NLP (Natural Language Processing) with Python

Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data.

In this article, we will discuss a higher-level overview of the basics of Natural Language Processing, which basically consists of combining machine learning techniques with text, and using math and statistics to get that text in a format that the machine learning algorithms can understand!

Agenda

1. Representing text as numerical data
2. Reading a text-based dataset into pandas
3. Vectorizing our dataset
4. Building and evaluating a model
5. Comparing models
6. Examining a model for further insight
7. Practicing this workflow on another dataset
8. Tuning the vectorizer (discussion)

Notebook Goals

> In this notebook we will discuss a higher level overview of the basics of Natural Language Processing, which basically consists of combining machine learning techniques with text, and using math and statistics to get that text in a format that the machine learning algorithms can understand!

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
sns.set_style("whitegrid")
plt.style.use("fivethirtyeight")
```

[] Representing text as numerical data

```
# example text for model training (SMS messages)
simple_train = ['call you tonight', 'Call me a cab', 'Please call
me... PLEASE!']
```

[] From the scikit-learn documentation:

> Text Analysis is a major application field for machine learning algorithms. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect **numerical feature vectors with a fixed size** rather than the **raw text documents with variable length**.

We will use CountVectorizer to "convert text into a matrix of token counts":

```
# import and instantiate CountVectorizer (with the default parameters)
from sklearn.feature_extraction.text import CountVectorizer

vect = CountVectorizer()

# learn the 'vocabulary' of the training data (occurs in-place)
vect.fit(simple_train)

# examine the fitted vocabulary
vect.get_feature_names_out()
```

```
array(['cab', 'call', 'me', 'please', 'tonight', 'you'], dtype=object)
```

```
# transform training data into a 'document-term matrix'
simple_train_dtm = vect.transform(simple_train)
simple_train_dtm
```

```
<3x6 sparse matrix of type '<class 'numpy.int64'>'
	with 9 stored elements in Compressed Sparse Row format>
```

```
# convert sparse matrix to a dense matrix
simple_train_dtm.toarray()
```

```
array([[0, 1, 0, 0, 1, 1],
       [1, 1, 1, 0, 0, 0],
       [0, 1, 1, 2, 0, 0]])
```

```
# examine the vocabulary and document-term matrix together
pd.DataFrame(simple_train_dtm.toarray(),
columns=vect.get_feature_names())
```

```
/opt/conda/lib/python3.7/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)
```

```
   cab  call  me  please  tonight  you
0    0     1   0       0        1    1
1    1     1   1       0        0    0
2    0     1   1       2        0    0
```

 From the [scikit-learn documentation](#):

> In this scheme, features and samples are defined as follows:
>
> - Each individual token occurrence frequency (normalized or not) is treated as a **feature**.
> - The vector of all the token frequencies for a given document is considered a multivariate **sample**.
>
> A **corpus of documents** can thus be represented by a matrix with **one row per document** and **one column per token** (e.g. word) occurring in the corpus.
>
> We call **vectorization** the general process of turning a collection of text documents into numerical feature vectors. This specific strategy (tokenization, counting and normalization) is called the **Bag of Words** or "Bag of n-grams" representation. Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

```python
# check the type of the document-term matrix
print(type(simple_train_dtm))

# examine the sparse matrix contents
print(simple_train_dtm)
```

```
<class 'scipy.sparse.csr.csr_matrix'>
  (0, 1)	1
  (0, 4)	1
  (0, 5)	1
  (1, 0)	1
  (1, 1)	1
  (1, 2)	1
  (2, 1)	1
  (2, 2)	1
  (2, 3)	2
```

 From the [scikit-learn documentation](#):

> As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have **many feature values that are zeros** (typically more than 99% of them).
>
> For instance, a collection of 10,000 short text documents (such as emails) will use a vocabulary with a size in the order of 100,000 unique words in total while each document will use 100 to 1000 unique words individually.
>
> In order to be able to **store such a matrix in memory** but also to **speed up operations**, implementations will typically use a **sparse representation** such as the implementations available in the `scipy.sparse` package.

