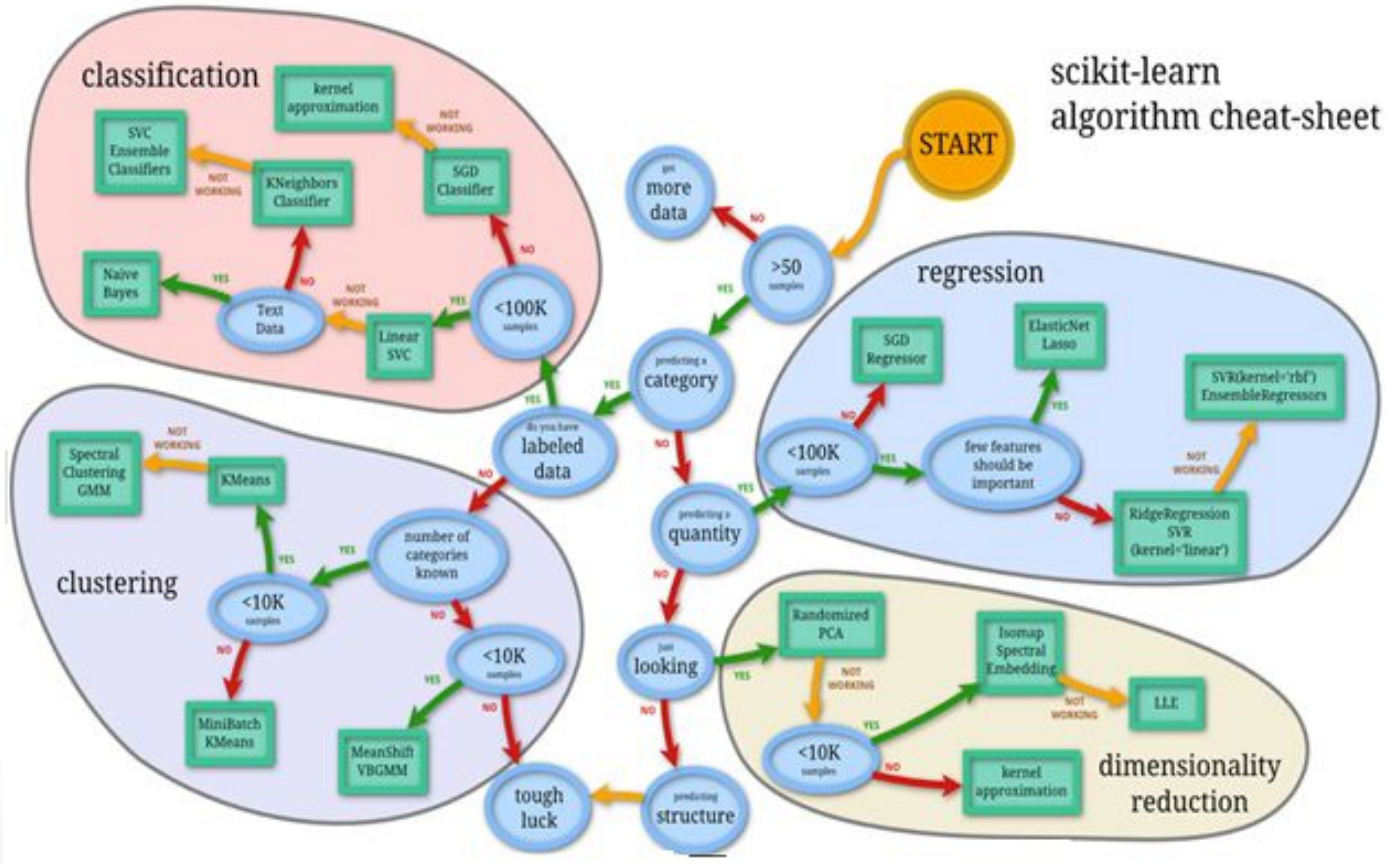
Topic 6 (Pt 2) :
Python Machine Learning for NLP



Text Clustering

- The task of **grouping a set of texts into the same group** (called a **cluster**) in such a way that the texts are **more similar to each other** than to those in other clusters
- Discover new categories in an unsupervised manner (no sample category labels provided).

