Reference

https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/ (https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/)

Table of Contents

1. Introduction to NLP
2. Text Preprocessing
    - Noise Removal
    - Lexicon Normalization
        - Lemmatization
        - Stemming
    - Object Standardization
3. Text to Features (Feature Engineering on text data)
    - Syntactical Parsing
        - Dependency Grammar
        - Part of Speech Tagging
    - Entity Parsing
        - Phrase Detection
        - Named Entity Recognition
        - Topic Modelling

        - N-Grams
    - Statistical features
        - TF – IDF
        - Frequency / Density Features
        - Readability Features
    - Word Embeddings
4. Important tasks of NLP
    - Text Classification
    - Text Matching
        - Levenshtein Distance
        - Phonetic Matching
        - Flexible String Matching
    - Coreference Resolution
    - Other Problems
5. Important NLP libraries

Introduction

- Only 21% of the available data is present in structured form.
- Despite having high dimension data, the information present in it is not directly accessible unless it is processed (read and understood) manually or analyzed by an automated system.
- In order to produce significant and actionable insights from text data, it is important to get acquainted with the techniques and principles of Natural Language Processing (NLP).

Introduction to Natural Language Processing

- By utilizing NLP and its components, one can organize the massive chunks of text data, perform numerous automated tasks and solve a wide range of problems such as – automatic summarization, machine translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation etc.
- Some terms that are used in the article:
    - Tokenization – process of converting a text into tokens
    - Tokens – words or entities present in the text
    - Text object – a sentence or a phrase or a word or an article

Text Preprocessing

- Noise Removal
- Lexicon Normalization
- Object Standardization

Noise Removal

Any piece of text which is not relevant to the context of the data and the end-output can be specified as the noise.

For example – language stopwords (commonly used words of a language – is, am, the, of, in etc), URLs or links, social media entities (mentions, hashtags), punctuations and industry specific words. This step deals with removal of all types of noisy entities present in the text.

A general approach for noise removal is to prepare a dictionary of noisy entities, and iterate the text object by tokens (or by words), eliminating those tokens which are present in the noise dictionary.

```
In [1]: # Sample code to remove noisy words from a text
noise_list = ["is", "a", "this", "..."]
def _remove_noise(input_text):
    words = input_text.split()
    noise_free_words = [word for word in words if word not in noise_list]
    noise_free_text = " ".join(noise_free_words)
    return noise_free_text

_remove_noise("this is a sample text")
```

```
Out[1]: 'sample text'
```

Another approach is to use the regular expressions while dealing with special patterns of noise. We have explained regular expressions in detail in one of our previous article. Following python code removes a regex pattern from the input text:

```
In [2]: # Sample code to remove a regex pattern
import re

def _remove_regex(input_text, regex_pattern):
    urls = re.finditer(regex_pattern, input_text)
    for i in urls:
        input_text = re.sub(i.group().strip(), '', input_text)
    return input_text

regex_pattern = "#[\w]*"

_remove_regex("remove this #hashtag from analytics vidhya", regex_pattern)
```

```
Out[2]: 'remove this  from analytics vidhya'
```

Lexicon Normalization

Another type of textual noise is about the multiple representations exhibited by single word.

For example – “play”, “player”, “played”, “plays” and “playing” are the different variations of the word – “play”, Though they mean different but contextually all are similar. The step converts all the disparities of a word into their normalized form (also known as lemma). **Normalization is a pivotal step for feature engineering with text as it converts the high dimensional features (N different features) to the low dimensional space (1 feature), which is an ideal ask for any ML model.

The most common lexicon normalization practices are :

- Stemming: Stemming is a rudimentary rule-based process of stripping the suffixes (“ing”, “ly”, “es”, “s” etc) from a word.
- Lemmatization: Lemmatization, on the other hand, is an organized & step by step procedure of obtaining the root form of the word, it makes use of vocabulary (dictionary importance of words) and morphological analysis (word structure and grammar relations).

Below is the sample code that performs lemmatization and stemming using python’s popular library – NLTK.

```
In [10]: from nltk.stem.wordnet import WordNetLemmatizer
         lem = WordNetLemmatizer()

         from nltk.stem.porter import PorterStemmer
         stem = PorterStemmer()

         word = "multiplying"
         lem.lemmatize(word, "v")
```

```
Out[10]: 'multiply'
```

```
In [4]: stem.stem(word)
```

```
Out[4]: 'multipli'
```

Object Standardization

Text data often contains words or phrases which are not present in any standard lexical dictionaries. These pieces are not recognized by search engines and models.

Some of the examples are – acronyms, hashtags with attached words, and colloquial slangs. With the help of regular expressions and manually prepared data dictionaries, this type of noise can be fixed, the code below uses a dictionary lookup method to replace social media slangs from a text.

```
In [9]: lookup_dict = {'rt':'Retweet', 'dm':'direct message', "awsm" : "awesome", "luv" :"love"}
        def _lookup_words(input_text):
            words = input_text.split()
            new_words = []
            for word in words:
                if word.lower() in lookup_dict:
                    word = lookup_dict[word.lower()]
                new_words.append(word)
                new_text = " ".join(new_words)
                return new_text

        _lookup_words("RT this is a retweeted tweet by Shivam Bansal")
```

```
Out[9]: 'Retweet'
```

Output should be: "Retweet this is a retweeted tweet by Shivam Bansal". Fix it later.

Apart from three steps discussed so far, other types of text preprocessing includes encoding-decoding noise, grammar checker, and spelling correction etc. The detailed article about preprocessing and its methods is given in https://www.analyticsvidhya.com/blog/2014/11/text-data-cleaning-steps-python/ (https://www.analyticsvidhya.com/blog/2014/11/text-data-cleaning-steps-python/)

Text to Features: Feature Engineering on text data

Syntactic Parsing

Syntactical parsing involves the analysis of words in the sentence for grammar and their arrangement in a manner that shows the relationships among the words. Dependency Grammar and Part of Speech tags are the important attributes of text syntactics.

Dependency Trees – Sentences are composed of some words sewed together. The relationship among the words in a sentence is determined by the basic dependency grammar. Dependency grammar is a class of syntactic text analysis that deals with (labeled) asymmetrical binary relations between two lexical items (words). Every relation can be represented in the form of a triplet (relation, governor, dependent). For example: consider the sentence – “Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas.” The relationship among the words can be observed in the form of a tree representation as shown:

Statistical Features

Term Frequency – Inverse Document Frequency (TF – IDF).

see details in other notes.

In [17]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
obj = TfidfVectorizer()
corpus = ['This is sample document.', 'another random document.', 'third sample document text']
X = obj.fit_transform(corpus)
print (X)
```

```
  (0, 7)	0.5844829010200651
  (0, 2)	0.5844829010200651
  (0, 4)	0.444514311537431
  (0, 1)	0.34520501686496574
  (1, 1)	0.3853716274664007
  (1, 0)	0.652490884512534
  (1, 3)	0.652490884512534
  (2, 4)	0.444514311537431
  (2, 1)	0.34520501686496574
  (2, 6)	0.5844829010200651
  (2, 5)	0.5844829010200651
```

The model creates a vocabulary dictionary and assigns an index to each word. Each row in the output contains a tuple (i,j) and a tf-idf value of word **at index j in document i.**

Count / Density / Readability Features

Count or Density based features can also be used in models and analysis. These features might seem trivial but shows a great impact in learning models. Some of the features are: Word Count, Sentence Count, Punctuation Counts and Industry specific word counts. Other types of measures include readability measures such as syllable counts, smog index and flesch reading ease. Refer to Textstat library to create such features. https://github.com/shivam5992/textstat (https://github.com/shivam5992/textstat)

Word Embedding (text vectors)

See other notes under the same folder for more details on this topic.
Word embedding is the modern way of representing words as vectors. The aim of word embedding is to redefine the high dimensional word features into low dimensional feature vectors by preserving the contextual similarity in the corpus. They are widely used in deep learning models such as Convolutional Neural Networks and Recurrent Neural Networks.

Word2Vec and GloVe are the two popular models to create word embedding of a text. These models takes a text corpus as input and produces the word vectors as output.

Word2Vec model is composed of preprocessing module, a shallow neural network model called Continuous Bag of Words and another shallow neural network model called skip-gram. These models are widely used for all other nlp problems. It first constructs a vocabulary from the training corpus and then learns word embedding representations. Following code using gensim package prepares the word embedding as the vectors.

In [ ]:
```
from gensim.models import Word2Vec
sentences = [['data', 'science'], ['vidhya', 'science', 'data', 'analytics'],['machine', 'learning'], ['deep', '

# train the model on your corpus
model = Word2Vec(sentences, min_count = 1)

# print (model.similarity('data', 'science'))
print (model.wv.similarity('data', 'science'))
# >>> 0.11222489293

print (model['learning'])
# >>> array([ 0.00459356  0.00303564 -0.00467622  0.00209638, ...])
```

They can be used as feature vectors for ML model, used to measure text similarity using cosine similarity techniques, words clustering and text classification techniques.

Important tasks of NLP

Text Classification

Text classification is one of the classical problem of NLP. Notorious examples include – **Email Spam Identification, topic classification of news, sentiment classification and organization of web pages by search engines**. Here is a code that uses naive bayes classifier using text blob library (built on top of nltk).

In [18]:
```
from textblob.classifiers import NaiveBayesClassifier as NBC
from textblob import TextBlob
training_corpus = [
                   ('I am exhausted of this work.', 'Class_B'),
                   ("I can't cooperate with this", 'Class_B'),
                   ('He is my badest enemy!', 'Class_B'),
                   ('My management is poor.', 'Class_B'),
                   ('I love this burger.', 'Class_A'),
                   ('This is an brilliant place!', 'Class_A'),
                   ('I feel very good about these dates.', 'Class_A'),
                   ('This is my best work.', 'Class_A'),
                   ("What an awesome view", 'Class_A'),
                   ('I do not like this dish', 'Class_B')]
test_corpus = [
                ("I am not feeling well today.", 'Class_B'),
                ("I feel brilliant!", 'Class_A'),
                ('Gary is a friend of mine.', 'Class_A'),
                ("I can't believe I'm doing this.", 'Class_B'),
                ('The date was good.', 'Class_A'), ('I do not enjoy my job', 'Class_B')]

model = NBC(training_corpus)
print(model.classify("Their codes are amazing."))
print(model.classify("I don't like their computer."))
print(model.accuracy(test_corpus))
```

```
Class_A
Class_B
0.8333333333333334
```

In [ ]:
```
Scikit.Learn also provides a pipeline framework for text classification:
```

In [24]:
```python
# from sklearn.feature_extraction.text
from sklearn.metrics import classification_report

# from sklearn.metrics import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn import svm

# preparing data for SVM model (using the same training_corpus, test_corpus from naive bayes example)
train_data = []
train_labels = []
for row in training_corpus:
    train_data.append(row[0])
    train_labels.append(row[1])

test_data = []
test_labels = []
for row in test_corpus:
    test_data.append(row[0])
    test_labels.append(row[1])

# Create feature vectors
vectorizer = TfidfVectorizer(min_df=4, max_df=0.9)
# Train the feature vectors
train_vectors = vectorizer.fit_transform(train_data)
# Apply model on test data
test_vectors = vectorizer.transform(test_data)

# Perform classification with SVM, kernel=linear
model = svm.SVC(kernel='linear')
model.fit(train_vectors, train_labels)
prediction = model.predict(test_vectors)
# >>> ['Class_A' 'Class_A' 'Class_B' 'Class_B' 'Class_A' 'Class_A']

print (classification_report(test_labels, prediction))
```

```
             precision    recall  f1-score   support

    Class_A       0.50      0.67      0.57         3
    Class_B       0.50      0.33      0.40         3
```

```
   micro avg       0.50      0.50      0.50         6
   macro avg       0.50      0.50      0.49         6
weighted avg       0.50      0.50      0.49         6
```

Text Matching / Similarity

One of the important areas of NLP is the matching of text objects to find similarities. Important applications of text matching includes **automatic spelling correction, data de-duplication and genome analysis etc.**

Levenshtein Distance

The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character. Following is the implementation for efficient memory computations.

In [25]:
```
def levenshtein(s1,s2):
    if len(s1) > len(s2):
        s1,s2 = s2,s1
    distances = range(len(s1) + 1)
    for index2,char2 in enumerate(s2):
        newDistances = [index2+1]
        for index1,char1 in enumerate(s1):
            if char1 == char2:
                newDistances.append(distances[index1])
            else:
                 newDistances.append(1 + min((distances[index1], distances[index1+1], newDistances[-1])))
        distances = newDistances
    return distances[-1]

print(levenshtein("analyze","analyse"))
```

```
1
```

Phonetic Matching

A Phonetic matching algorithm takes a keyword as input (person’s name, location name etc) and produces a character string that identifies a set of words that are (roughly) phonetically similar. It is very useful for searching large text corpuses, correcting spelling errors and matching relevant names. Soundex and Metaphone are two main phonetic algorithms used for this purpose. Python’s module Fuzzy is used to compute soundex strings for different words, for example –

```
In [ ]: import  fuzz
        soundex = fuzzy.Soundex(4)
        print(soundex('ankit'))
        # >>> “A523”
        print(soundex('aunkit'))
        # >>> “A523”
```

Flexible String Matching

A complete text matching system includes different algorithms pipelined together to compute variety of text variations. Regular expressions are really helpful for this purposes as well. Another common techniques include – exact string matching, lemmatized matching, and compact matching (takes care of spaces, punctuation’s, slangs etc).

Cosine Similarity

When the text is represented as vector notation, a general cosine similarity can also be applied in order to measure vectorized similarity. Following code converts a text to vectors (using term frequency) and applies cosine similarity to provide closeness among two text.

In [31]:
```
import math
from collections import Counter
def get_cosine(vec1, vec2):
    common = set(vec1.keys()) & set(vec2.keys())
    numerator = sum([vec1[x] * vec2[x] for x in common])

    sum1 = sum([vec1[x]**2 for x in vec1.keys()])
    sum2 = sum([vec2[x]**2 for x in vec2.keys()])
    denominator = math.sqrt(sum1) * math.sqrt(sum2)

    if not denominator:
        return 0.0
    else:
        return float(numerator) / denominator

def text_to_vector(text):
    words = text.split()
    return Counter(words)

text1 = 'This is an article on analytics vidhya'
text2 = 'article on analytics vidhya is about natural language processing'

vector1 = text_to_vector(text1)
vector2 = text_to_vector(text2)
cosine = get_cosine(vector1, vector2)
print(cosine)
```

```
0.629940788348712
```

Coreference Resolution

Coreference Resolution is a process of finding relational links among the words (or phrases) within the sentences. Consider an example sentence: ” Donald went to John’s office to see the new table. He looked at it for an hour.“

Humans can quickly figure out that “he” denotes Donald (and not John), and that “it” denotes the table (and not John’s office). Coreference Resolution is the component of NLP that does this job automatically. It is used in document summarization, question answering, and information extraction. Stanford CoreNLP provides a python wrapper for commercial purposes.

Other NLP problems / tasks

- Text Summarization – Given a text article or paragraph, summarize it automatically to produce most important and relevant sentences in order.
- Machine Translation – Automatically translate text from one human language to another by taking care of grammar, semantics and information about the real world, etc.
- Natural Language Generation and Understanding – Convert information from computer databases or semantic intents into readable human language is called language generation. Converting chunks of text into more logical structures that are easier for computer programs to manipulate is called language understanding.
- Optical Character Recognition – Given an image representing printed text, determine the corresponding text.
- Document to Information – This involves parsing of textual data present in documents (websites, files, pdfs and images) to analyzable and clean format.

Important Libraries for NLP (python)

- Scikit-learn: Machine learning in Python
- Natural Language Toolkit (NLTK): The complete toolkit for all NLP techniques.
- Pattern – A web mining module for the with tools for NLP and machine learning.
- TextBlob – Easy to use nlp tools API, built on top of NLTK and Pattern.
- spaCy – Industrial strength N LP with Python and Cython.
- Gensim – Topic Modelling for Humans
- Stanford Core NLP – NLP services and packages by Stanford NLP Group.