# Natural Language Processing

**Unit 4**

## Semantic and Sentiment Analysis

## Unit IV

Semantic and Sentiment Analysis: Semantic Analysis, Exploring WordNet, Word Sense Disambiguation, Named Entity Recognition, Analysing Semantic Representation, Sentiment Analysis.

**Textbook-2 Link :**

https://revaedu-my.sharepoint.com/:b:/g/personal/shilpa_mathpati_reva_edu_in/Ed0Ohc19rdlKuyC-9pLm0d0BfIipYTU9xEW6NcfCngI05Q?e=aOob0T

# Semantic and Sentiment Analysis

Natural Language Processing (NLP) has a wide variety of applications that try to use natural language understanding to infer the meaning and context behind text and use it to solve various problems.

- Question Answering Systems
- Contextual recognition
- Speech recognition (for some applications)

➢ Text semantics specifically deals with understanding the meaning of text or language.

➢ Sentiment analysis is the popular application of text analytics, focusing on analyzing sentiment of various text resources ranging from corporate surveys to movie reviews.

➢ The key aspect of sentiment analysis is to analyze a body of text for understanding the opinion expressed by it and other factors like mood and modality.

➢ It covers techniques such as word sense disambiguation and named entity recognition

## Semantic Analysis

- It is more about understanding the actual context and meaning behind words in text and how they relate to other words to convey some information as a whole.

The following topics are covered under semantic analysis:

1. Exploring WordNet and synsets
2. Analyzing lexical semantic relations
3. Word sense disambiguation
4. Named entity recognition
5. Analyzing semantic representations

## 1. Exploring WordNet

- WordNet is a huge lexical database for the English Language.
- It was originally created in around 1985, in Princeton University’s Cognitive Science Laboratory under the direction of Professor G. A. Miller.
- This lexical database consists of nouns, adjective, verbs, and adverbs, and related lexical terms are grouped together based on some common concepts into sets, known as cognitive synonym sets or synsets .
- WordNet is used extensively as a lexical database, in text analytics, NLP, and artificial intelligence (AI)-based applications.
- The WordNet database consists of over 155,000 words, represented in more than 117,000 synsets, and contains over 206,000 word-sense pairs. The database is roughly 12 MB in size and can be accessed through various interfaces and APIs.

## Understanding Synsets

- A synset is a set or collection of synonyms that are interchangeable and revolve around a specific concept.
- Polysemous word forms (words that sound and look the same but have different but relatable meanings) are assigned to different synsets based on their meaning.
- Consider an example by using nltk ’s WordNet interface to explore synsets associated with the term, 'fruit’.

```
from nltk.corpus import wordnet as wn
import pandas as pd
term = 'fruit'
synsets = wn.synsets(term)
# display total synsets
print ('Total Synsets:', len(synsets))
```

```
Total Synsets: 5
```

## Example of five synsets associated with the term 'fruit'.

```
for synset in synsets:
    ...:     print 'Synset:', synset
    ...:     print 'Part of speech:', synset.lexname()
    ...:     print 'Definition:', synset.definition()
    ...:     print 'Lemmas:', synset.lemma_names()
    ...:     print 'Examples:', synset.examples()
    ...:     print
```

**Output:**

- Synset: Synset('fruit.n.01’)
- Part of speech: noun.plant
- Definition: the ripened reproductive body of a seed plant
- Lemmas: [u'fruit’]
- Examples: []

- Synset: Synset('yield.n.03')
- Part of speech: noun.artifact
- Definition: an amount of a product
- Lemmas: [u'yield', u'fruit']
- Examples: []

- Synset: Synset('fruit.n.03')
- Part of speech: noun.event
- Definition: the consequence of some effort or action
- Lemmas: [u'fruit']
- Examples: [u'he lived long enough to see the fruit of his policies’]

- Synset: Synset('fruit.v.01')
- Part of speech: verb.creation
- Definition: cause to bear fruit
- Lemmas: [u'fruit']
- Examples: []

- Synset: Synset('fruit.v.02')
- Part of speech: verb.creation
- Definition: bear fruit
- Lemmas: [u'fruit']
- Examples: [u'the trees fruited early this year']

## 2. Analysing Lexical Semantic Relations (ALSR)

- Text semantics refers to the study of meaning and context. Synsets give a nice abstraction over various terms and provide useful information like definition, examples, POS, and lemmas.
- ALSR explores semantic relationships among entities using synsets.

**Entailments**

- The term entailment usually refers to some event or action that logically involves or is associated with some other action or event that has taken place or will take place.

Ex: if A entails B, then whenever A is true, B must also be true.

- "If it is raining, the ground must be wet." (Raining entails wet ground.)

```
from nltk.corpus import wordnet as wn
import nltk
nltk.download('wordnet’)
for action in ['walk', 'eat', 'digest']:
    synsets = wn.synsets(action, pos='v')  # Get verb synsets
    if synsets:  # Ensure synset exists
        action_syn = synsets[0]  # Select the first synset
        print(f"{action_syn} -- entails --> {action_syn.entailments()}")
    else:
        print(f"No verb synset found for '{action}’”)
```

Output:

```
Synset('walk.v.01')    --    entails    -->    [Synset('step.v.01')]
Synset('eat.v.01')    --    entails    -->    [Synset('swallow.v.01'),
Synset('chew.v.01’)]
Synset('digest.v.01') -- entails --> [Synset('consume.v.02’)]
```

Actions like walking involve or entail stepping , and eating entails chewing and swallowing .

## Homonyms and Homographs

- ***homonyms*** refer to words or terms having the same written form or pronunciation but different meanings.
- Homonyms are a superset of homographs, which are words with same spelling but may have different pronunciation and meaning.
- The following code snippet shows how we can get homonyms/homographs:

```
for synset in wn.synsets('bank'):
    print(synset.name(), '-', synset.definition())
```

**Output:**

```
bank.n.01 - sloping land (especially the slope beside a body of water)
depository_financial_institution.n.01 - a financial institution that accepts
deposits and channels the money into lending activities
bank.n.03 - a long ridge or pile
bank.n.04 - an arrangement of similar objects in a row or in tiers
...
deposit.v.02 - put into a bank account
bank.v.07 - cover with ashes so to control the rate of burning
trust.v.01 - have confidence or faith in
```

## Synonyms and Antonyms

- Synonyms are words having similar meaning and context, and antonyms are words having opposite or contrasting meaning.

```
term = 'large’
   ...: synsets = wn.synsets(term)
   ...: adj_large = synsets[1]
   ...: adj_large = adj_large.lemmas()[0]
   ...: adj_large_synonym = adj_large.synset()
   ...: adj_large_antonym = adj_large.antonyms()[0].synset()
   ...: # print synonym and antonym
   ...: print 'Synonym:', adj_large_synonym.name()
   ...: print 'Definition:', adj_large_synonym.definition()
   ...: print 'Antonym:', adj_large_antonym.name()
   ...: print 'Definition:', adj_large_antonym.definition()
```