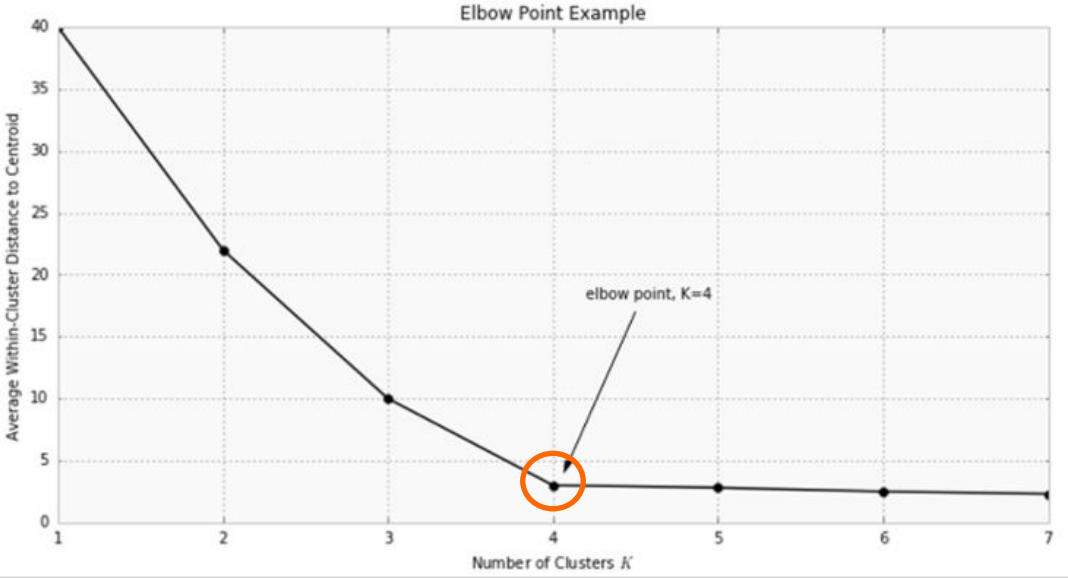
"I have these millions of documents (unstructured data). Is there a way I can group them into some meaningful categories?"

K-Means Clustering

- **Goal :**
  - Find groups in the data, with the number of groups represented by the variable *K*.
- Works iteratively to assign each data point to one of *K* groups based on the features provided.
- Data points are clustered based on feature similarity.
- Results of the *K*-means clustering algorithm :
  - The centroids of the *K* clusters, which can be used to label new data
  - Labels for the training data (each data point is assigned to a single cluster)
- Instead of defining groups before looking at data, we want to find and analyze the groups that have formed naturally.

K-Means Algorithm

- Uses iterative refinement to produce a final result.
- **Inputs** : number of clusters *K* and the data set.
- **Data set** : a collection of features for each data point.
- Starts with initial estimates for the *K* centroids, either randomly generated or selected from data set.
- The distribution of data points across groups provides insight into how the algorithm is splitting the data for each *K*. Example: using the elbow point



K-Means Algorithm (cont.)

- **Iterates between two steps:**

  Step 1. Data assignment:

  Each centroid defines one of the clusters. In this step, each data point is assigned to its nearest centroid, based on the squared Euclidean distance. More formally, if c_i is the collection of centroids in set C, then each data point x is assigned to a cluster based on

$$\underset{c_i \in C}{\operatorname{argmin}}\ dist(c_i, x$$

  where $dist(\cdot)$ is the standard (L_2) Euclidean distance. Let the set of data point assignments for each i^{th} cluster centroid be S_i.