```
# example text for model testing
simple_test = ["please don't call me"]
```

In order to **make a prediction**, the new observation must have the **same features as the training observations**, both in number and meaning.

```
# transform testing data into a document-term matrix (using existing vocabulary)
simple_test_dtm = vect.transform(simple_test)
simple_test_dtm.toarray()
```

```
array([[0, 1, 1, 1, 0, 0]])
```

```
# examine the vocabulary and document-term matrix together
pd.DataFrame(simple_test_dtm.toarray(),
columns=vect.get_feature_names_out())
```

	cab	call	me	please	tonight	you
0	0	1	1	1	0	0

Summary:

- `vect.fit(train)` **learns the vocabulary** of the training data
- `vect.transform(train)` uses the **fitted vocabulary** to build a document-term matrix from the training data
- `vect.transform(test)` uses the **fitted vocabulary** to build a document-term matrix from the testing data (and **ignores tokens** it hasn't seen before)

Reading a text-based dataset into pandas

```
# read file into pandas using a relative path
sms =
pd.read_csv("/kaggle/input/sms-spam-collection-dataset/spam.csv",
encoding='latin-1')
sms.dropna(how="any", inplace=True, axis=1)
sms.columns = ['label', 'message']

sms.head()
```

	label	message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

[] Exploratory Data Analysis (EDA)

```
sms.describe()
```

```
       label                   message
count   5572                      5572
unique     2                      5169
top      ham  Sorry, I'll call later
freq    4825                        30
```

```
sms.groupby('label').describe()
```

```
      message

        count unique
top freq
label

ham      4825   4516                             Sorry, I'll call
later   30
spam      747    653  Please call our customer service
representativ...    4
```

We have 4825 ham message and 747 spam message

```
# convert label to a numerical variable
sms['label_num'] = sms.label.map({'ham':0, 'spam':1})
sms.head()
```

```
  label                                            message  label_num
0   ham  Go until jurong point, crazy.. Available only ...          0
1   ham                      Ok lar... Joking wif u oni...          0
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...          1
3   ham  U dun say so early hor... U c already then say...          0
4   ham  Nah I don't think he goes to usf, he lives aro...          0
```

As we continue our analysis we want to start thinking about the features we are going to be using. This goes along with the general idea of feature engineering. The better your domain knowledge on the data, the better your ability to engineer more features from it. Feature engineering is a very large part of spam detection in general.