**Output:**

```
Synonym: large.a.01
Definition: above average in size or number or quantity or magnitude or extent
Antonym: small.a.01
Definition: limited or below average in number or quantity or magnitude or  extent
```

```
term = 'rich'
    ...: synsets = wn.synsets(term)[:3]
    ...: for synset in synsets: ...:                                        # print synonym and antonym for different synsets
    ...:     rich = synset.lemmas()[0]
    ...:     rich_synonym = rich.synset()
    ...:     rich_antonym = rich.antonyms()[0].synset()
    ...:     print 'Synonym:', rich_synonym.name()
    ...:     print 'Definition:', rich_synonym.definition()
    ...:     print 'Antonym:', rich_antonym.name()
    ...:     print 'Definition:', rich_antonym.definition()
```

**Output:**

```
Synonym: rich_people.n.01
Definition: people who have possessions and wealth (considered as a group)
Antonym: poor_people.n.01
Definition: people without possessions or wealth (considered as a group)

Synonym: rich.a.01
Definition: possessing material wealth
Antonym: poor.a.02
Definition: having little money or few possessions

Synonym: rich.a.02
Definition: having an abundant supply of desirable qualities or substances  (especially natural resources)
Antonym: poor.a.04
```

## Hyponyms and Hypernyms

- **Hypernym (Superordinate Term):** A **more general** word that includes the meaning of more specific words. **Example:** *"Fruit"* is a hypernym of *"Apple"*.
- **Hyponym (Subordinate Term):** A **more specific** word that falls under a broader category. **Example:** *"Rose"* is a hyponym of *"Flower"*.

The following snippet shows the hyponyms for the entity 'tree' :

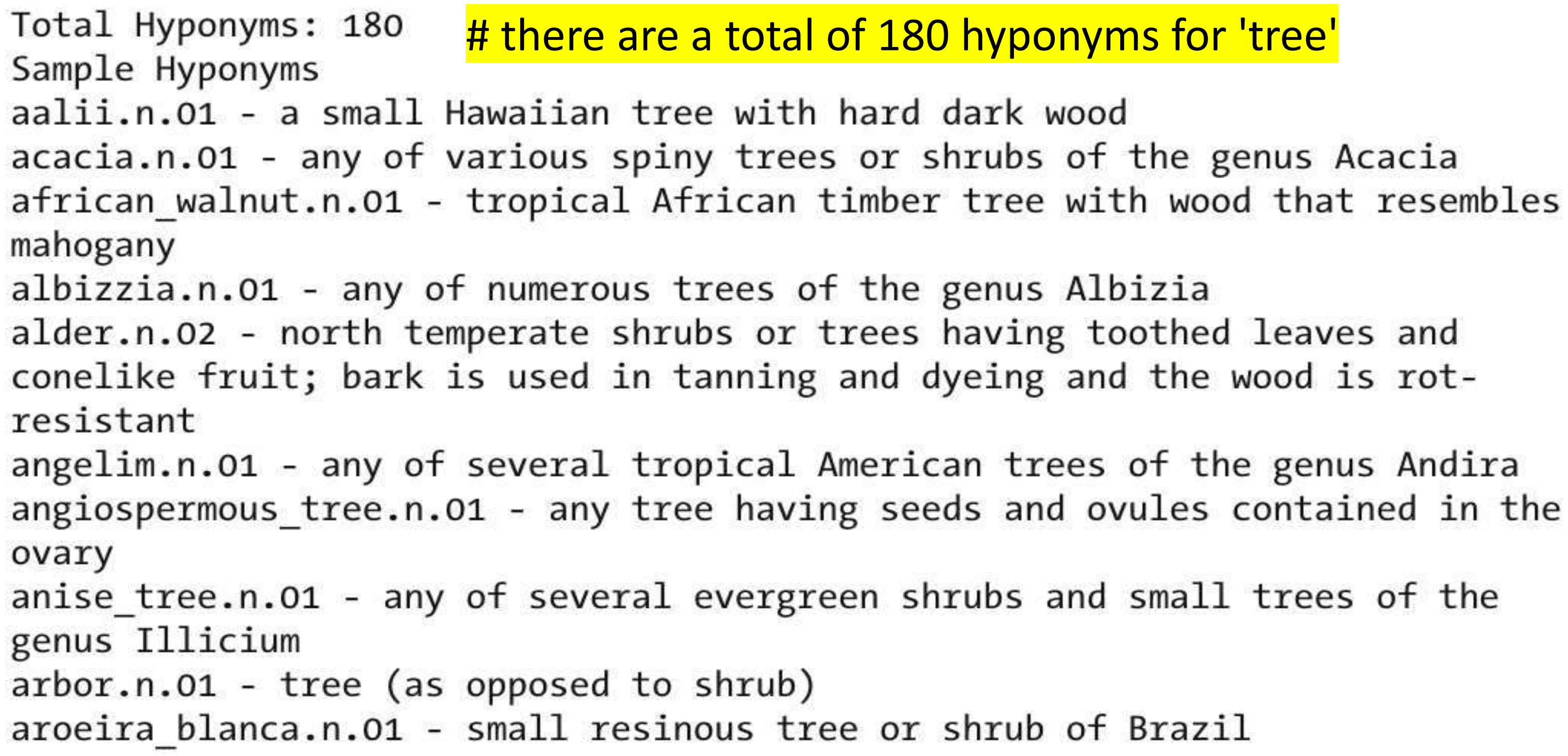
```
term = 'tree'

synsets = wn.synsets(term)

tree = synsets[0]

# print the entity and its meaning

        print 'Name:', tree.name()

        print 'Definition:', tree.definition()
```

**Output:**

```
Name: tree.n.01

Definition: a tall perennial woody plant having a main trunk and branches forming a distinct
elevated crown; includes both gymnosperms and angiosperms.
```

```
# print total hyponyms and some sample hyponyms for 'tree'
 hyponyms = tree.hyponyms()
    ...: print 'Total Hyponyms:', len(hyponyms)
    ...: print 'Sample Hyponyms'
    ...: for hyponym in hyponyms[:10]:
    ...:     print hyponym.name(), '-', hyponym.definition()
```

**Output:**



```
Total Hyponyms: 180
Sample Hyponyms
aalii.n.01 - a small Hawaiian tree with hard dark wood
acacia.n.01 - any of various spiny trees or shrubs of the genus Acacia
african_walnut.n.01 - tropical African timber tree with wood that resembles
mahogany
albizzia.n.01 - any of numerous trees of the genus Albizia
alder.n.02 - north temperate shrubs or trees having toothed leaves and
conelike fruit; bark is used in tanning and dyeing and the wood is rot-
resistant
angelim.n.01 - any of several tropical American trees of the genus Andira
angiospermous_tree.n.01 - any tree having seeds and ovules contained in the
ovary
anise_tree.n.01 - any of several evergreen shrubs and small trees of the
genus Illicium
arbor.n.01 - tree (as opposed to shrub)
aroeira_blanca.n.01 - small resinous tree or shrub of Brazil
```

# there are a total of 180 hyponyms for 'tree'

## The following snippet shows the immediate superclass hyponym for 'tree' :

```
hypernyms = tree.hypernyms()
        print hypernyms
```

**Output:** [Synset('woody_plant.n.01’)]

- You can even explore the entire hierarchy of concepts related to "tree," showing all the more specific types (hyponyms) or broader categories (parent classes) that it belongs to

```
# get total hierarchy pathways for 'tree'
hypernym_paths = tree.hypernym_paths()
        print 'Total Hypernym paths:', len(hypernym_paths)
```

**Output:** Total Hypernym paths: 1

```
# print the entire hypernym hierarchy
print 'Hypernym Hierarchy'
        print ' -> '.join(synset.name() for synset in hypernym_paths[0])
```

**Output:** Hypernym Hierarchy

entity.n.01 -> physical_entity.n.01 -> object.n.01 -> whole.n.02 -> living_thing.n.01 -> organism.n.01 -> plant.n.02 -> vascular_plant.n.01 -> woody_plant.n.01 -> tree.n.01

# Holonyms and Meronyms

- **Holonym:** A **whole** that is made up of smaller parts. **Example:** *"Tree"* is a holonym of *"Leaf".*
- **Meronym:** A **part** of something that is considered a whole. **Example:** *"Leaf"* is a meronym of *"Tree".*
- Code to show holonyms for 'tree'

```
member_holonyms = tree.member_holonyms()
        print 'Total Member Holonyms:', len(member_holonyms)
        print 'Member Holonyms for [tree]:-'
        for holonym in member_holonyms:
        print holonym.name(), '-', holonym.definition()
```

**Output:** Total Member Holonyms: 1

```
Member Holonyms for [tree]:-
forest.n.01 - the trees and other plants in a large densely wooded area
```

# Semantic Relationships and Similarity

- **Semantic Relationships** refer to the connections between words based on their meanings, such as **synonymy** (similar meanings), **antonymy** (opposite meanings), and **hyponymy** (specific vs. general).
- **Semantic Similarity** measures how closely two words or concepts are related in meaning, often based on shared characteristics or context.

- Use some sample synsets related to living entities for analysis

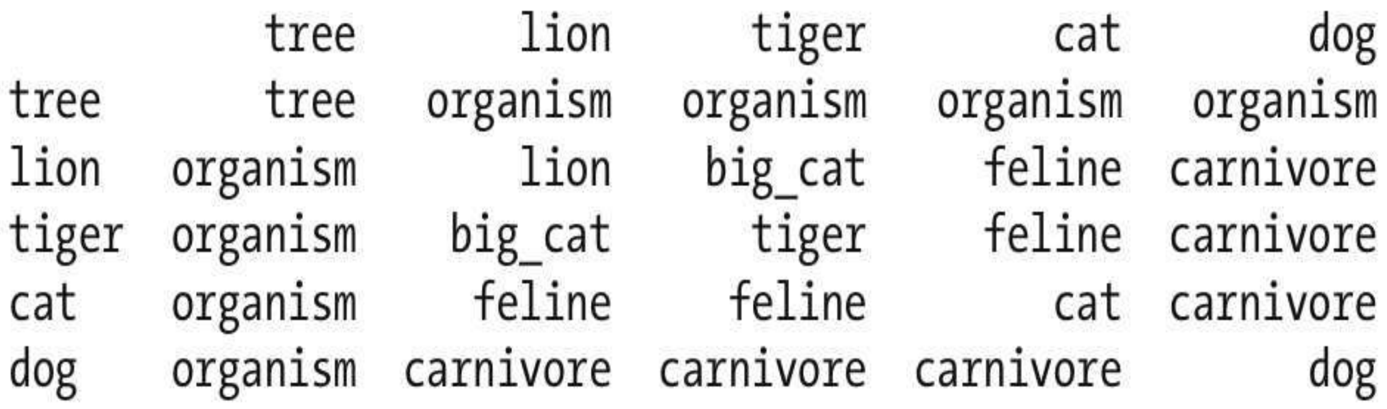
```
tree = wn.synset('tree.n.01')

lion = wn.synset('lion.n.01')

tiger = wn.synset('tiger.n.02')

cat = wn.synset('cat.n.01')

dog = wn.synset('dog.n.01’)

# create entities and extract names and definitions

entities = [tree, lion, tiger, cat, dog]

entity_names = [entity.name().split('.')[0] for entity in entities]

entity_definitions = [entity.definition() for entity in entities]

# print entities and their definitions

for entity, definition in zip(entity_names, entity_definitions):

                print entity, '-', definition
```

**Output:**

- tree - a tall perennial woody plant having a main trunk and branches forming
- a distinct elevated crown; includes both gymnosperms and angiosperms
- lion - large gregarious predatory feline of Africa and India having a tawny
- coat with a shaggy mane in the male
- tiger - large feline of forests in most of Asia having a tawny coat with
- black stripes; endangered
- cat - feline mammal usually having thick soft fur and no ability to roar:
- domestic cats; wildcats
- dog - a member of the genus Canis (probably descended from the common wolf)
- that has been domesticated by man since prehistoric times; occurs in many
- breeds

- To correlate entities, we find the **lowest common hypernym** they share, which represents their most specific common category in the hierarchy. Entities that are closely related will share a specific hypernym, while unrelated entities will share a broader, more general hypernym. This helps identify how closely or distantly the entities are related based on their shared categories.Use following code

```
common_hypernyms = []
for entity in entities:
    # get pairwise lowest common hypernyms
    common_hypernyms.append([entity.lowest_common_hypernyms(compared_entity)[0]
    .name().split('.')[0]
                        for compared_entity in entities])
# build pairwise lower common hypernym matrix
common_hypernym_frame = pd.DataFrame(common_hypernyms, index=entity_names,
                                columns=entity_names)
# print the matrix
print common_hypernym_frame
```



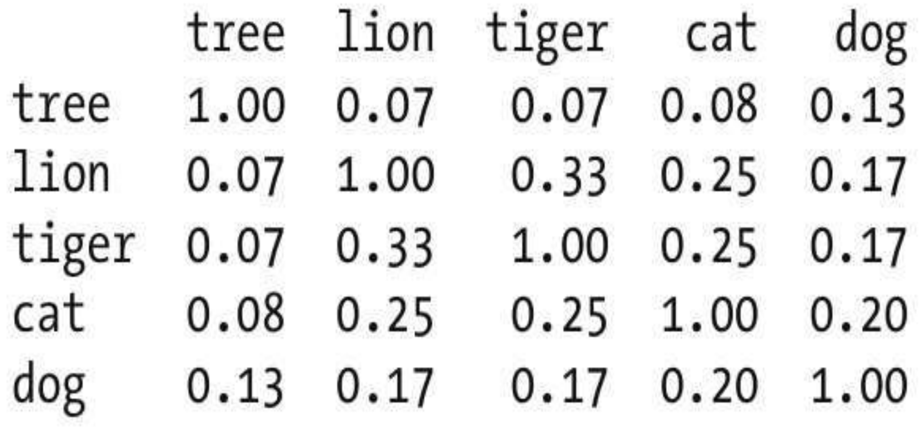
```
           tree      lion     tiger       cat        dog
tree       tree  organism  organism  organism   organism
lion   organism      lion   big_cat    feline  carnivore
tiger  organism   big_cat     tiger    feline  carnivore
cat    organism    feline    feline       cat  carnivore
dog    organism carnivore carnivore carnivore        dog
```

Ignoring the main diagonal of the matrix, for each pair of entities, we can see their lowest common hypernym which depicts the nature of relationship between them. Trees are unrelated to the other animals except that they are all living **organisms**. Hence, we get the 'organism' relationship amongst them. Cats are related to lions and tigers with respect to being **feline creatures (cat family)**, and we can see the same in the preceding output. Tigers and lions are connected to each other with the **'big cat'** relationship. Finally, we can see dogs having the relationship of **'carnivore'** with the other animals since they all typically eat meat.

- Find semantic similarity between these entities using **'path similarity’,** which returns a value between [0, 1] based on the shortest path connecting two terms based on their hypernym/hyponym-based taxonomy.
- Generate the similarity matrix as follows

```
similarities = []
for entity in entities:
    # get pairwise similarities
    similarities.append([round(entity.path_similarity(compared_entity), 2)
                         for compared_entity in entities])
# build pairwise similarity matrix
similarity_frame = pd.DataFrame(similarities,
                                index=entity_names,
                                columns=entity_names)
# print the matrix
print similarity_frame
```

**OUTPUT:**



```
       tree  lion  tiger   cat   dog
tree   1.00  0.07   0.07  0.08  0.13
lion   0.07  1.00   0.33  0.25  0.17
tiger  0.07  0.33   1.00  0.25  0.17
cat    0.08  0.25   0.25  1.00  0.20
dog    0.13  0.17   0.17  0.20  1.00
```

## 3. Word Sense Disambiguation

- **Word Sense Disambiguation (WSD)** using the **Lesk Algorithm (**a classic algorithm invented by Michael. E. Lesk in 1986) is a method to determine the correct meaning (sense) of a word based on its context in a sentence. The algorithm works by comparing the word's surrounding context to the definitions of all possible senses (meanings) of the word in a dictionary or lexicon.

**How the Lesk Algorithm Works:**

1. **Identify all possible senses** of the target word (using a lexical resource like WordNet).
2. **Compare the context of the word** (surrounding words) to the definitions of its senses.
3. **Choose the sense** whose definition overlaps the most with the context. The more overlap between the word’s meaning and its context, the more likely that sense is the correct one.

```
from nltk.wsd import lesk
from nltk import word_tokenize
# sample text and word to disambiguate
samples = [('The fruits on that plant have ripened', 'n'),
('He finally reaped the fruit of his hard work as he won the
race', 'n')]
word = 'fruit'
# perform word sense disambiguation
          for sentence, pos_tag in samples:
    ...:    word_syn = lesk(word_tokenize(sentence.lower()), word, pos_tag)
    ...:    print 'Sentence:', sentence
    ...:    print 'Word synset:', word_syn
    ...:    print 'Corresponding definition:', word_syn.definition()
    ...:    print
```

**Output:** Sentence: The fruits on that plant have ripened

Word synset: Synset('fruit.n.01')

Corresponding definition: the ripened reproductive body of a seed plant

Sentence: He finally reaped the fruit of his hard work as he won the race

Word synset: Synset('fruit.n.03')

Corresponding definition: the consequence of some effort or  action.

**# sample text and word to disambiguate**

samples = [('Lead is a very soft, malleable metal', 'n'), ('John is the actor who plays the lead in that movie', 'n'), ('This road leads to nowhere', 'v')]

word = 'lead'

**# perform word sense disambiguation**

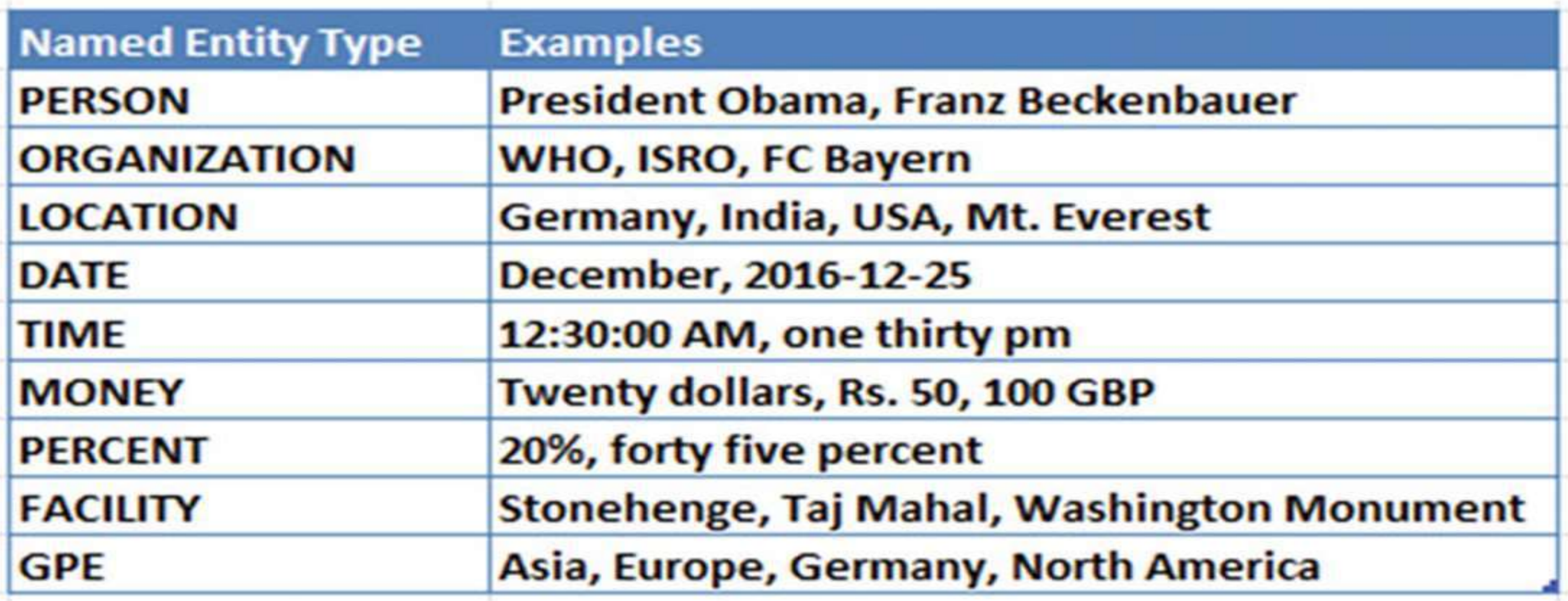
```
for sentence, pos_tag in samples:
    word_syn = lesk(word_tokenize(sentence.lower()), word, pos_tag)
     ...:     print 'Sentence:', sentence
     ...:     print 'Word synset:', word_syn
     ...:     print 'Corresponding definition:', word_syn.definition()
     ...:     print
```

