WordNetLemmatizer()
```

In [ ]:

```
lemmatizer.lemmatize("scarves")
```

In [ ]:

```
string_for_lemmatizing = "The friends of DeSoto love scarves."
```

In [ ]:

```
words = word_tokenize(string_for_lemmatizing)
words
```

In [ ]:
```
lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
lemmatized_words
```

In [ ]:
```
lemmatizer.lemmatize("worst")
lemmatizer.lemmatize("worst", pos="a")
```

Chunking

In [ ]:
```
lotr_quote = "It's a dangerous business, Frodo, going out your door."
```

In [ ]:
```
words_in_lotr_quote = word_tokenize(lotr_quote)
print(words_in_lotr_quote)
```

In [ ]:
```
lotr_pos_tags = nltk.pos_tag(words_in_lotr_quote)
print(lotr_pos_tags)
```

In [ ]:
```
grammar = "NP: {<DT>?<JJ>*<NN>}"
```

In [ ]:
```
chunk_parser = nltk.RegexpParser(grammar)
```

In [ ]:
```
tree = chunk_parser.parse(lotr_pos_tags)
display(tree)
```

Chinking

In [ ]:
```
grammar = r"""
Chunk: {<.*>+}
}<JJ>{
"""
```

In [ ]:
```
chunk_parser = nltk.RegexpParser(grammar)
```

In [ ]:
```
tree = chunk_parser.parse(lotr_pos_tags)
display(tree)
```

Using Named Entity Recognition (NER)

In [ ]:
```
tree = nltk.ne_chunk(lotr_pos_tags)
display(tree)
```

In [ ]:
```
tree = nltk.ne_chunk(lotr_pos_tags, binary=True)
display(tree)
```

In [ ]:
```
>>> quote = """
Men like Schiaparelli watched the red planet—it is odd, by-the-bye, that
for countless centuries Mars has been the star of war—but failed to
All that time the Martians must have been getting ready.

During the opposition of 1894 a great light was seen on the illuminated
```

```
part of the disk, first at the Lick Observatory, then by Perrotin of Nice,
and then by other observers. English readers heard of it first in the
issue of Nature dated August 2.
"""

quote = quote.strip("\n").replace("\n", " ")
quote
```

In [ ]:

```
def extract_ne(quote, language="english"):
    words = word_tokenize(quote, language)
    tags = nltk.pos_tag(words)
    tree = nltk.ne_chunk(tags, binary=True)

    return set(" ".join(i[0] for i in t) for t in tree if hasattr(t, "label") a
```

In [ ]:

```
extract_ne(quote)
```

Using a Concordance

In [ ]:

```
text8.concordance("man")
```

In [ ]:

```
text8.concordance("woman")
```

Making a Dispersion Plot

In [ ]:

```
text8.dispersion_plot(["woman", "lady", "girl", "gal", "man", "gentleman", "boy
```

In [ ]:

```
# Sense and Sensibility
text2.dispersion_plot(["Allenham", "Whitwell", "Cleveland", "Combe"])
```

Making a Frequency Distribution

In [ ]:

```
frequency_distribution = nltk.FreqDist(text8)
print(frequency_distribution)
```

In [ ]:

```
frequency_distribution.most_common(20)
```

In [ ]:

```
meaningful_words = [word for word in text8 if word.isalpha() and not word.casef
```

In [ ]:

```
frequency_distribution = nltk.FreqDist(meaningful_words)
print(frequency_distribution)
```

In [ ]:

```
frequency_distribution.most_common(20)
```

In [ ]:

```
frequency_distribution.plot(20, cumulative=True)
```

Finding Collocations

In [ ]:

```
text8.collocations()
```

In [ ]:

```
lemmatizer = WordNetLemmatizer()
```

In [ ]:

```
lemmatized_words = [lemmatizer.lemmatize(word) for word in text8]
```

In [ ]:

```
new_text = nltk.Text(lemmatized_words)
```

In [ ]:

```
new_text.collocations()
```

In [ ]:

In [ ]:

Sentiment Analysis: First Steps With Python's NLTK Library

Sentiment Analysis: First Steps With Python's NLTK Library

Creating Frequency Distributions

In [ ]:

```
text = """
For some quick analysis, creating a corpus could be overkill.
If all you need is a word list,
there are simpler ways to achieve that goal.
"""

pprint(nltk.word_tokenize(text), width=79, compact=True)
```

In [ ]:

```
text: list[str] = nltk.word_tokenize(text)
fd = nltk.FreqDist(text)
```

In [ ]:

```
fd.most_common(3)
```

In [ ]:

```
lower_fd.tabulate(3)
```

Extracting Concordance and Collocations

In [ ]:

```
text = nltk.Text(nltk.corpus.state_union.words())
text.concordance("america", lines=5)
```

In [ ]:

```
concordance_list = text.concordance_list("america", lines=2)
for entry in concordance_list:
    print(entry.line)
```

`.vocab()` is essentially a shortcut to create a frequency distribution from an instance of `nltk.Text`. That way, you don't have to make a separate call to instantiate a new `nltk.FreqDist` object.

In [ ]:
```
text: list[str] = nltk.word_tokenize("""
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
""")

text = nltk.Text(text)
fd = text.vocab()  # Equivalent to fd = nltk.FreqDist(words)
fd.tabulate(3)
```

Another powerful feature of NLTK is its ability to quickly find **collocations** with simple function calls. Collocations are series of words that frequently appear together in a given text. In the State of the Union corpus, for example, you'd expect to find the words United and States appearing next to each other very often. Those two words appearing together is a collocation.

Collocations can be made up of two or more words. NLTK provides classes to handle several types of collocations:

- **Bigrams**: Frequent two-word combinations
- **Trigrams**: Frequent three-word combinations
- **Quadgrams**: Frequent four-word combinations

Using NLTK's Pre-Trained Sentiment Analyzer

In [5]:
```
sia = SentimentIntensityAnalyzer()
```

In [6]:
```
sia.polarity_scores("Wow, NLTK is really powerful!")
```

Out[6]:
```
{'neg': 0.0, 'neu': 0.295, 'pos': 0.705, 'compound': 0.8012}
```

In [7]:
```
sia.polarity_scores("It was ok")
```

Out[7]:
```
{'neg': 0.0, 'neu': 0.476, 'pos': 0.524, 'compound': 0.296}
```

In [8]:
```
sia.polarity_scores("well duh")
```

Out[8]:
```
{'neg': 0.0, 'neu': 0.323, 'pos': 0.677, 'compound': 0.2732}
```

In [9]:
```
sia.polarity_scores("Drew is a rat!")
```

Out[9]:
```
{'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound': 0.0}
```

In [10]:
```
sia.polarity_scores("Drew is a snitch!")
```

Out[10]:
```
{'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound': 0.0}
```

In [11]:
```
sia.polarity_scores("Drew is a opp!")
```

Out[11]:

```
{'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound': 0.0}
```

Tweets

In [12]:

```
tweets = [t.replace("://", "//") for t in nltk.corpus.twitter_samples.strings()
```

In [13]:

```
def is_positive_tweet(tweet: str) -> bool:
    """True if tweet has positive compound sentiment, False otherwise."""
    return sia.polarity_scores(tweet)["compound"] > 0

shuffle(tweets)
for tweet in tweets[:10]:
    print(">", is_positive_tweet(tweet), tweet)
```

```
> False RT @StewartHosieSNP: @theSNP want to lock the Tories out of power. Why
is Miliband threatening to allow Cameron back into Downing Street?  …
> True RT @AndyJakeryancov: @Nigel_Farage I'm voting UKIP. Unbelievable that p
eople will vote for either lab or tor. Same old c**p every 5 years. …
> False Can't believe I was too ill to go to work today. Wish I was there :(
> True RT @ronwindward: @JimForScotland Crazy statements like that just confir
ms to me I made the right decision in leaving Labour for the SNP.Tim…
> False RT @The45Storm: @ANG_B49 I think she knows the deal has already been a
lready been done between Labour &amp; Tory
> True @rubysnipples whats phoebe's name on shots? can't find her :(
> True I think #BBCNewsnight Ms Stretton was clearly watching a different deba
te to the majority of the population: Clegg 'least rememberable?' Ok.
> False RT @BenjaminWillsSJ: Ed Milliband rocking out the white guy on a dance
floor moves https//t.co/UvtcYjDHo0
> True RT @carrieapples: Even left-wing New Statesman says David Cameron did b
est tonight #BBCQT https//t.co/e6ljgfecek
> False RT @guardiannews: Guardian front page, Friday 1 May 2015: Miliband har
dens his line: I will not do deal with SNP http//t.co/T5josh3wNc
```

Movie Reviews

In [14]:

```
positive_review_ids = nltk.corpus.movie_reviews.fileids(categories=["pos"])
negative_review_ids = nltk.corpus.movie_reviews.fileids(categories=["neg"])
all_review_ids = positive_review_ids + negative_review_ids
```

In [15]:

```
def is_positive_review(review_id: str) -> bool:
    """True if the average of all sentence compound scores is positive."""
    text = nltk.corpus.movie_reviews.raw(review_id)
    scores = [
        sia.polarity_scores(sentence)["compound"]
        for sentence in nltk.sent_tokenize(text)
    ]
    return mean(scores) > 0
```

In [16]:

```
shuffle(all_review_ids)
correct = 0
for review_id in all_review_ids:
    if is_positive_review(review_id):
        if review_id in positive_review_ids:
            correct += 1
    else:
        if review_id in negative_review_ids:
            correct += 1

print(F"{correct / len(all_review_ids):.2%} correct")
```

```
64.00% correct
```

Customizing NLTK’s Sentiment Analysis

In [17]:

```
unwanted = nltk.corpus.stopwords.words("english")
unwanted.extend([w.lower() for w in nltk.corpus.names.words()])
```

In [18]:

```
def skip_unwanted(pos_tuple):
    word, tag = pos_tuple
    if not word.isalpha() or word in unwanted:
        return False
    if tag.startswith("NN"):
        return False
    return True

positive_words = [word for word, tag in filter(skip_unwanted, nltk.pos_tag(nltk
negative_words = [word for word, tag in filter(skip_unwanted, nltk.pos_tag(nltk
```

In [19]:

```
positive_fd = nltk.FreqDist(positive_words)
negative_fd = nltk.FreqDist(negative_words)

common_set = set(positive_fd).intersection(negative_fd)
len(common_set)
```

Out[19]:

```
9511
```

In [20]:

```
for word in common_set:
    del positive_fd[word]
    del negative_fd[word]

top_100_positive = {word for word, count in positive_fd.most_common(100)}
top_100_negative = {word for word, count in negative_fd.most_common(100)}
```

In [21]:

```
print(top_100_positive)
```

```
{'shanghai', 'deftly', 'belgian', 'monetary', 'criticized', 'superficially',
'biased', 'sparks', 'lovingly', 'addresses', 'falter', 'rico', 'freed', 'organ
izing', 'galactic', 'conveys', 'methodical', 'ghost', 'legally', 'pink', 'apos
tle', 'broadcast', 'watson', 'balancing', 'melancholy', 'uncompromising', 'rad
io', 'textured', 'kimble', 'narrates', 'masterfully', 'indistinguishable', 'so
viet', 'flynt', 'maximus', 'amistad', 'argento', 'safely', 'trimmed', 'nello',
'brisk', 'unnerving', 'vertical', 'sobbing', 'profile', 'en', 'deft', 'vividl
y', 'danish', 'understatement', 'weir', 'pun', 'forceful', 'mulan', 'elegantl
y', 'lumumba', 'seahaven', 'notoriously', 'unquestionably', 'shrek', 'horned',
'unzipped', 'tale', 'supreme', 'ordell', 'valjean', 'curdled', 'benefit', 'kud
os', 'motta', 'attentive', 'matches', 'tibbs', 'audacious', 'redefines', 'hank
s', 'niccol', 'tibetan', 'farquaad', 'spacey', 'donkey', 'ulee', 'powerfully',
'unrestrained', 'perceived', 'stendhal', 'funnest', 'embeth', 'claiborne', 'je
di', 'taxing', 'fei', 'exhilarating', 'unassuming', 'uncut', 'sweetback', 'soc
ietal', 'weaves', 'fa', 'propelled'}
```

In [22]:

```
print(top_100_negative)
```

```
{'performances', 'chi', 'busted', 'undercut', 'godzilla', 'precinct', 'termina
l', 'mandingo', 'ordering', 'segal', 'sans', 'topless', 'heckerling', 'myster
y', 'traced', 'supergirl', 'unentertaining', 'rabid', 'joely', 'artemus', 'jer
icho', 'stupidest', 'grunting', 'sphere', 'disguise', 'degenerates', 'flippe
d', 'verhoven', 'comment', 'amish', 'droppingly', 'interspersed', 'deems', 'st
inks', 'embarassing', 'warranted', 'snipes', 'schumacher', 'horrid', 'nitro',
'flubber', 'digested', 'chuckled', 'brenner', 'popped', 'squabble', 'monumenta
lly', 'tectonic', 'battlefield', 'lamest', 'favors', 'wcw', 'harlem', 'incoher
ent', 'spawn', 'fetch', 'negated', 'virus', 'crucible', 'glancing', 'autisti
c', 'sneering', 'stupidly', 'weighed', 'tearing', 'undeveloped', 'patheticall
y', 'enticing', 'gordy', 'consecutive', 'modeled', 'unhealthy', 'plodding', 's
talks', 'iii', 'goo', 'babe', 'psychlo', 'mumbo', 'peripheral', 'forgetful',
'ego', 'club', 'pad', 'leaden', 'potty', 'tediously', 'bean', 'nbsp', 'wisecra
cking', 'rambo', 'rotating', 'injury', 'abysmal', 'audible', 'brazilian', 'leg
uizamo', 'manchurian', 'putrid', 'geronimo'}
```

Positive and negative bigram finders

In [23]:

```python
unwanted = nltk.corpus.stopwords.words("english")
unwanted.extend([w.lower() for w in nltk.corpus.names.words()])

positive_bigram_finder = nltk.collocations.BigramCollocationFinder.from_words([
    w for w in nltk.corpus.movie_reviews.words(categories=["pos"])
    if w.isalpha() and w not in unwanted
])

negative_bigram_finder = nltk.collocations.BigramCollocationFinder.from_words([
    w for w in nltk.corpus.movie_reviews.words(categories=["neg"])
    if w.isalpha() and w not in unwanted
])
```

In [24]:

```python
positive_bigram_finder.ngram_fd.tabulate(5)
```

```
('special', 'effects')        ('new', 'york')     ('even', 'though')
('one', 'best')        ('year', 'old')
                   179                    131                    120
117                    106
```

In [25]:

```python
negative_bigram_finder.ngram_fd.tabulate(5)
```

```
('special', 'effects')        ('new', 'york')     ('even', 'though')     ('hig
h', 'school')      ('looks', 'like')
                   208                    118                    102
99                     92
```

In [ ]:

Training and Using a Classifier

In [26]:

```python
def extract_features(text):
    features = dict()
    wordcount = 0
    compound_scores = list()
    positive_scores = list()

    for sentence in nltk.sent_tokenize(text):
        for word in nltk.word_tokenize(sentence):
            if word.lower() in top_100_positive:
                wordcount += 1
        compound_scores.append(sia.polarity_scores(sentence)["compound"])
        positive_scores.append(sia.polarity_scores(sentence)["pos"])

    # Adding 1 to the final compound score to always have positive numbers
    # since some classifiers you'll use later don't work with negative numbers.
    features["mean_compound"] = mean(compound_scores) + 1
    features["mean_positive"] = mean(positive_scores)
    features["wordcount"] = wordcount

    return features
```

In [27]:

```
features = [
    (extract_features(nltk.corpus.movie_reviews.raw(review)), "pos")
    for review in nltk.corpus.movie_reviews.fileids(categories=["pos"])
]
features.extend([
    (extract_features(nltk.corpus.movie_reviews.raw(review)), "neg")
    for review in nltk.corpus.movie_reviews.fileids(categories=["neg"])
])
```

In [28]:

```
JSON(features[:10])
```

Out[28]:

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

In [29]:

```
# Use 1/4 of the set for training
train_count = len(features) // 4
shuffle(features)
classifier = nltk.NaiveBayesClassifier.train(features[:train_count])
classifier.show_most_informative_features(10)
```

```
Most Informative Features
               wordcount = 4                 pos : neg    =      5.7 : 1.0
               wordcount = 3                 pos : neg    =      5.0 : 1.0
               wordcount = 2                 pos : neg    =      4.4 : 1.0
               wordcount = 0                 neg : pos    =      1.6 : 1.0
               wordcount = 1                 pos : neg    =      1.3 : 1.0
           mean_positive = 0.09154545454545454    pos : neg    =      1.0 : 1.
0
           mean_positive = 0.162             pos : neg    =      1.0 : 1.0
```

In [30]:

```
nltk.classify.accuracy(classifier, features[train_count:])
```

Out[30]:

```
0.668
```

In [31]:

```
new_review = """
Movie Review/'The Little Mermaid'
BY BOB GARVER

Back in 1989, the animated version of “The Little Mermaid” ushered in what came

Now in 2023, the company is looking to a live-action version of “The Little Mer

The pandemic forced “Soul,” “Luca,” and “Turning Red” to go directly to streami

The best performer since 2019 was last year's critical flop “Lightyear” with $1
four-day Memorial Day weekend.
```

```
The story, as before, is that mermaid princess Ariel (Halle Bailey) wants to le