K-Means Algorithm (cont.)

- **Iterates between two steps:**

  Step 2. Centroid update:

  In this step, the centroids are recomputed. This is done by taking the mean of all data points assigned to that centroid's cluster.

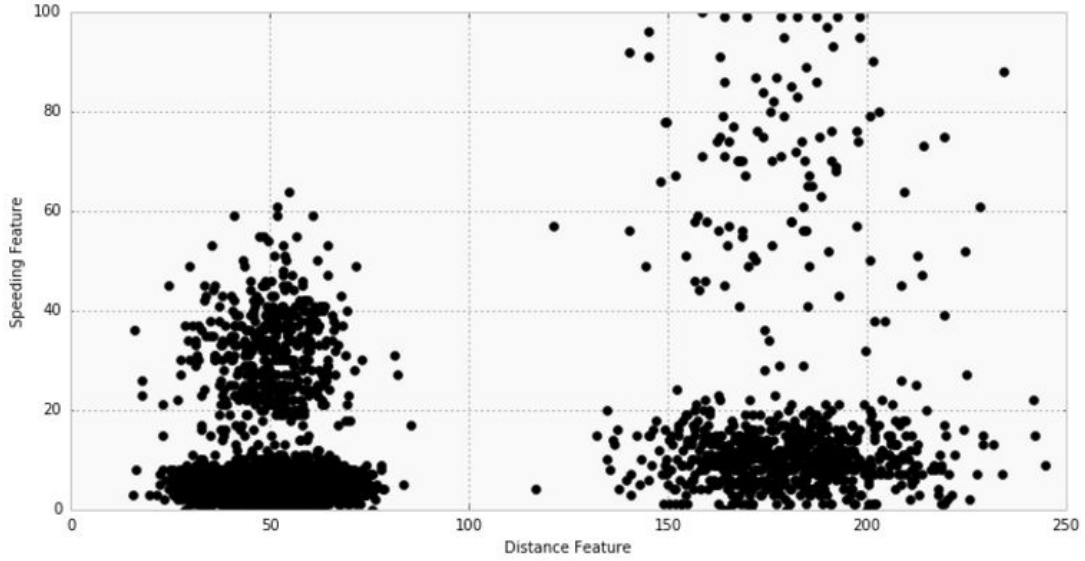
$$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

- Algorithm iterates between Step 1 and Step 2 until a stopping criteria is met (i.e., no data points change clusters, the sum of the distances is minimized, or some maximum number of iterations is reached).
- Algorithm is guaranteed to converge to a result. The result may be a local optimum (i.e. not necessarily the best possible outcome), meaning that assessing more than one run of the algorithm with randomized starting centroids may give a better outcome.

K-Means Algorithm (cont.)

- **Sample data (fleet):**

	Driver_ID	Distance_Feature	Speeding_Feature
0	3423311935	71.24	28
1	3423313212	52.53	25
2	3423313724	64.54	27
3	3423311373	55.69	22
4	3423310999	54.58	25



K-Means Text Clustering with NLTK & Scikit-learn

- **Problem: Document clustering based common terms**
  - Dataset: News articles from 3 different categories
  - Each article is plotted in relation to its similarity to the terms used across all articles.
  - Produce a 2-dimensional scatterplot of the cosine distance of each of the article (colored by cluster)

Scikit-learn

- Scikit-learn is among the most reliable machine learning libraries for Python. It has support for most of the widely used machine learning algorithms such as :
  - Decision trees (*sklearn.DecisionTreeClassifier)*
  - K-Nearest Neighbor (*sklearn.KNeighborsClassifier*)
  - Support Vector Machine (*sklearn.SVC*)
  - Naïve Bayes (*sklearn.MultinomialNB*, *sklearn.GaussianNB*, *etc*…)
- Advantages over NLTK

Scikit-learn	NLTK
Feature vectors can hold floating point numerical values	Feature vectors are considered dictionaries of names:values, where the values are strictly binary
Supports most commonly used machine learning algorithms	Supports limited no. of machine learning algorithms
Easy installation, can be imported directly into Python and NLTK as well	Needs to download NLTK package first
Works efficiently with large datasets	Works only with small datasets

Data Pre-processing

- **Remove all "stop words"**
  - **Stop words** are **commonly used words** which do not convey significant meaning.
  - These words are usually filtered out/removed, before or after the processing of natural language data
  - Example of **stop words**: {a, an, the, is, are, in, an, on, with}
- **Normalization**
  - Change all uppercase words into lower case
- **Tokenize texts (Tokenization)**
  - Split sentences into words and collect their counts
- **Stem texts (Stemming)**
  - Strip off affixes to find basic morphological structures of the terms used in articles

Data Preparation

- **Vectorization**
  - A process of turning text into numerical vectors.
  - For each article, we will produce a vector with 100 components (set through command line option).

- **Use tf-idf (term freq-inverse doc. freq)**
  - The **tf-idf** weight is a statistical measure used to evaluate how important a word/term is to a document in a collection or corpus.

tf-idf

- **tf (term frequency)**
  - Measures how frequently a term occurs in a document (a term may appear more times in long documents versus shorter ones). All terms are considered important.
  - The term frequency is often divided by the document length (i.e. the total number of terms in the document) as a way of normalization.

$$\text{tf}(t) = \frac{\text{Number of times term } t \text{ appears in a document}}{\text{Total number of terms in the document}} \quad \text{[Eq. 1]}$$

- **idf (inverse document frequency)**
  - Measures how important a term is.
  - Weighs down the frequent terms of less importance while scales up the rare ones:

$$\text{idf}(t) = log_e \left[\frac{\text{Total number of documents}}{\text{Number of documents with term } t \text{ in it}} \right] \quad \text{[Eq. 2]}$$

tf-idf (cont.)

- **Example :**

  Consider **a document containing 50 terms** wherein the word ***cat*** **appears 4 times**. Assume we have **10,000 documents** and the word ***cat*** **appears in 100** of these documents.

  Term frequency (tf) for *cat* = (4 / 50)
  = 0.08

  Inverse document frequency (idf) = $log_e(10{,}000 / 100)$
  = 4.605 (freq of doc not term)

  ∴ Tf-idf weight is the product of these quantities:

  tf-idf = 0.08 * 4.605 = 0.3684

Step 1: Tokenization with NLTK

```
import nltk
import string
import os
from collections import Counter

def get_tokens():
    os.chdir('E:/Prog/Sem2-1617/CSC 4309/data/cluster')
    with open('art1.dat', 'r') as article:
     text = article.read()
     lowers = text.lower()
     #remove the punctuation using the character deletion step of translate
     no_punctuation = lowers.translate(string.punctuation)
     tokens = nltk.word_tokenize(no_punctuation)
     return tokens

tokens = get_tokens()
count = Counter(tokens)
print(count.most_common(10))
```

Change to your own working directory

```
[('the', 9), (',', 6), ('to', 4), ('a', 4), ('ringgit', 3), ('in', 3), ('on', 3)
, ('.', 3), ('%', 3), ('us', 2)]
```

Step 2: Stop words Removal

- We remove words that are uninformative using nltk.
- Note the improved list of common words

```
from nltk.corpus import stopwords
tokens = get_tokens()
filtered = [w for w in tokens if not w in stopwords.words('english')]
count = Counter(filtered)
print(count.most_common(10))
```

```
[(',', 6), ('ringgit', 3), ('.', 3), ('%', 3), ('us', 2), ('death-cross', 2), ('
pattern', 2), ('previous', 2), ('took', 2), ('dollar', 2)]
```

Step 3: Stemming with Porter Stemmer

- Note that the counts are so much different that the ordering has been affected.
- Compare the two lists (Step 2 & 3), especially the bottom. We will notice substantial differences.

```
from nltk.stem.porter import *

def stem_tokens(tokens, stemmer):
    stemmed = []
    for item in tokens:
        stemmed.append(stemmer.stem(item))
    return stemmed

stemmer = PorterStemmer()
stemmed = stem_tokens(filtered, stemmer)
count = Counter(stemmed)
print(count.most_common(10))
```

```
[(',', 6), ('ringgit', 3), ('.', 3), ('%', 3), ('us', 2), ('death-cross', 2), ('
pattern', 2), ('currenc', 2), ('report', 2), ('move', 2)]
```

Step 4: tf-idf with Scikit-learn (combined)

- After data is cleaned, we can now use it for searching, document similarity, or other tasks (clustering, classification).

```
import nltk
import string
import os
import math
from collections import Counter
from nltk.stem.porter import PorterStemmer
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score

path = 'E:/Prog/Sem2-1617/CSC 4309/data/cluster'

token_dict = {}
stemmer = PorterStemmer()

def stem_tokens(tokens, stemmer):
    stemmed = []
    for item in tokens:
        stemmed.append(stemmer.stem(item))
    return stemmed
```

Step 4: tf-idf with Scikit-learn (cont.)

```
def tokenize(text):
    tokens = nltk.word_tokenize(text)
    stems = stem_tokens(tokens, stemmer)
    return stems

for subdir, dirs, files in os.walk(path):
    for file in files:
        file_path = subdir + os.path.sep + file
        article = open(file_path, 'r')
        text = article.read()
        lowers = text.lower()
        no_punctuation = lowers.translate(string.punctuation)
        token_dict[file] = no_punctuation
```

Step 4: tf-idf with Scikit-learn (cont.)

```
#this can take some time
tfidf = TfidfVectorizer(tokenizer=tokenize, stop_words='english')
tfs = tfidf.fit_transform(token_dict.values())

def k_means(tfs):
    true_k = 2
    model = KMeans(n_clusters = true_k, init='k-means++', max_iter=50, n_init=1)
    model.fit(tfs)
    print("Top terms per cluster:")
    order_centroids = model.cluster_centers_.argsort()[:, ::-1]
    terms = tfidf.get_feature_names()

    for i in range(true_k):
        print("Cluster %d:" % i)
        for ind in order_centroids[i, :10]:
            print(' %s' % terms[ind])

k_means(tfs)
```

Step 4: tf-idf with Scikit-learn (cont.)

- First, we iterate through every file in the cluster_data collection, converting the text to lowercase and removing punctuation.
- Next, we initialize TfidfVectorizer. In particular, we pass the TfIdfVectorizer using our own function that performs custom tokenization and stemming, but we use scikit-learn's built in stop word remove rather than NLTK's.
- Then we call fit_transform which does a few things:
  - it creates a dictionary of 'known' words based on the input text given to it.
  - Then it calculates the tf-idf for each term found in an article.
- This results in a matrix, where the rows are the individual cluster_data files and the columns are the terms. Thus, every cell represents the tf-idf score of a term in a file.

Assignment 2: Pair
(Due Tues, 19 Nov 2019)

1) Calculate your own tf, idf and tf-idf manually for the following 5 different terms (refer to Slides 12 & 13 for the equations and examples). You may then use the Scikit-learn package to compare your answers
    - win.
    - ringgit
    - trade
    - game
    - killed
2) Create a Python program with Scikit-learn that prints out the tf scores for each 't' above per document, idf scores (for 't' relative to all documents) and tf-idf scores for each term above in your Python interpreter. Plot your cluster.

- **Submit your .py solutions in pair by Tuesday, 19 Nov 2019, 11.55 pm.**

Clustering Matrix
tf-idf score

	article_ID	win	ringgit	trade	game	killed
1	art1					
2	art2					
3	art3					
4	art4					
5	art5					
6	art6					