```
sms['message_len'] = sms.message.apply(len)
sms.head()
```

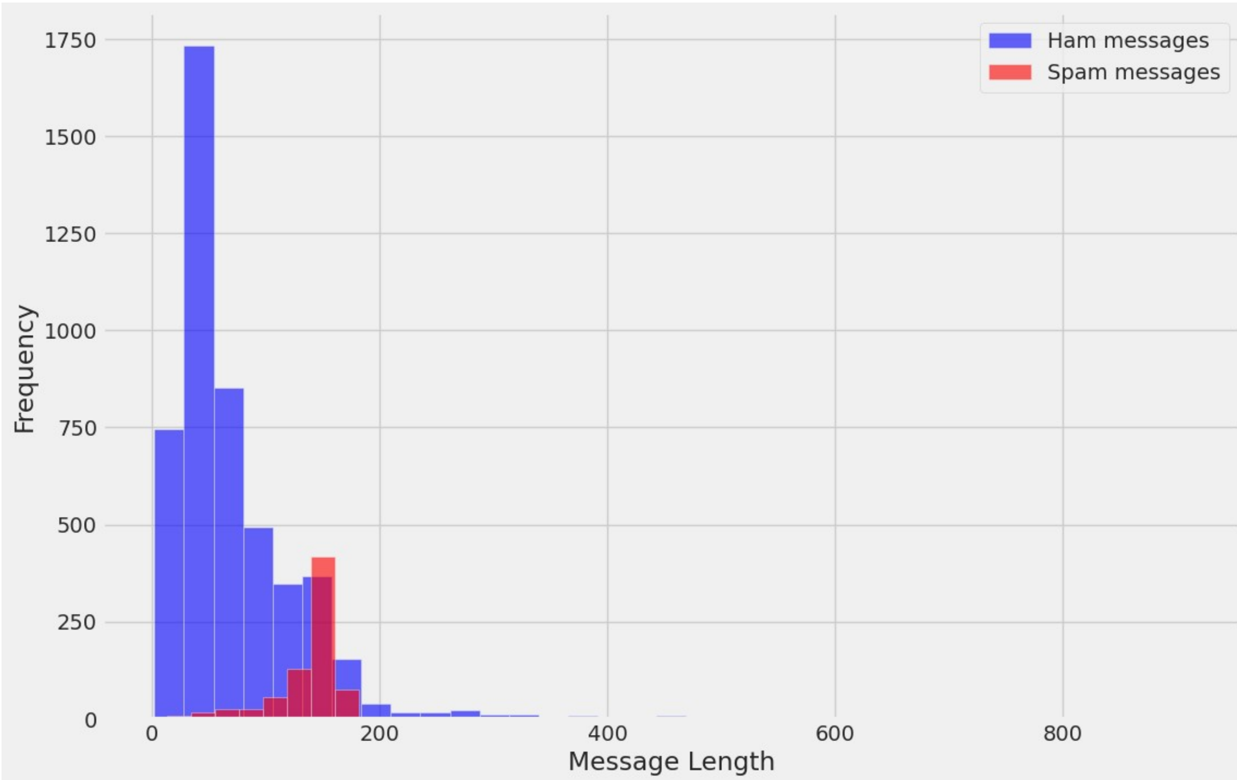
```
  label                                            message  label_num
\
0   ham  Go until jurong point, crazy.. Available only ...          0

1   ham                      Ok lar... Joking wif u oni...          0
```

```
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...          1

3   ham  U dun say so early hor... U c already then say...          0

4   ham  Nah I don't think he goes to usf, he lives aro...          0
```

```
   message_len
0          111
1           29
2          155
3           49
4           61
```

```
plt.figure(figsize=(12, 8))

sms[sms.label=='ham'].message_len.plot(bins=35, kind='hist',
color='blue',
                                       label='Ham messages',
alpha=0.6)
sms[sms.label=='spam'].message_len.plot(kind='hist', color='red',
                                       label='Spam messages',
alpha=0.6)
plt.legend()
plt.xlabel("Message Length")
```

```
Text(0.5, 0, 'Message Length')
```



Very interesting! Through just basic EDA we've been able to discover a trend that spam messages tend to have more characters.

```
sms[sms.label=='ham'].describe()
```

	label_num	message_len
count	4825.0	4825.000000
mean	0.0	71.023627
std	0.0	58.016023
min	0.0	2.000000
25%	0.0	33.000000
50%	0.0	52.000000
75%	0.0	92.000000
max	0.0	910.000000

```
sms[sms.label=='spam'].describe()
```

	label_num	message_len
count	747.0	747.000000
mean	1.0	138.866131
std	0.0	29.183082
min	1.0	13.000000
25%	1.0	132.500000
50%	1.0	149.000000
75%	1.0	157.000000
max	1.0	224.000000

Woah! 910 characters, let's use masking to find this message:

```
sms[sms.message_len == 910].message.iloc[0]
```

```
"For me the love should start with attraction.i should feel that I
need her every time around me.she should be the first thing which
comes in my thoughts.I would start the day and end it with her.she
should be there every time I dream.love will be then when my every
breath has her name.my life should happen around her.my life will be
named to her.I would cry for her.will give all my happiness and take
all her sorrows.I will be ready to fight with anyone for her.I will be
in love when I will be doing the craziest things for her.love will be
when I don't have to proove anyone that my girl is the most beautiful
lady on the whole planet.I will always be singing praises for her.love
will be when I start up making chicken curry and end up makiing
sambar.life will be the most beautiful then.will get every morning and
thank god for the day because she is with me.I would like to say a
lot..will tell later.."
```

Text Pre-processing

Our main issue with our data is that it is all in text format (strings). The classification algorithms that we usally use need some sort of numerical feature vector in order to perform the classification task. There are actually many methods to convert a corpus to a vector format. The simplest is the `bag-of-words` approach, where each unique word in a text will be represented by one number.

In this section we'll convert the raw messages (sequence of characters) into vectors (sequences of numbers).

As a first step, let's write a function that will split a message into its individual words and return a list. We'll also remove very common words, ('the', 'a', etc..). To do this we will take advantage of the `NLTK` library. It's pretty much the standard library in Python for processing text and has a lot of useful features. We'll only use some of the basic ones here.

Let's create a function that will process the string in the message column, then we can just use **apply()** in pandas do process all the text in the DataFrame.

First removing punctuation. We can just take advantage of Python's built-in **string** library to get a quick list of all the possible punctuation:

```
import string
from nltk.corpus import stopwords

def text_process(mess):
    """
    Takes in a string of text, then performs the following:
    1. Remove all punctuation
    2. Remove all stopwords
    3. Returns a list of the cleaned text
```

```
    """
    STOPWORDS = stopwords.words('english') + ['u', 'ü', 'ur', '4',
'2', 'im', 'dont', 'doin', 'ure']
    # Check characters to see if they are in punctuation
    nopunc = [char for char in mess if char not in string.punctuation]

    # Join the characters again to form the string.
    nopunc = ''.join(nopunc)

    # Now just remove any stopwords
    return ' '.join([word for word in nopunc.split() if word.lower()
not in STOPWORDS])

sms.head()

  label                                            message  label_num
\
0   ham  Go until jurong point, crazy.. Available only ...          0

1   ham                      Ok lar... Joking wif u oni...          0

2  spam  Free entry in 2 a wkly comp to win FA Cup fina...          1

3   ham  U dun say so early hor... U c already then say...          0

4   ham  Nah I don't think he goes to usf, he lives aro...          0


   message_len
0          111
1           29
2          155
3           49
4           61
```

Now let's "tokenize" these messages. Tokenization is just the term used to describe the process of converting the normal text strings in to a list of tokens (words that we actually want).

```
sms['clean_msg'] = sms.message.apply(text_process)

sms.head()

  label                                            message  label_num
\
0   ham  Go until jurong point, crazy.. Available only ...          0

1   ham                      Ok lar... Joking wif u oni...          0

2  spam  Free entry in 2 a wkly comp to win FA Cup fina...          1
```

```
3   ham  U dun say so early hor... U c already then say...        0

4   ham  Nah I don't think he goes to usf, he lives aro...        0
```

```
   message_len                                           clean_msg
0          111  Go jurong point crazy Available bugis n great ...
1           29                               Ok lar Joking wif oni
2          155  Free entry wkly comp win FA Cup final tkts 21s...
3           49                       dun say early hor c already say
4           61                  Nah think goes usf lives around though
```

```python
type(stopwords.words('english'))
```

```
list
```

```python
from collections import Counter

words = sms[sms.label=='ham'].clean_msg.apply(lambda x: [word.lower()
for word in x.split()])
ham_words = Counter()

for msg in words:
    ham_words.update(msg)

print(ham_words.most_common(50))
```

```
[('get', 303), ('ltgt', 276), ('ok', 272), ('go', 247), ('ill', 236),
('know', 232), ('got', 231), ('like', 229), ('call', 229), ('come',
224), ('good', 222), ('time', 189), ('day', 187), ('love', 185),
('going', 167), ('want', 163), ('one', 162), ('home', 160), ('lor',
160), ('need', 156), ('sorry', 153), ('still', 146), ('see', 137),
('n', 134), ('later', 134), ('da', 131), ('r', 131), ('back', 129),
('think', 128), ('well', 126), ('today', 125), ('send', 123), ('tell',
121), ('cant', 118), ('ì', 117), ('hi', 117), ('take', 112), ('much',
112), ('oh', 111), ('night', 107), ('hey', 106), ('happy', 105),
('great', 100), ('way', 100), ('hope', 99), ('pls', 98), ('work', 96),
('wat', 95), ('thats', 94), ('dear', 94)]
```

```python
words = sms[sms.label=='spam'].clean_msg.apply(lambda x: [word.lower()
for word in x.split()])
spam_words = Counter()

for msg in words:
    spam_words.update(msg)

print(spam_words.most_common(50))
```

```
[('call', 347), ('free', 216), ('txt', 150), ('mobile', 123), ('text',
120), ('claim', 113), ('stop', 113), ('reply', 101), ('prize', 92),
('get', 83), ('new', 69), ('send', 67), ('nokia', 65), ('urgent', 63),
```

```
('cash', 62), ('win', 60), ('contact', 56), ('service', 55),
('please', 52), ('guaranteed', 50), ('customer', 49), ('16', 49),
('week', 49), ('tone', 48), ('per', 46), ('phone', 45), ('18', 43),
('chat', 42), ('awarded', 38), ('draw', 38), ('latest', 36),
('å£1000', 35), ('line', 35), ('150ppm', 34), ('mins', 34),
('receive', 33), ('camera', 33), ('1', 33), ('every', 33), ('message',
32), ('holiday', 32), ('landline', 32), ('shows', 31), ('å£2000', 31),
('go', 31), ('box', 30), ('number', 30), ('apply', 29), ('code', 29),
('live', 29)]
```

Vectorization

Currently, we have the messages as lists of tokens (also known as lemmas) and now we need to convert each of those messages into a vector the SciKit Learn's algorithm models can work with.

Now we'll convert each message, represented as a list of tokens (lemmas) above, into a vector that machine learning models can understand.

We'll do that in three steps using the bag-of-words model:

1. Count how many times does a word occur in each message (Known as term frequency)
2. Weigh the counts, so that frequent tokens get lower weight (inverse document frequency)
3. Normalize the vectors to unit length, to abstract from the original text length (L2 norm)

Let's begin the first step:

Each vector will have as many dimensions as there are unique words in the SMS corpus. We will first use SciKit Learn's **CountVectorizer**. This model will convert a collection of text documents to a matrix of token counts.

We can imagine this as a 2-Dimensional matrix. Where the 1-dimension is the entire vocabulary (1 row per word) and the other dimension are the actual documents, in this case a column per text message.

For example:

Since there are so many messages, we can expect a lot of zero counts for the presence of that word in that document. Because of this, SciKit Learn will output a Sparse Matrix.

```
# split X and y into training and testing sets
from sklearn.model_selection import train_test_split

# how to define X and y (from the SMS data) for use with
COUNTVECTORIZER
X = sms.clean_msg
y = sms.label_num
print(X.shape)
print(y.shape)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=1)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(5572,)
(5572,)
(4179,)
(1393,)
(4179,)
(1393,)
```

There are a lot of arguments and parameters that can be passed to the CountVectorizer. In this case we will just specify the **analyzer** to be our own previously defined function:

```
from sklearn.feature_extraction.text import CountVectorizer

# instantiate the vectorizer
vect = CountVectorizer()
vect.fit(X_train)

# learn training data vocabulary, then use it to create a document-
term matrix
X_train_dtm = vect.transform(X_train)

# equivalently: combine fit and transform into a single step
X_train_dtm = vect.fit_transform(X_train)


# examine the document-term matrix
print(type(X_train_dtm), X_train_dtm.shape)

# transform testing data (using fitted vocabulary) into a document-
term matrix
X_test_dtm = vect.transform(X_test)
print(type(X_test_dtm), X_test_dtm.shape)
```

```
<class 'scipy.sparse.csr.csr_matrix'> (4179, 7996)
<class 'scipy.sparse.csr.csr_matrix'> (1393, 7996)
```

```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer = TfidfTransformer()
tfidf_transformer.fit(X_train_dtm)
tfidf_transformer.transform(X_train_dtm)
```

```
<4179x7996 sparse matrix of type '<class 'numpy.float64'>'
	with 34796 stored elements in Compressed Sparse Row format>
```

🤓 Building and evaluating a model

We will use multinomial Naive Bayes:

The multinomial Naive Bayes classifier is suitable for classification with **discrete features** (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

```
# import and instantiate a Multinomial Naive Bayes model
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()

# train the model using X_train_dtm (timing it with an IPython "magic
command")
%time nb.fit(X_train_dtm, y_train)
```

```
CPU times: user 4.27 ms, sys: 41 µs, total: 4.31 ms
Wall time: 3.92 ms
```

```
MultinomialNB()
```

```
from sklearn import metrics

# make class predictions for X_test_dtm
y_pred_class = nb.predict(X_test_dtm)

# calculate accuracy of class predictions
print("=======Accuracy Score===========")
print(metrics.accuracy_score(y_test, y_pred_class))

# print the confusion matrix
print("=======Confision Matrix===========")
metrics.confusion_matrix(y_test, y_pred_class)
```

```
=======Accuracy Score===========
0.9827709978463748
=======Confision Matrix===========
```

```
array([[1205,    8],
       [  16,  164]])
```

```
# print message text for false positives (ham incorrectly classifier)
# X_test[(y_pred_class==1) & (y_test==0)]
X_test[y_pred_class > y_test]
```

```
2418    Madamregret disturbancemight receive reference...
4598                                 laid airtel line rest
386                                    Customer place call
1289    HeyGreat dealFarm tour 9am 5pm 95pax 50 deposi...
5094    Hi ShanilRakhesh herethanksi exchanged uncut d...
494                                       free nowcan call
759     Call youcarlos isare phones vibrate acting mig...
3140                                   Customer place call
Name: clean_msg, dtype: object
```

```
# print message text for false negatives (spam incorrectly classifier)
X_test[y_pred_class < y_test]
```

```
4674    Hi babe Chloe r smashed saturday night great w...
3528    Xmas New Years Eve tickets sale club day 10am ...
3417    LIFE never much fun great came made truly spec...
2773    come takes little time child afraid dark becom...
1960    Guess Somebody know secretly fancies Wanna fin...
5       FreeMsg Hey darling 3 weeks word back Id like ...
2078                             85233 FREERingtoneReply REAL
1457    CLAIRE havin borin time alone wanna cum 2nite ...
190     unique enough Find 30th August wwwareyouunique...
2429    Guess IThis first time created web page WWWASJ...
3057    unsubscribed services Get tons sexy babes hunk...
1021    Guess Somebody know secretly fancies Wanna fin...
4067    TBSPERSOLVO chasing us since Sept forå£38 defi...
3358         Sorry missed call lets talk time 07090201529
2821    ROMCAPspam Everyone around responding well pre...
2247    Back work 2morro half term C 2nite sexy passio...
Name: clean_msg, dtype: object
```

```
# example of false negative
X_test[4949]
```

```
'Hi probably much fun get message thought id txt cos bored james
farting night'
```

```
# calculate predicted probabilities for X_test_dtm (poorly calibrated)
y_pred_prob = nb.predict_proba(X_test_dtm)[:, 1]
y_pred_prob
```

```
array([2.11903975e-02, 3.97831612e-04, 1.06470895e-03, ...,
       1.31939653e-02, 9.99821127e-05, 6.04083365e-06])
```

```
# calculate AUC
metrics.roc_auc_score(y_test, y_pred_prob)
```

```
0.9774342768159751
```

```
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import Pipeline
```

```
pipe = Pipeline([('bow', CountVectorizer()),
                 ('tfid', TfidfTransformer()),
                 ('model', MultinomialNB())])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)

# calculate accuracy of class predictions
print("=======Accuracy Score===========")
print(metrics.accuracy_score(y_test, y_pred))

# print the confusion matrix
print("=======Confision Matrix===========")
metrics.confusion_matrix(y_test, y_pred)
```

```
=======Accuracy Score===========
0.9669777458722182
=======Confision Matrix===========

array([[1213,    0],
       [  46,  134]])
```

Comparing models

We will compare multinomial Naive Bayes with logistic regression:

> Logistic regression, despite its name, is a **linear model for classification** rather than regression. Logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

```
# import an instantiate a logistic regression model
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(solver='liblinear')

# train the model using X_train_dtm
%time logreg.fit(X_train_dtm, y_train)
```

```
CPU times: user 17.9 ms, sys: 112 µs, total: 18.1 ms
Wall time: 21 ms

LogisticRegression(solver='liblinear')
```

```
# make class predictions for X_test_dtm
y_pred_class = logreg.predict(X_test_dtm)
```

```python
# calculate predicted probabilities for X_test_dtm (well calibrated)
y_pred_prob = logreg.predict_proba(X_test_dtm)[:, 1]
y_pred_prob
```

```
array([0.01694418, 0.0152182 , 0.08261755, ..., 0.02198942,
0.00531726,
       0.00679188])
```

```python
# calculate accuracy of class predictions
print("=======Accuracy Score===========")
print(metrics.accuracy_score(y_test, y_pred_class))

# print the confusion matrix
print("=======Confision Matrix===========")
print(metrics.confusion_matrix(y_test, y_pred_class))

# calculate AUC
print("=======ROC AUC Score===========")
print(metrics.roc_auc_score(y_test, y_pred_prob))
```

```
=======Accuracy Score===========
0.9842067480258435
=======Confision Matrix===========
[[1213    0]
 [  22  158]]
=======ROC AUC Score===========
0.9835714940001832
```

Tuning the vectorizer

Thus far, we have been using the default parameters of CountVectorizer:

```python
# show default parameters for CountVectorizer
vect
```

```
CountVectorizer()
```

> However, the vectorizer is worth tuning, just like a model is worth tuning! Here are a few parameters that you might want to tune:

- **stop_words**: string {'english'}, list, or None (default)
- If 'english', a built-in stop word list for English is used.
- If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens.
- If None, no stop words will be used.

```python
# remove English stop words
vect = CountVectorizer(stop_words='english')
```

- □ **ngram_range**: tuple (min_n, max_n), default=(1, 1)
- The lower and upper boundary of the range of n-values for different n-grams to be extracted.
- All values of n such that min_n <= n <= max_n will be used.

```
# include 1-grams and 2-grams
vect = CountVectorizer(ngram_range=(1, 2))
```

- □ **max_df**: float in range [0.0, 1.0] or int, default=1.0
- When building the vocabulary, ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words).
- If float, the parameter represents a proportion of documents.
- If integer, the parameter represents an absolute count.

```
# ignore terms that appear in more than 50% of the documents
vect = CountVectorizer(max_df=0.5)
```

- □ **min_df**: float in range [0.0, 1.0] or int, default=1
- When building the vocabulary, ignore terms that have a document frequency strictly lower than the given threshold. (This value is also called "cut-off" in the literature.)
- If float, the parameter represents a proportion of documents.
- If integer, the parameter represents an absolute count.

```
# only keep terms that appear in at least 2 documents
vect = CountVectorizer(min_df=2)
```

- □ **Guidelines for tuning CountVectorizer**:
- Use your knowledge of the problem and the text, and your understanding of the tuning parameters, to help you decide what parameters to tune and how to tune them.
- Experiment, and let the data tell you the best approach!