A falling-out between father and daughter sends Ariel right into the tentacles

She sets out on the adventure of a lifetime on land, aided by Sebastian and her

The good news is that the musical numbers fans love are well-translated here wi

Also, the cinematography is beautiful with luscious blues and greens (sadly not

The bad news is that the film goes for some additions that don’t work. The new

Eric is given a parallel storyline similar to Ariel’s, which does add some much

It all balances out to a pretty good movie, perhaps the best of Disney’s live-a

Grade: B-
“The Little Mermaid” is rated PG for action/peril and some scary images. Its ru
"""

new_review = new_review.strip("\n").replace("\n\n", " ").replace("\n", " ")
```

In [32]:

```
featureset = {"raw": new_review}
classifier.classify(featureset)
```

Out[32]:

```
'neg'
```

In [33]:

```
extract_features(new_review)
```

Out[33]:

```
{'mean_compound': 1.164552, 'mean_positive': 0.1244, 'wordcount': 0}
```

Comparing Additional Classifiers

In [34]:

```
classifiers = {
    "BernoulliNB": BernoulliNB(),
    "ComplementNB": ComplementNB(),
    "MultinomialNB": MultinomialNB(),
    "KNeighborsClassifier": KNeighborsClassifier(),
    "DecisionTreeClassifier": DecisionTreeClassifier(),
    "RandomForestClassifier": RandomForestClassifier(),
    "LogisticRegression": LogisticRegression(),
    "MLPClassifier": MLPClassifier(max_iter=1000),
    "AdaBoostClassifier": AdaBoostClassifier(),
}
```

In [36]:

```
# Use 1/4 of the set for training
train_count = len(features) // 4
shuffle(features)
for name, sklearn_classifier in classifiers.items():
    classifier = nltk.classify.SklearnClassifier(sklearn_classifier)
    classifier.train(features[:train_count])
    accuracy = nltk.classify.accuracy(classifier, features[train_count:])
    print(F"{accuracy:.2%} - {name}")
```

Out[36]:

```
<SklearnClassifier(BernoulliNB())>
```

```
           66.33% - BernoulliNB
           <SklearnClassifier(ComplementNB())>
Out[36]:
           66.33% - ComplementNB
           <SklearnClassifier(MultinomialNB())>
Out[36]:
           66.20% - MultinomialNB
           <SklearnClassifier(KNeighborsClassifier())>
Out[36]:
           69.87% - KNeighborsClassifier
           <SklearnClassifier(DecisionTreeClassifier())>
Out[36]:
           63.40% - DecisionTreeClassifier
           <SklearnClassifier(RandomForestClassifier())>
Out[36]:
           69.27% - RandomForestClassifier
           <SklearnClassifier(LogisticRegression())>
Out[36]:
           71.13% - LogisticRegression
           <SklearnClassifier(MLPClassifier(max_iter=1000))>
Out[36]:
           72.87% - MLPClassifier
           <SklearnClassifier(AdaBoostClassifier())>
Out[36]:
           69.67% - AdaBoostClassifier
```

State of Union

In [37]:
```
stop_words = stopwords.words("english")
```

In [38]:
```
words = [w for w in nltk.corpus.state_union.words() if w.isalpha() and not w.ca
```

Frequency Distributions

In [ ]:
```
fd = nltk.FreqDist(words)
```

In [ ]:
```
fd.most_common(10)
```

In [ ]:
```
fd.tabulate(10)
```

In [ ]:
```
fd["America"]
fd["america"]
fd["AMERICA"]
```

In [ ]:
```
lower_fd = nltk.FreqDist([w.lower() for w in fd])
```

In [ ]:
```
lower_fd.tabulate(10)
```

Concordance and Collocations

Bigrams

In [ ]:
```
bigram_finder = nltk.collocations.BigramCollocationFinder.from_words(words)
```

In [ ]:
```
bigram_finder.ngram_fd.most_common(5)
```

In [ ]:
```
bigram_finder.ngram_fd.tabulate(5)
```

Trigrams

In [ ]:
```
trigrams_finder = nltk.collocations.TrigramCollocationFinder.from_words(words)
```

In [ ]:
```
trigrams_finder.ngram_fd.most_common(5)
```

In [ ]:
```
trigrams_finder.ngram_fd.tabulate(5)
```

Quadgrams

In [ ]:
```
quadgram_finder = nltk.collocations.QuadgramCollocationFinder.from_words(words)
```

In [ ]:
```
quadgram_finder.ngram_fd.most_common(5)
```

In [ ]:
```
quadgram_finder.ngram_fd.tabulate(3)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Practical Text Classification With Python and Keras

Practical Text Classification With Python and Keras

In [ ]: