Institute for Advanced Studies
in Basic Sciences
Gava Zang, Zanjan, Iran

# NATURAL LANGUAGE PROCESSING

## ONTOLOGY REPRESENTATION

# Author

Sina Heydari

Master's Student of AI
at
IASBS
Institute for Advanced Studies in Basic Sciences

Spring 2024

## Table Of Contents

# 1 A Brief Introduction To Natural Language Ontology

Natural language ontology involves studying the ontological categories, structures, and concepts that are inherent in natural language. It is a branch of "descriptive metaphysics," as coined by Strawson (1959), distinct from what Fine (2017) describes as "foundational metaphysics," which focuses on what fundamentally exists. This field intersects philosophy and linguistics, specifically within metaphysics and natural language semantics. Although recognized as a distinct area only recently due to advancements in natural language semantics, philosophers have historically engaged in natural language ontology by using language to support metaphysical arguments. It differs from the ontology accepted by individuals through philosophical or naïve reasoning and from the ontology reflected in general cognition. Essentially, the ontology of a natural language represents the implicit ontological commitments made by competent speakers through their use of the language.

# 2 Ontology In Natural Language Processing

Understanding a domain has long been acknowledged as essential for comprehending and processing information about that domain. Ontologies, which are explicit specifications of conceptualizations, are now recognized as crucial components of information systems and information processing. This homework describes a project where ontologies are integral to the reasoning process for managing and presenting information. The processes of accessing and presenting information are facilitated through natural language, with the ontologies being integrated with the lexicon used in the natural language component.

# 3 Code

## 3.1 Code Documentation

### 3.1.1 Importing The Libraries

At the beginning, we import a variety of libraries and modules related to natural language processing. These imports set up the environment for a program involving processing text data, cleaning it, analyzing it, and lastly representing the results in a graph structure.

```
import spacy
```

**Spacy:** spacy is a popular library in Python for Natural Language Processing (NLP). It provides tools for processing and analyzing large volumes of text.

```
import glob
```

**Globe:** globe finds all the pathnames matching a specified pattern according to the rules used by the Unix shell. It is used for file and directory handling.

```
from nltk.corpus import wordnet as wn
```

**WordNet:** the WordNet corpus from the Natural Language Toolkit (NLTK) library aliased wn; WordNet is a lexical database for the English language, which groups words into sets of synonyms and provides short definitions and usage examples.

```
from spacy_cleaner import Cleaner
from spacy_cleaner.processing import mutators,
    removers
```

**Spacy Cleaner:** the spacy_cleaner is a third-party module for cleaning text data processed with spaCy. This part imports mutators and removers from the spacy_cleaner, processing submodules. These are functions or classes used to modify or remove certain parts of the text.

```
1 import networkx as nx
```

**Networkx:** The networkx library is imported here, which is used for creating, manipulating, and studying the structure, dynamics, and functions of graphs.

```
1 import matplotlib.pyplot as plt
2 from matplotlib.lines import Line2D
```

**Matplotlib:** Generally, matplotlib is a library for visualization purposes. Here the Line2D class from matplotlib.lines is imported. Line2D is a class used for creating and manipulating lines in a plot.

### 3.1.2 Main Class And The Constructor

```
class Ontology:
```

**The Ontology:** class is designed to, load and process a text file stored in the given path, initializing various data structures to store and handle text data and entities, setting up a directed graph structure for potential analysis. cleaning the text using specified cleaning functions and storing the cleaned content in a dictionary.This class sets up the foundation for further text processing, analysis, and graph-based operations on the text data. A class in Python is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables), and implementations of behavior (member functions or methods).

```
    def __init__(self, path):
```

**__init__:** This is the constructor method for the Ontology class. It initializes an instance of the class. The self parameter refers to the instance being created, and path is an argument that specifies the file path to a the text file the code want to work on.

```
        self.path = path
        self.language_model =
           spacy.load("en_core_web_sm")
```

- **self.path:** stores the file path provided during the creation of the Ontology object.
- **self.language_model:** loads an English language model from spaCy, used for NLP tasks.

```
    # self.graph = nx.MultiDiGraph()
    self.graph = nx.DiGraph()
    # self.graph = nx.Graph()
```

**self.graph:** initializes a directed graph using NetworkX's DiGraph class, which allows us to do the representation with directed graphs. The commented lines suggest alternatives (MultiDiGraph for a directed graph that can hold multiple edges between nodes, and Graph for an undirected graph).

```
        self.corpus_uniques = {}
        self.corpus_dict = {}
        self.entity_list = []
        self.corpus_list = []
```

These lines initialize several empty data structures:

- **self.corpus_uniques:** A dictionary intended to store unique elements from the corpus.
- **self.corpus_dict:** A dictionary to store the processed text content.
- **self.entity_list:** A list to store entities extracted from the text.
- **self.corpus_list:** A list to store the corpus or multiple corpora.

```
        self.lemma = Cleaner(self.language_model,
                             removers.remove_number_token,
                             removers.remove_punctuation_token,
                             mutators.mutate_lemma_token)
```

**self.lemma:** initializes a Cleaner object from the spacy_cleaner module, configured with the spaCy language model and several cleaning functions:

- **removers.remove_number_token:** Function to remove number tokens.
- **removers.remove_punctuation_token:** Function to remove punctuation tokens.
- **mutators.mutate_lemma_token:** Function to change tokens to their lemmas(lemmatizing the tokens).

```
        with open(path, 'r', encoding="latin-1") as
            file:
            content = file.read()
            # Remove unwanted characters and
                normalize whitespace
            content = content.replace('\r', '') \
                             .replace('\n', '␣') \
                             .replace('\x92', '') \
                             .replace('\x93', '') \
                             .replace('\x94', '') \
                             .replace('\x95', '') \
                             .replace('\x96', '') \
                             .replace('\x97', '') \
                             .replace('\x98', '') \
                             .replace('?', '␣') \
                             .replace('!', '␣') \
                             .replace('.', '␣') \
                             .replace(':', '␣') \
                             .replace(';', '␣')
```

This block opens the file specified by path for reading with the encoding latin-1. It reads the file content into the variable content. The replace method is used to clean the text by removing unwanted characters and normalizing whitespace. This includes removing carriage returns, newline characters, and various non-standard punctuation characters, and replacing punctuation marks with spaces.

```
            self.corpus_dict[0] = content
```

Finally, the clean text is stored in the self.corpus_dict dictionary with the key 0.

### 3.1.3 The Preprocess Method

```
def preprocess(self):
```

**Preprocess:** This line defines a **method** called preprocess within the Ontology class. This method will preprocess the text stored in self.corpus_dict. The preprocess method performs several steps to prepare the text data for analysis. Including, all text conversion to lowercase to ensure uniformity, text cleaning routines to remove unwanted elements and standardize tokens, stores the cleaned text in a list for easy processing, annotating the text with linguistic features using the spaCy language model, extracting unique, non-stopword tokens and mapping them to their WordNet synsets, and building a dictionary of unique tokens as keys, and their synsets as values. This method effectively prepares the text for further analysis, such as entity recognition, relationship extraction, and graph-based modeling.

```
    # Convert text to lowercase
    for key in self.corpus_dict:
        self.corpus_dict[key] =
           self.corpus_dict[key].lower()
```

This **loop** iterates over each key in self.corpus_dict. For each key, it converts the corresponding text to lowercase to standardize the text and make the subsequent processing case-insensitive.

```
    # Clean text using spacy_cleaner
    for key in self.corpus_dict:
        self.corpus_dict[key] =
           self.lemma.clean([str(self.corpus_dict[key])])
```

This **loop** again iterates over each key in self.corpus_dict. For each key, it applies the self.lemma.clean method to clean the text using the Cleaner object initialized earlier. This involves removing numbers and punctuation and converting tokens to their lemmas

```
    # Store cleaned text in a list
    for key in self.corpus_dict:
        self.corpus_list.append(self.corpus_dict[key])
```

This **loop** iterates over each key in self.corpus_dict. For each key, it appends the cleaned text to self.corpus_list, creating a list of cleaned text entries

```
    # Process text with spaCy language model
    for i in range(len(self.corpus_list)):
        self.corpus_list[i] =
           self.language_model(str(self.corpus_list[i]))
```

This **loop** iterates over each index in self.corpus_list. For each index, it processes the text using the spaCy language model (self.language_model). This converts each text entry into a spaCy Doc object, which contains linguistic annotations.

```
    # Extract unique non-stop tokens and find their
       WordNet synsets
    for i in range(len(self.corpus_list)):
        tokens = self.corpus_list[i]
        for token in tokens:
            if not token.is_stop:
                if str(token) not in
                   self.corpus_uniques:
                    self.corpus_uniques[str(token)]
                       = set(wn.synsets(str(token)))
```

This **nested loop** iterates over each Doc object in self.corpus_list. For each Doc object, it iterates over each token in the Doc. For each token, it checks if the token is not a stop word (token.is_stop). If the token is not a stop word and is not already in self.corpus_uniques, it adds the token to self.corpus_uniques with a set of its corresponding WordNet synsets. This maps each unique token to its potential meanings as defined in WordNet.

### 3.1.4 The Build Graph Method

```
def build_graph(self):
```

The **build_graph** is a **method** within the **Ontology class**. This method will construct a graph based on the unique tokens and their relationships found in WordNet. The **build_graph method** constructs a graph by:

- **Nodes:** Each unique token from the preprocessed text is added as a node.
- **Edges:** Edges are added between nodes based on semantic relationships found in WordNet, including synset overlap, hypernym, hyponym, synonym, antonym, meronym, and holonym relationships.
- **Relationship Detection:** The method checks for each type of relationship and creates edges with appropriate attributes, building a semantically rich graph of the text data.

The graph, will further be used for ontology representation of the given documents.

```
    # Add unique tokens as nodes in the graph
    for node in self.corpus_uniques.keys():
        self.graph.add_node(node)
```

This loop iterates over each unique token in self.corpus_uniques. For each token, it adds the token as a node to the graph self.graph.

```
    nodes = list(self.corpus_uniques.keys())
    # Add edges between nodes based on
       relationships found in WordNet
    for i in range(len(nodes)):
        for j in range(i + 1, len(nodes)):
            node1 = nodes[i]
            node2 = nodes[j]
            synsets1 = self.corpus_uniques[node1]
            synsets2 = self.corpus_uniques[node2]

            relationship_found = False
            edge_attributes = {}
```

**nodes** is a list of all unique tokens (graph nodes). Two nested loops iterate over pairs of nodes. "i" is the index of the first node and "j" is the index of the second node. For each pair of nodes (node1 and node2), it retrieves their corresponding synsets (synsets1 and synsets2). relationship_found is a flag to track if a relationship is found between the nodes. And edge_attributes is a dictionary to store attributes of the edge if a relationship is found.

```
            # Check for direct synset overlap
            if synsets1 & synsets2:
                relationship_found = True
                edge_attributes['type'] =
                   'synset_overlap'
```

This part checks if there is a direct overlap between synsets1 and synsets2. If an overlap is found, it sets **relationship_found** to **True** and adds an attribute **'type'**:

- **synset_overlap** to **edge_attributes**.

```
            else:
                # Check for other meaningful
                   relationships
                for syn1 in synsets1:
                    for syn2 in synsets2:
                        if syn1 in syn2.hypernyms()
                           or syn2 in
                           syn1.hypernyms():
                            relationship_found =
                               True
                            edge_attributes['type']
                               = 'hypernym'
                        elif syn1 in
                           syn2.hyponyms() or syn2
                           in syn1.hyponyms():
                            relationship_found =
                               True
                            edge_attributes['type']
                               = 'hyponym'
                        elif any(lemma in
                           syn2.lemmas() for lemma
                           in syn1.lemmas()):
```

```
                relationship_found = True
                edge_attributes['type'] = 'synonym'
            elif any(antonym for lemma in syn1.lemmas()\
                    for antonym in lemma.antonyms()\
                    if antonym in syn2.lemmas()):
                relationship_found = True
                edge_attributes['type'] = 'antonym'
            elif syn1 in syn2.part_meronyms() or syn2 in syn1.part_meronyms():
                relationship_found = True
                edge_attributes['type'] = 'meronym'
            elif syn1 in syn2.part_holonyms() or syn2 in syn1.part_holonyms():
                relationship_found = True
                edge_attributes['type'] = 'holonym'

            if relationship_found:
                self.graph.add_edge(node1, node2, **edge_attributes)
                break
        if relationship_found:
            break
```

This **nested loop** iterates over each synset in synsets1 and synsets2. For each pair of synsets (syn1 and syn2), it checks for various types of relationships:

- **Hypernym:** If syn1 is a hypernym of syn2 or vice versa.
- **Hyponym:** If syn1 is a hyponym of syn2 or vice versa.
- **Synonym:** If any lemma in syn1 matches a lemma in syn2.
- **Antonym:** If any lemma in syn1 has an antonym in syn2.
- **Meronym:** If syn1 is a part meronym of syn2 or vice versa.
- **Holonym:** If syn1 is a part holonym of syn2 or vice versa.

If any relationship is found, it sets **relationship_found** to **True**, assigns the appropriate relationship type to **edge_attributes**, and adds an **edge** between **node1** and **node2** with the attributes.
The break statements exit the inner loops once a relationship is found to avoid redundant checks.

### 3.1.5 The Visualize Graph Method

```
def visualize_graph(self):
```

This method provides a visual representation of the semantic relationships between the unique tokens extracted from the document, allowing for easier interpretation and analysis.
The **method** visualizes the graph of entities and their relationships; cleans the graph by removing nodes without edges; uses a spring layout for positioning nodes; sets edge colors based on the types of relationships; plots the graph with labels, node sizes, colors, and edge colors; creates and displays a legend for the edge colors. sets the title and displays the graph.

```
        # Remove nodes without any edges
    isolated_nodes = list(nx.isolates(self.graph))
    self.graph.remove_nodes_from(isolated_nodes)
```

This line defines a **method visualize_graph** within the Ontology class. This method will visualize the graph constructed in the build_graph method.

```
    # Define layout for graph visualization
    pos = nx.spring_layout(self.graph, k=1,
       iterations=50)
    plt.figure(figsize=(15, 15))
```

This line defines the layout for the graph visualization using the spring_layout function from **NetworkX**. The spring_layout positions nodes using the Fruchterman-Reingold force-directed algorithm.

- **k=1** is the optimal distance between nodes, and iterations=50 is the number of iterations to perform.
- **plt.figure(figsize = (15, 15))** sets the figure size for the plot to 15x15 inches.

```
                # Define edge colors based on
                    relationship types
    edge_colors = []
    edge_color_map = {
        'synset_overlap': 'black',
        'hypernym': 'blue',
        'hyponym': 'green',
        'synonym': 'purple',
        'antonym': 'red',
        'meronym': 'orange',
        'holonym': 'brown'
    }

    for _, _, data in self.graph.edges(data=True):
        edge_colors.append(edge_color_map.get(data['type'],
            'gray'))
```

This section initializes an empty list edge_colors to store the colors of edges. edge_color_map is a dictionary that maps relationship types to specific colors. The loop iterates over all edges in the graph, retrieving the data associated with each edge. For each edge, it appends the appropriate color (based on the relationship type) to edge_colors. If the relationship type is not found in edge_color_map, it defaults to **gray**.

```
                # Draw the graph with labels,
                    colors, and sizes
    nx.draw(self.graph, pos, with_labels=True,
       node_size=2500,
            node_color="yellow", font_size=10,
               font_weight="bold",
               edge_color=edge_colors)
```

This line uses the nx.draw function to draw the graph with various parameters.
**pos:** Node positions defined by the spring layout.

- with_labels=True: Display node labels.
- node_size=2500: Set the size of the nodes.
- node_color="yellow": Set the color of the nodes.

- font_size=10: Set the font size of the labels.
- font_weight="bold": Set the font weight of the labels.
- edge_color=edge_colors: Set the color of the edges based on their relationship types

```
    # Create legend for edge colors
    legend_elements = [Line2D([0], [0],
       color=color, linewidth=4 , label=label)
                       for label, color in
                          edge_color_map.items()]
    plt.legend(handles=legend_elements, loc='best')
```

This section creates a legend for the edge colors using Line2D from **matplotlib.lines. legend_elements** is a list of Line2D objects, each representing a different relationship type and its corresponding color. **plt.legend(handles=legend_elements, loc='best')** adds the legend to the plot at the best location.

```
    # Set the title and display the graph
    plt.title(f'Document␣{self.path}␣Entity␣Graph')
    plt.show()
```

This line sets the title of the plot to indicate which document the graph represents, using the path of the document. **plt.show()** displays the graph.

### 3.1.6 Object Creation And Execution

```
# Representing one document
document_list = glob.glob('*.txt')

represent = Ontology(document_list[0])
represent.preprocess()
represent.build_graph()
represent.visualize_graph()

# Representing all the documents (in separate
   figures)
for document in document_list:
    represent = Ontology(document)
    represent.preprocess()
    represent.build_graph()
    represent.visualize_graph()
```

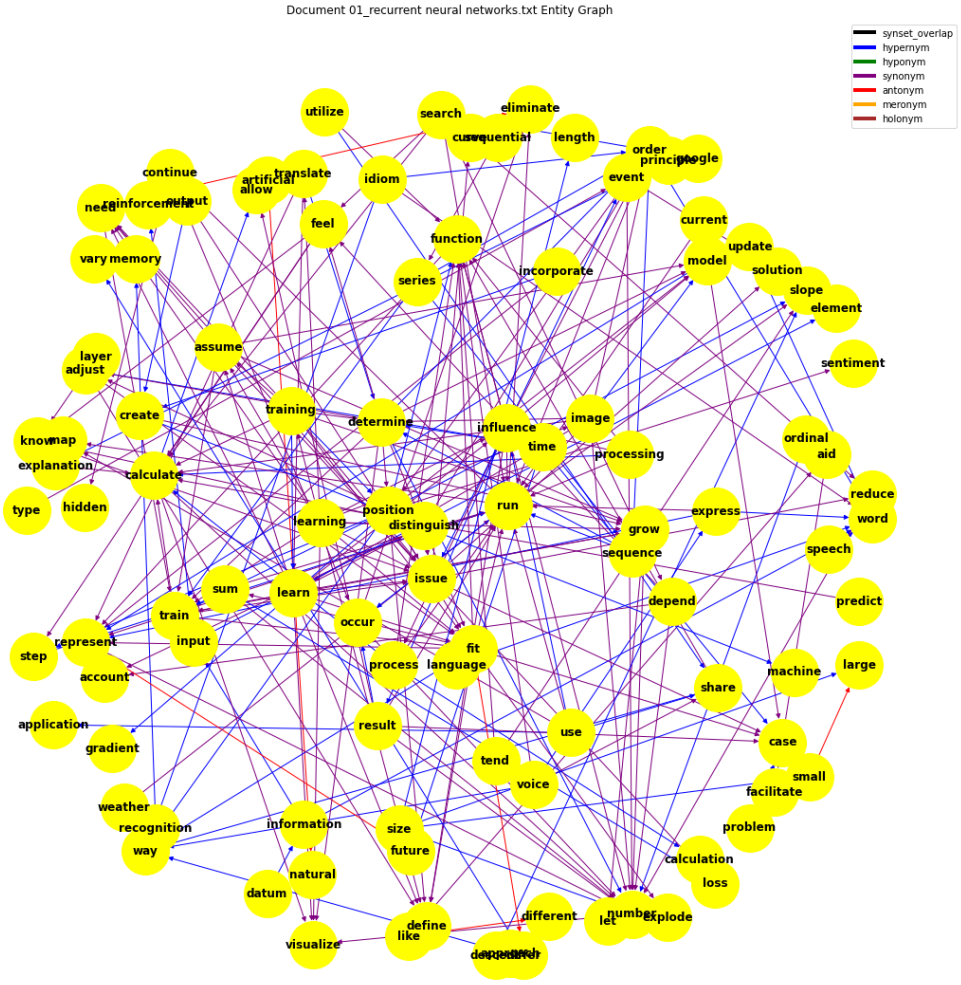
This code snippet demonstrates two main functionalities:

- **Single Document Processing and Visualization**, Processes and visualizes the graph for the first document in the list.
- **Multiple Document Processing and Visualization**, Iterates through each document in the list, processes, and visualizes the graph for each document in separate figures.

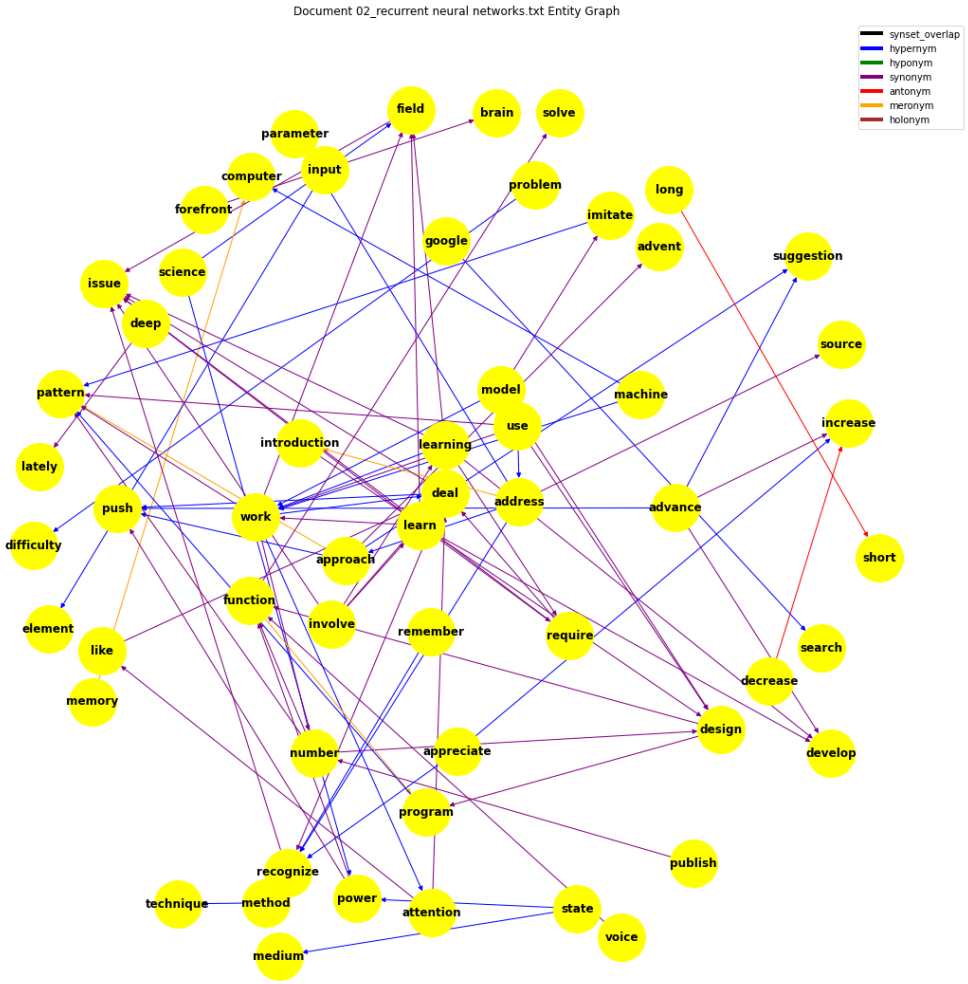
This part allows for detailed semantic analysis of individual documents and a comprehensive analysis of multiple documents by visualizing their entity relationships.

# 4 Results

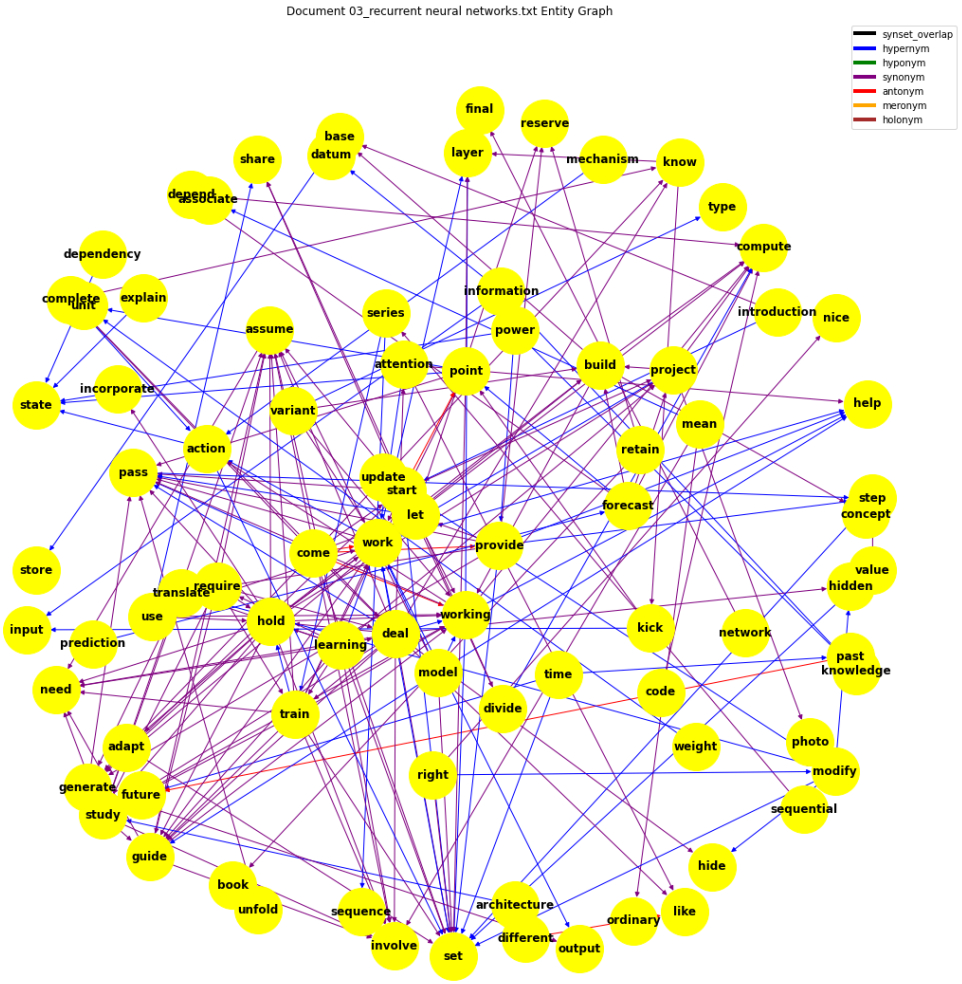
## 4.1 01_recurrent neural networks



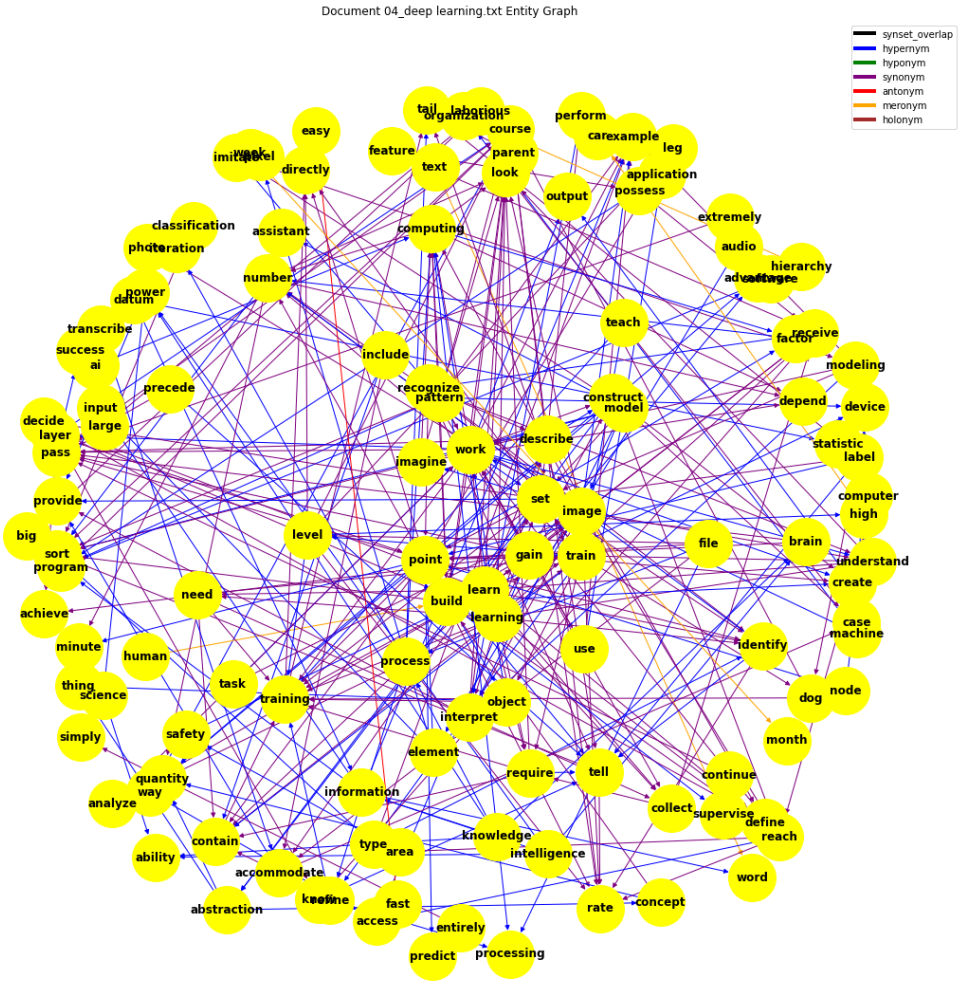
## 4.2 02_recurrent neural networks



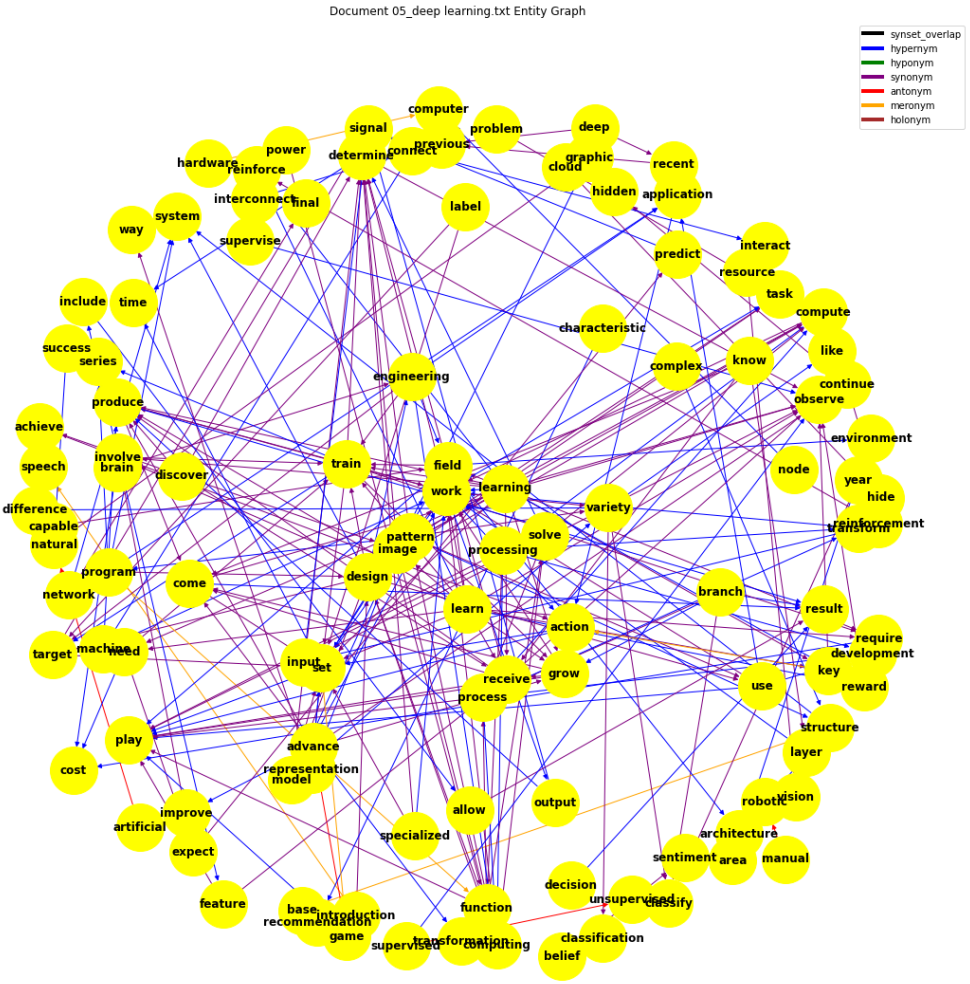
## 4.3 03_recurrent neural networks



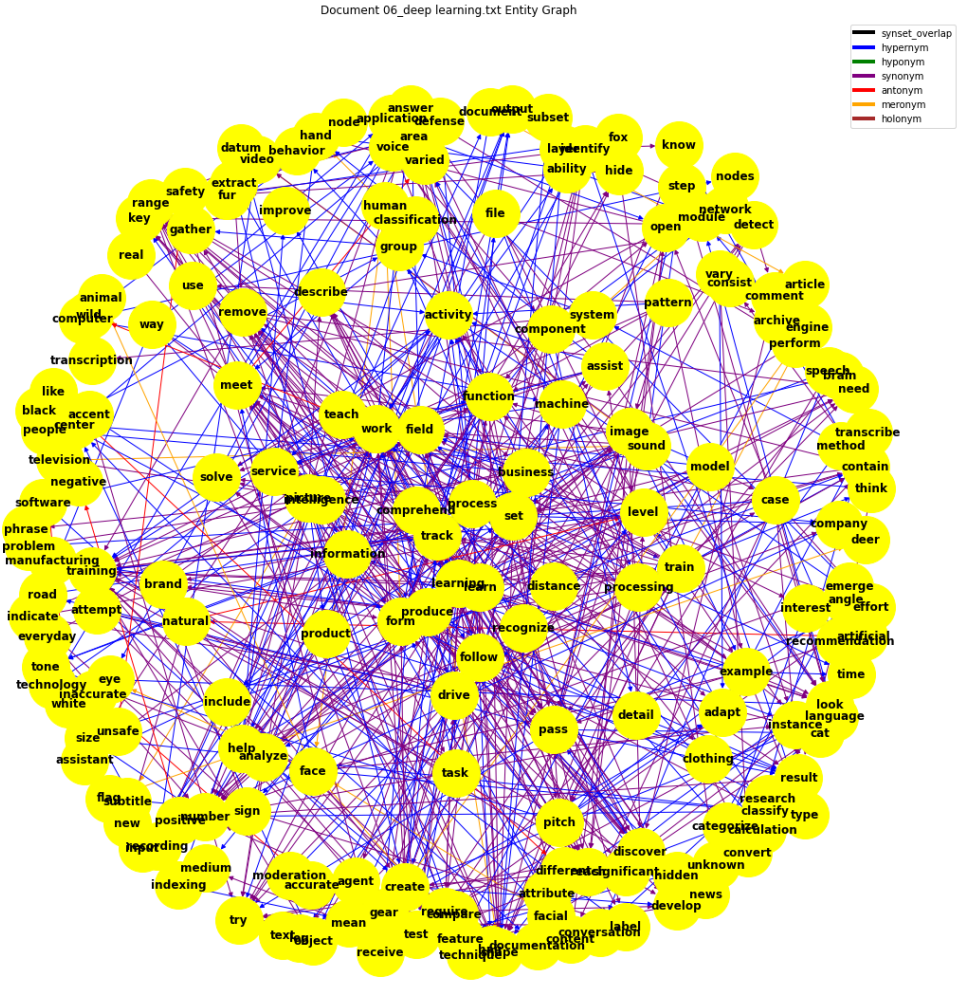
## 4.4 04_deep learning



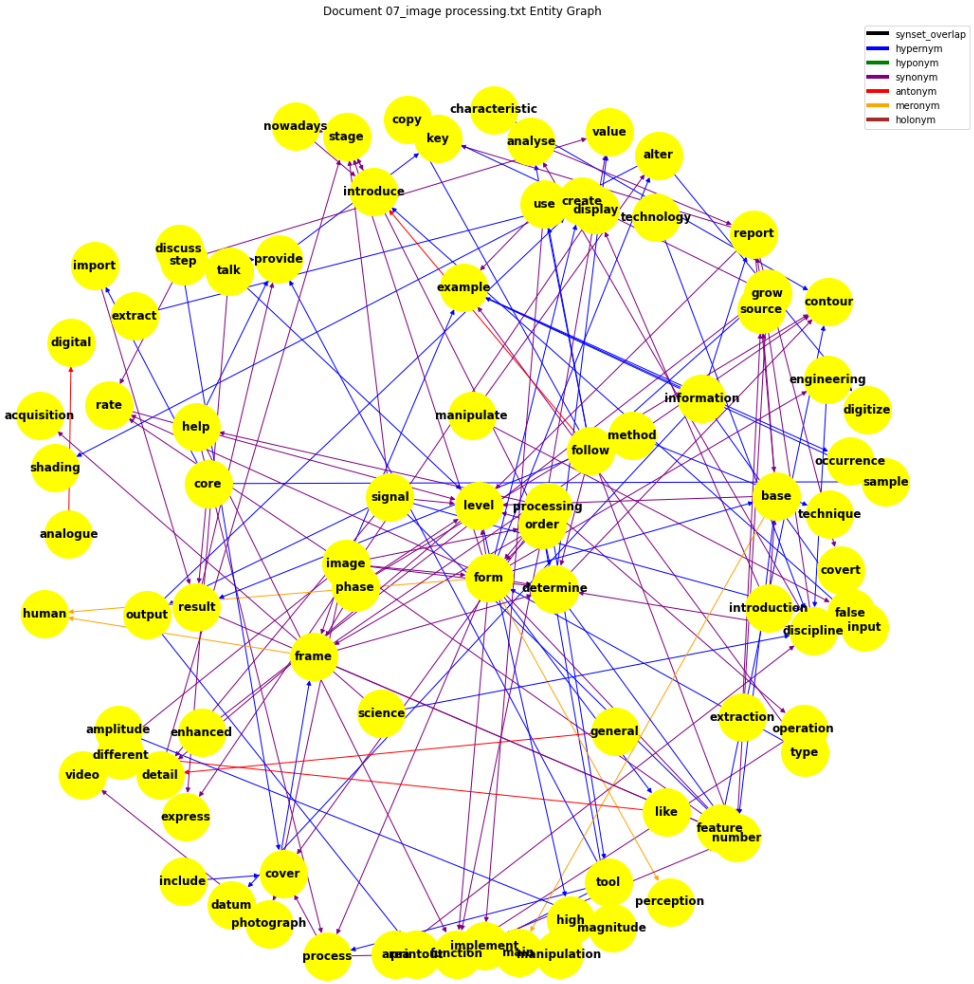
## 4.5 05_deep learning



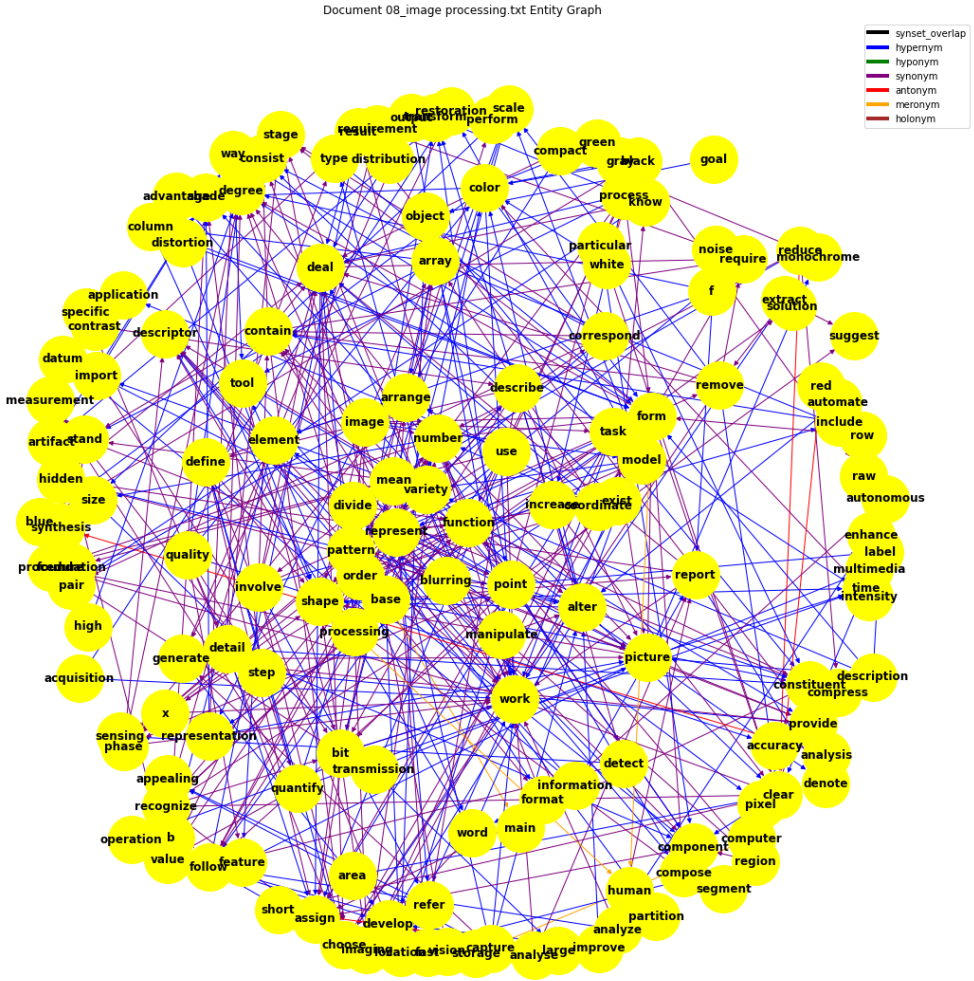
## 4.6 06_deep learning



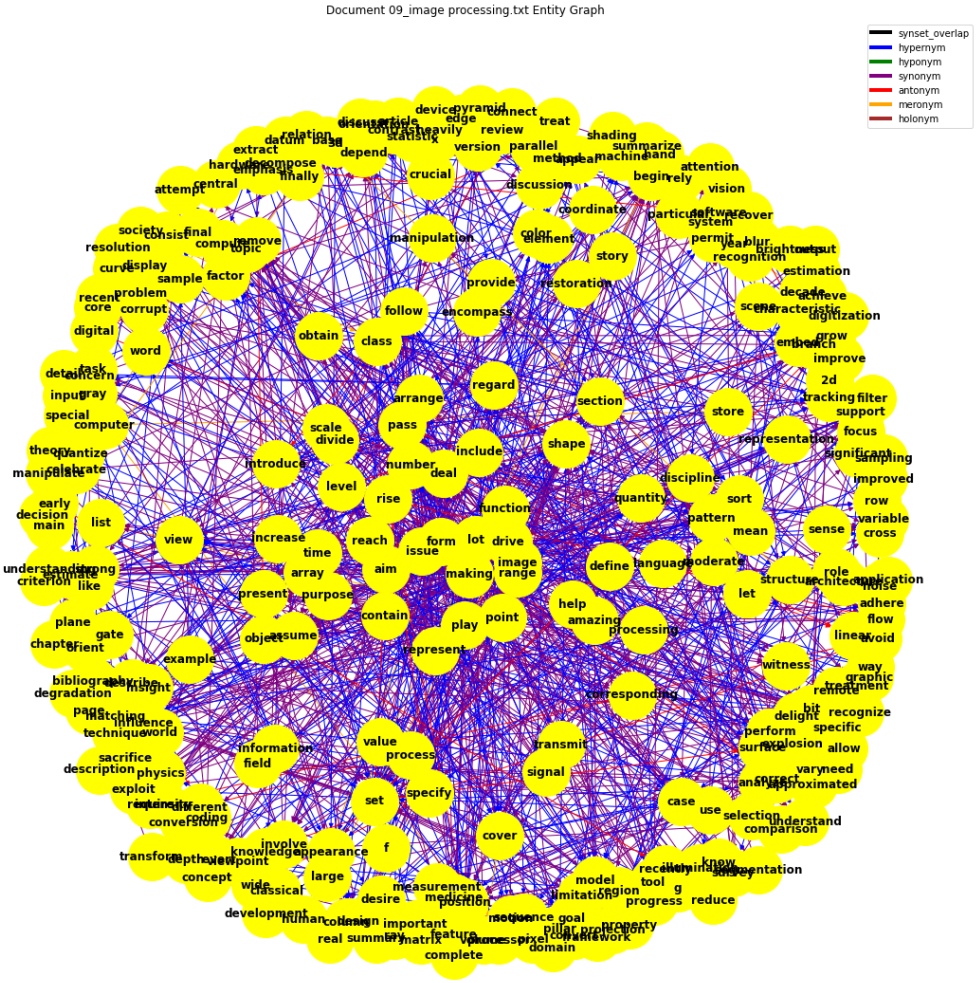
## 4.7 07_image processing



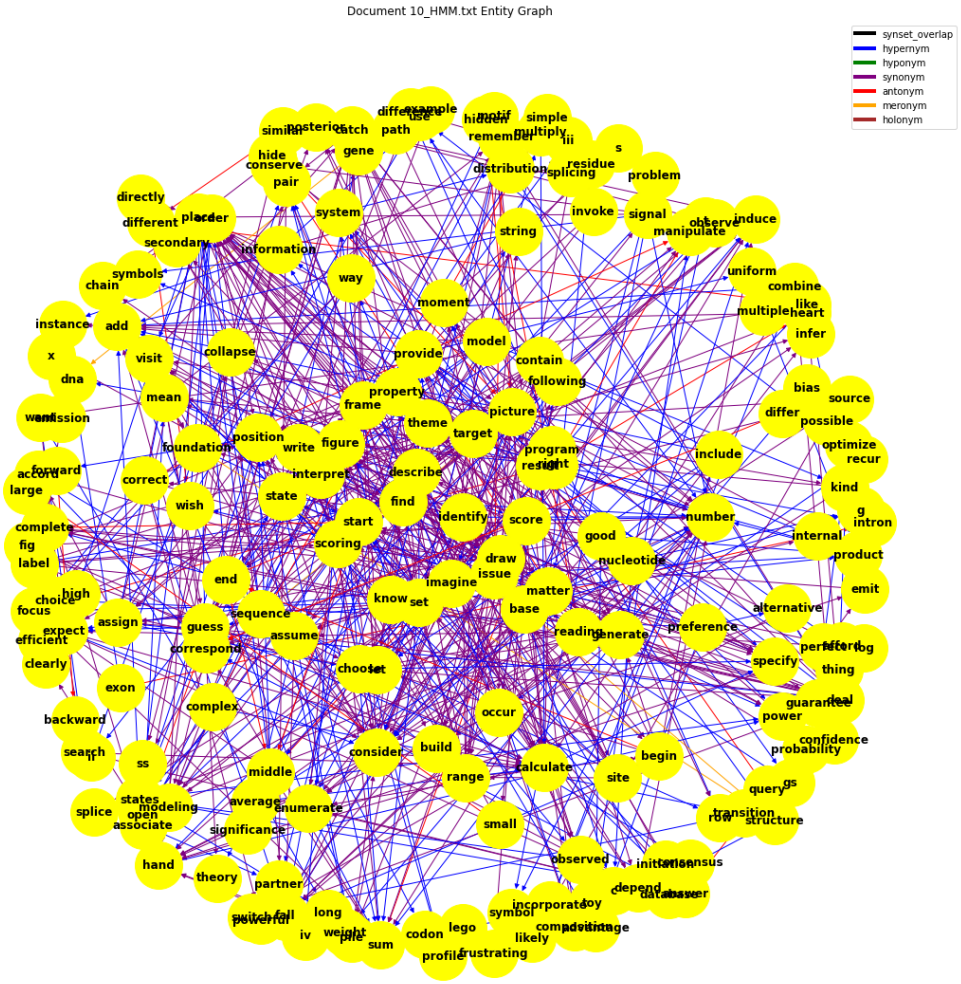
## 4.8 08_image processing



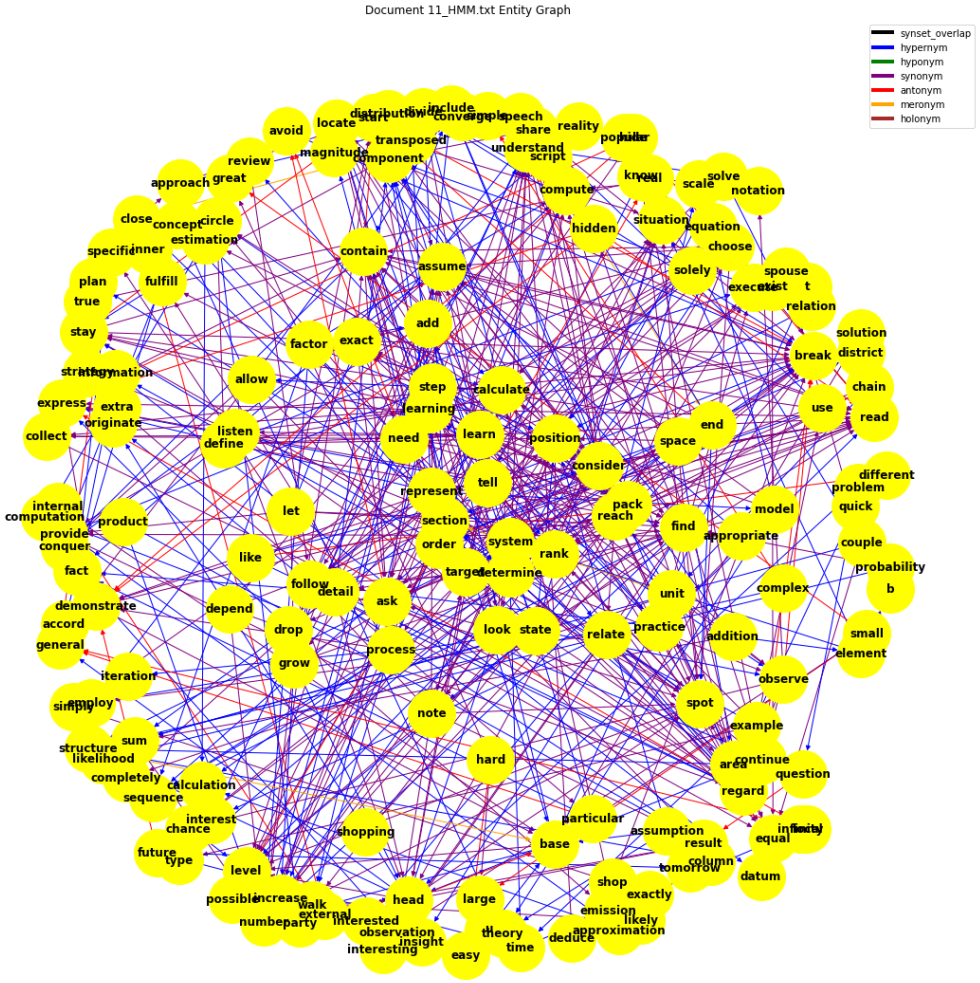
## 4.9 09_image processing



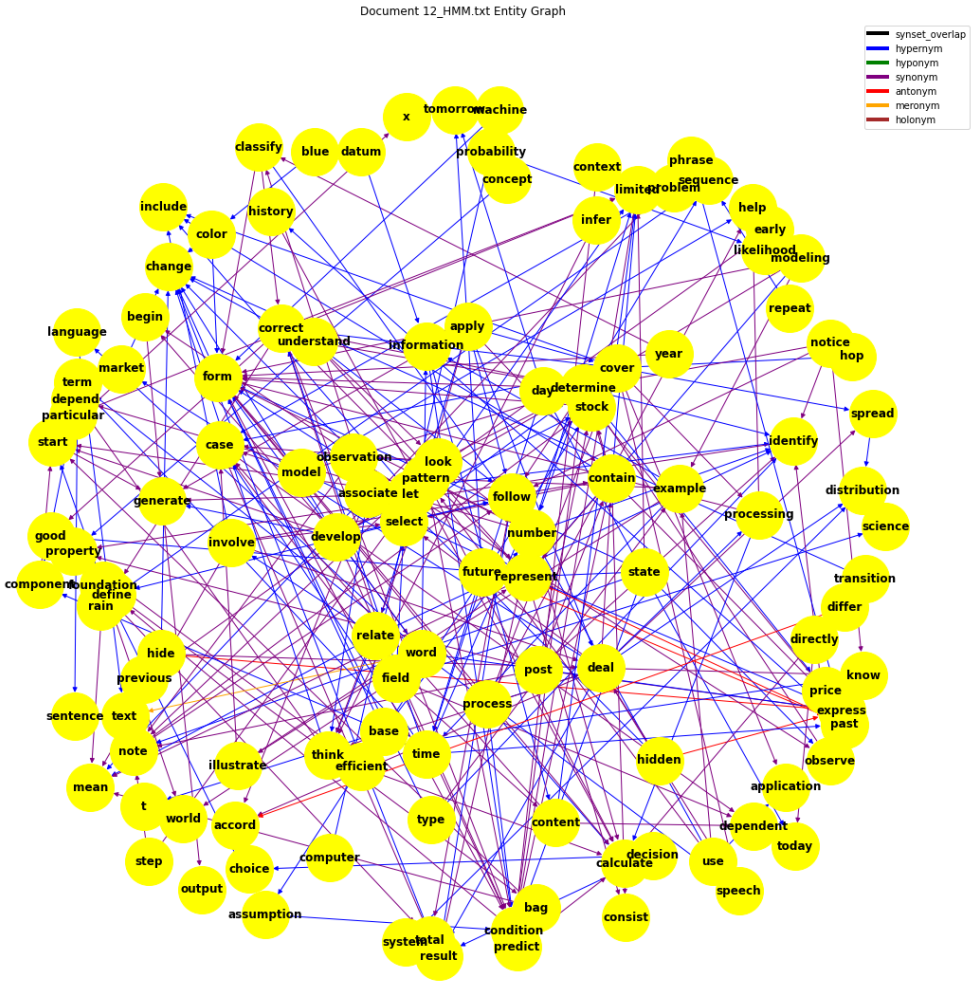
## 4.10 10_HMM


Document 10_HMM.txt Entity Graph
synset_overlap
hypernym
hyponym
synonym
antonym
meronym
holonym
example
motif
simple
hidden
multiply
posterior
catch
path
gene
distribution
residue
splicing
problem
hide
conserve
pair
directly
different
system
invoke
signal
observe
induce
manipulate
secondary
information
string
symbols
chain
way
uniform
combine
like
moment
multiple
heart
infer
instance
add
model
x
visit
collapse
provide
contain
dna
following
bias
source
property
mean
frame
picture
differ
possible
theme
target
position
write
figure
program
optimize
foundation
include
recur
forward
correct
interpret
describe
kind
large
state
find
wish
identify
score
number
g
intron
complete
start
good
internal
fig
scoring
nucleotide
product
label
draw
issue
imagine
matter
emit
end
know
set
base
choice
high
focus
sequence
alternative
assign
guess
reading
generate
preference
expect
assume
efficient
correspond
specify
clearly
choose
thing
exon
complex
guarantee
power
backward
occur
confidence
search
consider
build
probability
ss
middle
range
calculate
begin
site
query
states
average
transition
splice
modeling
enumerate
row
structure
open
associate
significance
small
hand
theory
partner
long
codon
lego
likely
profile
frustrating
sum

## 4.11 11_HMM


Document 11_HMM.txt Entity Graph
synset_overlap
hypernym
hyponym
synonym
antonym
meronym
holonym
avoid
locate
share
reality
review
great
approach
close
concept
circle
estimation
specific
inner
plan
fulfill
true
stay
factor
exact
add
assume
contain
allow
step
calculate
learning
express
extra
originate
collect
listen
define
need
learn
position
consider
tell
represent
let
internal
computation
product
provide
conquer
section
order
system
rank
like
fact
follow
detail
ask
target
determine
demonstrate
depend
drop
look
state
relate
accord
general
process
grow
iteration
simply
employ
note
sum
structure
likelihood
completely
calculation
sequence
hard
interest
chance
shopping
particular
assumption
result
future
type
level
base
column
tomorrow
possible
increase
walk
external
head
large
shop
exactly
emission
likely
number
party
interested
observation
insight
interesting
theory
deduce
approximation
easy
time
solve
notation
scale
compute
hidden
situation
equation
choose
solely
spouse
relation
solution
break
district
chain
use
read
end
space
different
problem
quick
model
pack
reach
find
appropriate
couple
probability
complex
unit
practice
addition
small
element
observe
spot
example
continue
area
question
regard
equal
datum

## 4.12 12_HMM


Document 12_HMM.txt Entity Graph
synset_overlap
hypernym
hyponym
synonym
antonym
meronym
holonym
x
tomorrow
machine
classify
blue
datum
probability
concept
context
phrase
sequence
problem
limited
include
history
infer
help
early
likelihood
modeling
color
change
repeat
begin
language
correct
understand
information
apply
notice
hop
market
form
cover
year
term
depend
particular
day
determine
stock
spread
start
case
observation
look
pattern
identify
model
contain
example
distribution
generate
associate
let
follow
processing
good
property
select
number
science
involve
develop
component
foundation
future
represent
state
transition
define
rain
differ
relate
word
directly
hide
previous
field
post
deal
know
price
sentence
text
process
express
past
base
note
think
time
hidden
observe
illustrate
efficient
mean
application
t
world
type
content
dependent
accord
today
step
choice
computer
decision
use
calculate
output
speech
assumption
bag
consist
condition
predict
system
total
result

# 5 References


- Ontology In Natural Language
- Introduction to Ontologies
- Semantic Web and RDF
- spacy-cleaner
- spacy
- glob
- NetworkX
- Matplotlib
- nltk