**Output:**

```
Sentence: Lead is a very soft, malleable metal
Word synset: Synset('lead.n.02')
Corresponding definition: a soft heavy toxic malleable metallic element;
bluish white when freshly cut but tarnishes readily to dull grey

Sentence: John is the actor who plays the lead in that movie
Word synset: Synset('star.n.04')
Corresponding definition: an actor who plays a principal role

Sentence: This road leads to nowhere
Word synset: Synset('run.v.23')
Corresponding definition: cause something to pass or lead  somewhere
```

## 4. Named Entity Recognition (NER)

- It is an NLP task that identifies and classifies entities in text into predefined categories such as **person names**, **locations**, **organizations**, **dates**, and more. It helps extract structured information from unstructured text. NER is widely used in applications like information extraction, question answering, and text analysis.
- It is also known as **entity chunking/extraction** , is a popular technique used in information extraction to identify and segment named entities and classify or categorize them under various predefined classes.



| Named Entity Type | Examples |
|---|---|
| PERSON | President Obama, Franz Beckenbauer |
| ORGANIZATION | WHO, ISRO, FC Bayern |
| LOCATION | Germany, India, USA, Mt. Everest |
| DATE | December, 2016-12-25 |
| TIME | 12:30:00 AM, one thirty pm |
| MONEY | Twenty dollars, Rs. 50, 100 GBP |
| PERCENT | 20%, forty five percent |
| FACILITY | Stonehenge, Taj Mahal, Washington Monument |
| GPE | Asia, Europe, Germany, North America |

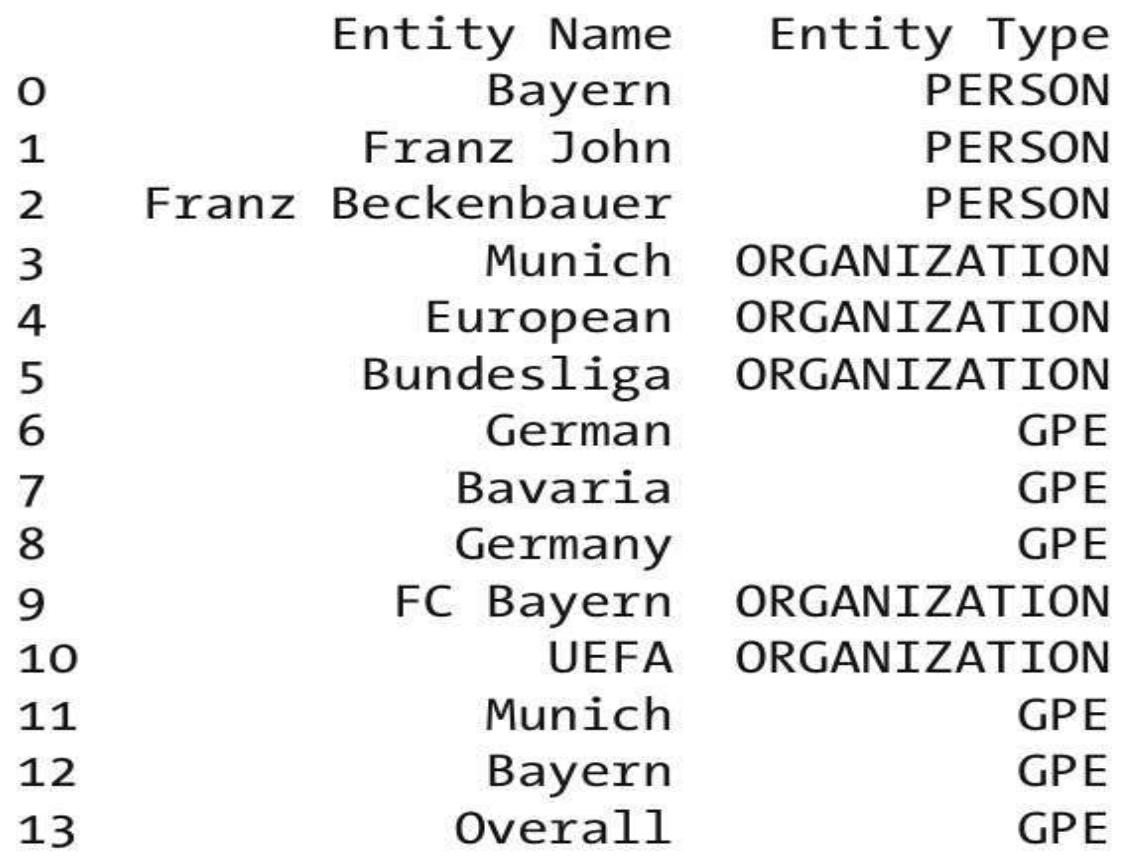
***Figure 7-1.*** *Common named entities with examples*

## Leveraging (utilizing) nltk ’s Named Entity Chunker:

```
# sample document

text = """ Bayern Munich, or FC Bayern, is a German sports club based in Munich,  Bavaria, Germany. It is best known for its professional football team,  which plays in the Bundesliga, the top tier of the German football league system, and is the most successful club in German football  history, having won a record 26 national titles and 18 national cups.  FC Bayern was founded in 1900 by eleven football players led by Franz John.  Although Bayern won its first national championship in 1932, the club  was not selected for the Bundesliga at its inception in 1963. The club had its period of greatest success in the middle of the 1970s when, under the captaincy of Franz Beckenbauer, it won the European Cup three  times in a row (1974-76). Overall, Bayern has reached ten UEFA Champions League finals, most recently winning their fifth title in 2013 as part  of a continental treble.

""“

import nltk

from normalization import parse_document

import pandas as pd

# tokenize sentences

sentences = parse_document(text)

tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in

sentences]

# tag sentences and use nltk's Named Entity Chunker

tagged_sentences = [nltk.pos_tag(sentence) for sentence in tokenized_sentences]

ne_chunked_sents = [nltk.ne_chunk(tagged) for tagged in tagged_sentences]
```

```
# extract all named entities
named_entities = []
for ne_tagged_sentence in ne_chunked_sents:
    for tagged_tree in ne_tagged_sentence:
        # extract only chunks having NE labels
        if hasattr(tagged_tree, 'label'):
            entity_name = ' '.join(c[0] for c in tagged_tree.leaves())   # get NE name
            entity_type = tagged_tree.label()                              # get NE category
            named_entities.append((entity_name, entity_type))
# get unique named entities
named_entities = list(set(named_entities))
# store named entities in a data frame
entity_frame = pd.DataFrame(named_entities,
                            columns=['Entity Name', 'Entity Type'])
# display results
print entity_frame
```

**Output:**



```
          Entity Name   Entity Type
0              Bayern        PERSON
1          Franz John        PERSON
2   Franz Beckenbauer        PERSON
3              Munich  ORGANIZATION
4            European  ORGANIZATION
5          Bundesliga  ORGANIZATION
6              German           GPE
7             Bavaria           GPE
8             Germany           GPE
9           FC Bayern  ORGANIZATION
10               UEFA  ORGANIZATION
11             Munich           GPE
12             Bayern           GPE
13            Overall           GPE
```

The Named Entity Chunker identifies named entities from the preceding text document, and we extract these named entities from the tagged annotated sentences and display them in the data frame as shown.

## 5. Analysing Semantic Representations

- Analyzing semantic representations involves studying how meaning is structured and understood in language using models like **word embeddings, semantic networks, or logic-based representations**. This helps in tasks like **text similarity, sentiment analysis, and machine translation**.

**Example:**

- **Word Embeddings (Vector Representation)**:
  - *"King" - "Man" + "Woman" = "Queen"* (shows gender-based semantic similarity)
- **Semantic Network (Concept Relations)**:
  - *Dog → (is a type of) → Animal*
  - *Apple → (is a kind of) → Fruit*

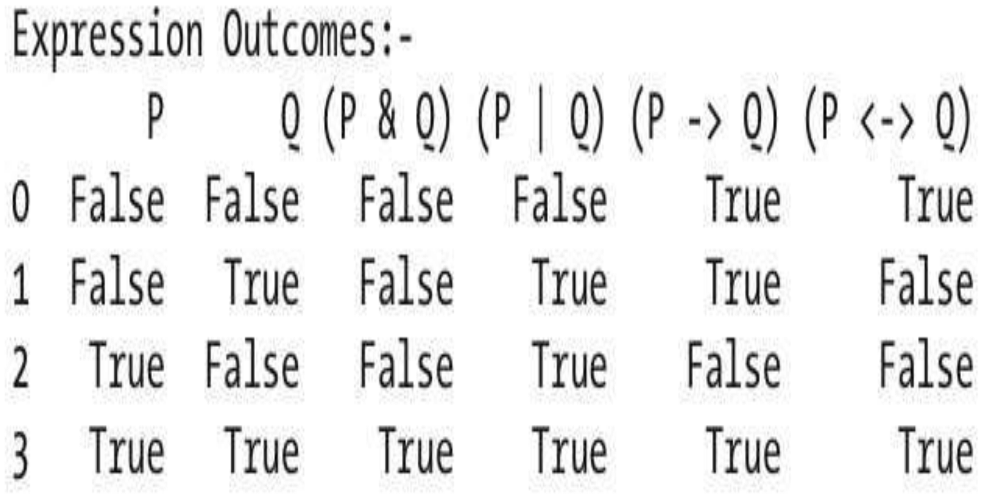
- Frameworks like propositional logic and first-order logic help us in representation of semantics.
- Propositional Logic : the study of propositions, statements, and sentences. A proposition is usually declarative, having a binary value of being either true or false.

- Consider two propositions P and Q such that they can be represented as follows: P : He is hungry Q : He will eat a sandwich We will now try to build the truth tables for various operations on these propositions using nltk based on the various logical operators

```
import pandas as pd

import os

# assign symbols and propositions

symbol_P = 'P'

symbol_Q = 'Q'

proposition_P = 'He is hungry'

propositon_Q = 'He will eat a sandwich'

# assign various truth values to the propositions

p_statuses = [False, False, True, True]

q_statuses = [False, True, False, True]

# assign the various expressions combining the logical operators

conjunction = '(P & Q)'

disjunction = '(P | Q)'

implication = '(P -> Q)'
```

```
 equivalence = '(P <-> Q)'
expressions = [conjunction, disjunction, implication, equivalence]
# evaluate each expression using propositional logic
results = []
for status_p, status_q in zip(p_statuses, q_statuses):
    dom = set([])
    val = nltk.Valuation([(symbol_P, status_p),
                          (symbol_Q, status_q)])
    assignments = nltk.Assignment(dom)
    model = nltk.Model(dom, val)
    row = [status_p, status_q]
    for expression in expressions:
        # evaluate each expression based on proposition truth values
        result = model.evaluate(expression, assignments)
        row.append(result)
    results.append(row)
```

```
# build the result table
columns = [symbol_P, symbol_Q, conjunction,
          disjunction, implication, equivalence]
result_frame = pd.DataFrame(results, columns=columns)
# display results
        print 'P:', proposition_P
    ...: print 'Q:', propositon_Q
    ...: print
    ...: print 'Expression Outcomes:-'
    ...: print result_frame
P: He is hungry
Q: He will eat a sandwich
```



```
Expression Outcomes:-
       P      Q (P & Q) (P | Q) (P -> Q) (P <-> Q)
0  False  False   False   False     True      True
1  False   True   False    True     True     False
2   True  False   False    True    False     False
3   True   True    True    True     True      True
```

- P & Q indicates He is hungry and he will eat a sandwich is True only when both of the individual propositions is True.
- Model class to evaluate each expression, where the evaluate() function internally calls the recursive function satisfy().

## First Order Logic:

- PL has limited expressive power because for each new proposition, it needs a unique symbolic representation, and it becomes very difficult to generalize facts.
- It is overcome by First order logic (FOL), which works well with features like functions, quantifiers, relations, connectives, and symbols.
- It provides a richer and more powerful representation for semantic information.
- On should know how to represent FOL representations in Python and how to perform FOL inference using proofs based on some goal and predefined rules and events in this section.

## FOL continued….

- There are several theorem provers you can use for evaluating expressions and proving theorems.
- nltk package has three main different types of provers: Prover9 , TableauProver , and ResolutionProver.

➢Prover9 is a logic tool that checks if one sentence logically follows from others using rules. It helps in semantic analysis by proving meaning relationships, like if A means B.

➢TableauProver builds a logic tree to see if a statement could be true or false.In semantics, it checks if meanings clash or if statements contradict each other.

➢ResolutionProver uses a method to break down and compare statements to find contradictions.It helps in both meaning and sentiment analysis by finding what must be true or false.

The following snippet helps in setting up the necessary dependencies for FOL expressions and evaluations

```
import  nltk
import os
# for reading FOL expressions
read_expr = nltk.sem.Expression.fromstring
# initialize theorem provers (you can choose any)
os.environ['PROVER9'] = r'E:/prover9/bin'
prover = nltk.Prover9()
# I use the following one for our examples
prover = nltk.ResolutionProver()
```

- Consider a simple expression that If an entity jumps over another entity, the reverse cannot happen .
- Assuming the entities to be x and y , we can represent this is FOL as ∀ x∀ y (jumps_over(x, y)→ ¬ jumps_over(y, x)) which signifies that for all x and y , if x jumps over y , it implies that y cannot jump over x .
- Consider now that we have two entities fox and dog such that the fox jumps over the dog is an event which has taken place and can be represented by jumps_over(fox, dog) .
- Goal or objective is to evaluate the outcome of jumps_over(dog, fox) considering the preceding expression and the event that has occurred. The following snippet shows us how we can do this:

```
# set the rule expression

rule = read_expr('all x. all y. (jumps_over(x, y) -> -jumps_over(y, x))')

# set the event occured
```

```
event = read_expr('jumps_over(fox, dog)')
# set the outcome we want to evaluate -- the goal
test_outcome = read_expr('jumps_over(dog, fox)')
# get the  result
In [132]: prover.prove(goal=test_outcome,
    ...:             assumptions=[event, rule],
    ...:             verbose=True)
[1] {-jumps_over(dog,fox)}                  A
[2] {jumps_over(fox,dog)}                   A
[3] {-jumps_over(z4,z3), -jumps_over(z3,z4)}  A
[4] {-jumps_over(dog,fox)}                  (2, 3)
Out[132]: False
```

The preceding output depicts the final result for our goal test_outcome is False , that is, the dog cannot jump over the fox if the fox has already jumped over the dog based on our rule expression and the events assigned to the assumption's parameter in the prover already given.

## Sentiment Analysis

- Textual data , mainly has two broad types of data points: factual based (objective) and opinion based (subjective).
- **Definition of Sentiment analysis:** popularly known as opinion analysis/mining , is defined as the process of using techniques like NLP, lexical resources, linguistics, and ML to extract opinion related information like emotions, attitude, mood, modality, and so on and try to use these to compute the polarity expressed by a text document.
- ***polarity*** , is used find out whether the document expresses a positive, negative, or a neutral sentiment

- ***Polarity analysis*** involves trying to assign some scores contributing to the positive and negative emotions expressed in the document and then finally assigning a label to the document based on the aggregate score.
- Two major techniques for sentiment analysis :

➢Supervised machine learning: You teach the model using labeled data, then it predicts sentiment for new text.

Ex: Input: "This product is amazing."

Model says: Positive

➢Unsupervised lexicon-based: uses a dictionary of words with sentiment scores (positive/negative) to figure out how someone feels without any training data.

Ex: Sentence:"I love this movie, but the ending was terrible.“

Lexicon says: "love" → +2, "terrible" → –3

Total score: +2 + (–3) = –1

Sentiment: Slightly negative

The key idea is to learn the various techniques typically used to tackle sentiment analysis problems so that you can apply them to solve your own problems.

# Sentiment Analysis of Internet Movie Database (IMDb) Movie Reviews

- **Dataset:** 50,000 movie reviews from the dataset containing the reviews and a corresponding sentiment polarity label which is either positive or negative. **(dataset link http://ai.stanford.edu/~amaas/data/sentiment/)**
- A positive review: movie review rated with more than six stars in IMDb.
- A negative review: rated with less than five stars in IMDb.
- An important thing is the fact that many of these reviews, even though labeled positive or negative, might have some elements of negative or positive context respectively, there is a possibility for some overlap in many reviews.
- Setting Up Dependencies: utility functions (for text normalization, feature extracting, and model evaluation), dataset, and package dependencies

# Continued……

1. **Getting and Formatting the Data:** download and unzip the files to a location of your choice and use the review_data_extractor.py file.

**Or** can directly download the parsed and formatted file from https://github.com/dipanjanS/text-analytics-with-python/tree/master/Chapter-7, which contains all datasets and code used and is the official repository for this book.

2. **Text Normalization:** re-using normalization.py module - adding an HTML stripper to remove unnecessary HTML characters from text documents, as shown

```
from HTMLParser import HTMLParser
class MLStripper(HTMLParser):
    def __init__(self):
        self.reset()
        self.fed = []
    def handle_data(self, d):
        self.fed.append(d)
    def get_data(self):
        return ' '.join(self.fed)
def strip_html(text):
    html_stripper = MLStripper()
    html_stripper.feed(text)
    return html_stripper.get_data()
```

- **Add a new function to normalize special accented (emphasized) characters and convert them into regular ASCII characters so as to standardize the text across all documents. The following snippet helps us achieve this:**

```
def normalize_accented_characters(text):
   text = unicodedata.normalize('NFKD',
                         text.decode('utf-8')
                         ).encode('ascii', 'ignore')
   return  text
```

- **Normalized code:**

```
def normalize_corpus(corpus, lemmatize=True,
                 only_text_chars=False,
                 tokenize=False):
   normalized_corpus = []
   for index, text in enumerate(corpus):
      text = normalize_accented_characters(text)
      text = html_parser.unescape(text)
      text = strip_html(text)
      text = expand_contractions(text, CONTRACTION_MAP)
      if lemmatize:
         text = lemmatize_text(text)
      else:
```

```
text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if only_text_chars:
            text = keep_text_characters(text)
        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)
        else:
            normalized_corpus.append(text)
    return normalized_ corpus
```

# Feature Extraction: use utils.py module

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
def build_feature_matrix(documents, feature_type='frequency',
                        ngram_range=(1, 1), min_df=0.0, max_df=1.0):
    feature_type = feature_type.lower().strip()
    if feature_type == 'binary':
        vectorizer = CountVectorizer(binary=True, min_df=min_df,
                                    max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'frequency':
        vectorizer = CountVectorizer(binary=False, min_df=min_df,
                                    max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(min_df=min_df, max_df=max_df,
                                    ngram_range=ngram_range)
    else:
raise Exception("Wrong feature type entered. Possible values:
'binary', 'frequency', 'tfidf'")
    feature_matrix = vectorizer.fit_transform(documents).astype(float)
    return vectorizer, feature_ matrix
```

# Model Performance Evaluation

**The following function will help us in getting the model accuracy, precision, recall, and F1-score**

```
from sklearn import metrics

import numpy as np

import pandas as pd

def display_evaluation_metrics(true_labels, predicted_labels, positive_class=1):

   print 'Accuracy:', np.round(metrics.accuracy_score(true_labels,
predicted_labels), 2)

   print 'Precision:', np.round(metrics.precision_score(true_labels,
predicted_labels,        pos_label=positive_class, average='binary’), 2)

   print 'Recall:', np.round( metrics.recall_score(true_labels, predicted_labels,
          pos_label=positive_class,       average='binary'), 2)

   print 'F1 Score:', np.round(metrics.f1_score(true_labels, predicted_labels,
          pos_label=positive_class,       average='binary’), 2)
```

Build the confusion matrix for evaluating the model predictions against the actual sentiment labels for the reviews

```
def display_confusion_matrix(true_labels, predicted_labels, classes=[1,0]):
   cm = metrics.confusion_matrix(y_true=true_labels,
   y_pred=predicted_labels, labels=classes)

   cm_frame = pd.DataFrame(data=cm,
   columns=pd.MultiIndex(levels=[['Predicted:’], classes], labels=[[0,0],[0,1]]),
   index=pd.MultiIndex(levels=[['Actual:'], classes], labels=[[0,0],[0,1]]))
 print cm_frame
```

Classification report per sentiment category (positive and negative) by displaying the precision, recall, F1-score, and support (number of reviews) for each of the classes:

```
def display_classification_report(true_labels, predicted_labels,
classes=[1,0]):
    report = metrics.classification_report(y_true=true_labels,
                                           y_pred=predicted_labels,
                                           labels=classes)
    print report
```

## Preparing Datasets

```
import pandas as pd
import numpy as np
# load movie reviews data
dataset = pd.read_csv(r'E:/aclImdb/movie_reviews.csv')
# print sample data
In [235]: print dataset.head()
                                                     review sentiment
0   One of the other reviewers has mentioned that ...  positive
1   A wonderful little production. <br /><br />The...  positive
2   I thought this was a wonderful way to spend ti...  Positive
3   Basically there's a family where a little boy ...  negative
4   Petter Mattei's "Love in the Time of Money" is...  positive
```

```
# prepare training and testing datasets
train_data = dataset[:35000]
test_data = dataset[35000:]

train_reviews = np.array(train_data['review'])
train_sentiments = np.array(train_data['sentiment'])
test_reviews = np.array(test_data['review'])
test_sentiments = np.array(test_data['sentiment’])

# prepare sample dataset for experiments
sample_docs = [100, 5817, 7626, 7356, 1008, 7155, 3533, 13010] sample_data =
[(test_reviews[index],
                test_sentiments[index])
                  for index in sample_docs]
```

## Supervised Machine Learning Technique

```
from normalization import normalize_corpus
from utils import build_feature_matrix
# normalization
norm_train_reviews = normalize_corpus(train_reviews,
lemmatize=True, only_text_chars=True)
# feature extraction
vectorizer, train_features =
build_feature_matrix(documents=norm_train_
 reviews, feature_type='tfidf', ngram_range=(1, 1), min_df=0.0,
max_df=1.0)
```

```
from sklearn.linear_model import SGDClassifier
# build the model
svm = SGDClassifier(loss='hinge', n_iter=200)
svm.fit(train_features, train_sentiments)
# normalize reviews
norm_test_reviews = normalize_corpus(test_reviews, lemmatize=True, only_
 text_chars=True)
# extract features
test_features = vectorizer.transform(norm_test_reviews)
 # predict sentiment for sample docs from test data
In [253]: for doc_index in sample_docs:
     ...:     print 'Review:-'
     ...:     print test_reviews[doc_index]
     ...:     print 'Actual Labeled Sentiment:', test_sentiments[doc_index]
     ...:     doc_features = test_features[doc_index]
     ...:     predicted_sentiment = svm.predict(doc_features)[0]
     ...:     print 'Predicted Sentiment:', predicted_sentiment
     ...:     print
```

Review:-

Worst movie, (with the best reviews given it) I've ever seen. Over the top

dialog, acting, and direction. more slasher flick than thriller.With all the

great reviews this movie got I'm appalled that it turned out so silly. shame

on you martin  scorsese

Actual Labeled Sentiment: negative

Predicted Sentiment: negative

Review:-

I hope this group of film-makers never re-unites.

Actual Labeled Sentiment: negative

Predicted Sentiment: negative

Review:-

no comment - stupid movie, acting average or worse... screenplay - no sense

at all... SKIP IT!

Actual Labeled Sentiment: negative

Predicted Sentiment: negative

Review:-

Add this little gem to your list of holiday regulars. It is<br /><br

/>sweet, funny, and endearing

Actual Labeled Sentiment: positive

Predicted Sentiment: positive

Review:-
a mesmerizing film that certainly keeps your attention... Ben Daniels is fascinating (and courageous) to watch.
Actual Labeled Sentiment: positive
Predicted Sentiment: positive
Review:-
This movie is perfect for all the romantics in the world. John Ritter has never been better and has the best line in the movie! "Sam" hits close to home, is lovely to look at and so much fun to play along with. Ben Gazzara was an excellent cast and easy to fall in love with. I'm sure I've met Arthur in my travels somewhere. All around, an excellent choice to pick up any evening.!:-)

Actual Labeled Sentiment: positive
Predicted Sentiment: positive
**Review:-**
I don't care if some people voted this movie to be bad. If you want the Truth this is a Very Good Movie! It has every thing a movie should have. You really should Get this one.
Actual Labeled Sentiment: positive
Predicted Sentiment: negative
**Review:-**
Worst horror film ever but funniest film ever rolled in one you have got to see this film it is so cheap it is unbelievable but you have to see it really!!!! P.s watch the carrot
Actual Labeled Sentiment: positive
Predicted Sentiment:  negative