COMP3074 Human-AI Interaction
Lab 1: Modelling and Generating Language

Jérémie Clos

October 2023

Contents

1 Introduction

The goal of this lab is to introduce you to basic parsing of text in Python and to the building and use of language models. This lab script is made up of multiple sections with code, explanations, and practice problems for you to experiment with as you read. I encourage you to do those tasks and play around with the code to become familiar with Python and its features. Some of the details will be left vague on purpose: you should be competent enough in core computer science skills to figure out simple stuff like finding a dataset, downloading resources, or reading/writing from/to text files.

WARNING: copy and pasting from a PDF file can sometimes mess up code alignment and/or quotation marks around strings. If you copy from the PDF and it does not work, make sure to check this.

How make good use of these labs You will have noticed that labs are long and full of details and code samples. How to make best use of them? Read them in detail and make sure to understand their content. Lab scripts are written to complement the lectures by providing additional details, examples of implementation, and ideas for things that can be done. Read the lab script, copy the code (or better, retype it), and spend some time modifying and playing with the code samples to ensure that you understand what they do. Try to extend them to perform additional things or to combine them with each other. By doing so, you will be able to build deeper knowledge of the theory and the practice, and you will make your coursework a lot easier.

2 Language models

A (statistical) language model can basically be thought of as a probability distribution over words of a vocabulary. For a unigram language model, the words are independent, so our probability distribution can be represented with a simple Python dictionary linking each word to its probability of appearing. For a bigram language model, those probabilities are conditioned on the previous word of the sentence, and therefore a more complicated data structure is necessary. One way to represent it would be a table, so that any pair of words (w_1, w_2) can be associated with a probability. Another way of representing it would be to use embedded Python dictionaries. The choice is yours here, because it doesn't really matter.

2.1 Building a statistical language model

Now let us see how we could about and build our own, in-house, custom, language model. Let us start by collecting data into a single text file, for ease of use purposes, which we will name `dataset.txt`. This file can contain anything you want: download some books from Gutenberg, or some random text files, or even copy paste the content of your project proposal. If you run out of ideas, I have added the dataset of the Berkeley Restaurant project on Moodle that you can use as an example - just make sure to change the filename in line 6.

We will then set up the libraries we will need for this task (namely NLTK, and some standard Python libraries) and pre-process our dataset for further analysis.

```
1 import nltk, re, pprint, string
2 from nltk import word_tokenize, sent_tokenize
3
4 string.punctuation = string.punctuation +"’"+"–"+"‘"+"—" # adds some additional
      characters to the punctuation
5 string.punctuation = string.punctuation.replace(".", "")
6 file = open("dataset.txt" , encoding = "utf8").read()
7 file_nl_removed = ""
8 for line in file:
```

```
  line_nl_removed = line.replace("\n", " ") # removes newline characters
  file_nl_removed += line_nl_removed # adds filtered line to the list
file_p = "".join([char for char in file_nl_removed if char not in string.
    punctuation]) # joins all the lines in the list in a single string
```

Once the pre-processing is done, we can start building our language model. For simplicity, we will only build unigram, bigram, and trigram language models.

```
from nltk.util import ngrams
from nltk import word_tokenize, sent_tokenize

unigram=[]
bigram=[]
trigram=[]
tokenized_text = []

sents = nltk.sent_tokenize(file_p)

for sentence in sents:
    sentence = sentence.lower()
    sequence = word_tokenize(sentence)
    for word in sequence:
        if word ==".":
            sequence.remove(word)
        else:
            unigram.append(word)
    tokenized_text.append(sequence)
    bigram.extend(list(ngrams(sequence, 2)))
#unigram, bigram, trigram models are created
    trigram.extend(list(ngrams(sequence, 3)))

freq_uni = nltk.FreqDist(unigram)
freq_bi = nltk.FreqDist(bigram)
freq_tri = nltk.FreqDist(trigram)

print("5 most common unigrams:" + str(freq_uni.most_common(5)))
print("5 most common bigrams: " + str(freq_bi.most_common(5)))
print("5 most common trigrams: " + str(freq_tri.most_common(5)))
```

Here we used the FreqDist types of NLTK, which is used to represent probability distributions, to build our language models. The advantage of this is that functions from packages tend to be much faster than hand-rolling your own features.

2.1.1 Additive smoothing

Now let us do some smoothing, to avoid ending up with zeroed out probabilities. The following code computes, rather inefficiently, smoothed out language models for unigrams, bigrams, and trigrams. You will notice that it takes ages. You might even decide to take my word for it and cancel the computation half-way through (reminder: you can cancel a program running in the console with Ctrl + C).

```
ngrams_all = {1:[], 2:[], 3:[]}
for i in range(3):
    for each in tokenized_text:
        for j in ngrams(each, i+1):
            ngrams_all[i+1].append(j)

ngrams_voc = {1:set([]), 2:set([]), 3:set([])}
for i in range(3):
    for gram in ngrams_all[i+1]:
        if gram not in ngrams_voc[i+1]:
            ngrams_voc[i+1].add(gram)

total_ngrams = {1:-1, 2:-1, 3:-1}
```

```
total_voc = {1:-1, 2:-1, 3:-1}
for i in range(3):
    total_ngrams[i+1] = len(ngrams_all[i+1])
    total_voc[i+1] = len(ngrams_voc[i+1])

ngrams_prob = {1:[], 2:[], 3:[]}
for i in range(3):
    for ngram in ngrams_voc[i+1]:
        tlist = [ngram]
        tlist.append(ngrams_all[i+1].count(ngram))
        ngrams_prob[i+1].append(tlist)

for i in range(3):
    for ngram in ngrams_prob[i+1]:
        ngram[-1] = (ngram[-1]+1)/(total_ngrams[i+1]+total_voc[i+1])

for i in range(3):
    ngrams_prob[i+1] = sorted(ngrams_prob[i+1], key = lambda x:x[1], reverse =
    True)

print("Most common (1,2,3)-grams")
print ("Most common unigrams: " + str(ngrams_prob[1][:10]))
print ("Most common bigrams: " + str(ngrams_prob[2][:10]))
print ("Most common trigrams: " + str(ngrams_prob[3][:10]))
```

2.1.2 The LM function

But why would we waste our time rolling our own implementations when language modelling is such a common task that libraries, such as our beloved NLTK, implement it? Indeed, the nltk.lm[1] package contains everything we need to roll our own optimised language models with a fraction of the code (and much faster!). Let us build a nice bigram model.

```
import nltk, re, pprint, string
from collections import Counter
from nltk import word_tokenize, sent_tokenize
from nltk.util import ngrams, bigrams
from nltk.lm import MLE
from nltk.lm.preprocessing import pad_both_ends, padded_everygram_pipeline

custom_punctuation = string.punctuation + "’" + "-" + "‘" + "-"
custom_punctuation = custom_punctuation.replace(".", "")

with open("transcript.txt", encoding="utf8") as file:
    file_content = file.read()

file_nl_removed = file_content.replace("\n", " ")
file_p = "".join([char for char in file_nl_removed if char not in
    custom_punctuation])

n_param = 2

tokenized_text = nltk.word_tokenize(file_p)
padded_text = [list(pad_both_ends(tokenized_text, n=n_param))]
corpus, vocab = padded_everygram_pipeline(n_param, padded_text)

lm = MLE(n_param)
lm.fit(corpus, vocab)

# Create a Counter object to get the vocabulary
vocabulary = Counter(tokenized_text)

# Print the vocabulary
```

[1] http://www.nltk.org/api/nltk.lm.html

```
30 for word, count in vocabulary.items():
31     print(f"{word}: {count}")
```

You will notice that this is a lot shorter than what we did before, because NLTK is doing all of the heavy lifting. On line 17 we define `n_param` to 2. This has an impact on lines 20 and 21, which prepare the data by padding it with `<s>` and `</s>` tokens to model the start and end of sentences, and break the text in n-grams through the `everygram_pipeline` function (which does almost everything). We then instantiate a language model with the MLE function on line 23, before training it with a specific vocabulary list on line 24. Finally, we print the vocab object in lines 30 and 31 to make sure it is working correctly.

What about smoothing? NTLK.lm implements most of the smoothed models, and you just have to instantiate them instead of MLE. For example, you could use a Laplace-smoothed bigram language model as follows:

```
1 from nltk.lm import Laplace
2 lm = Laplace(2)
```

Have a look and investigate the official NLTK page on the lm module[2] and see the other smoothing models which are provided.

2.2 Generating language

Another advantage of the LM NLTK module is that it makes sampling from a language model very easy. Calling the generate function lets you provide the model with a number of tokens you want to sample, a text seed that you use for sampling, and a random seed for reproducibility (specifying a random seed makes sure that re-running the same program multiple times always uses the same "random" numbers). For example, the following code samples 20 tokens conditioned on the starting token "i".

```
1 print(lm.generate(20, text_seed=['i'], random_seed=1))
```

Testing on my machine, using the Berkeley restaurant corpus, the following result is returned:

```
1 ['dont', 'want', 'to', 'eat', 'lunch', 'id', 'like', 'to', 'drive', '.', 'thai',
      'food', 'only', 'about', 'le', 'bateau', 'ivre', 'tell', 'me', 'a']
```

"I don't want to eat lunch I'd like to drive" is hardly Shakespeare, but it is intelligible English. "Thai food only about le bateau ivre tell me a" however, is a bit more shaky! Now what would happen with a trigram language model? Let us rerun the same sampling process but on a language model trained with `n_param = 3`. The result, again on my machine, was the following:

```
1 ['dont', 'want', 'to', 'eat', 'lunch', 'im', 'willing', 'to', 'go', 'an', '
      ethopian', 'restaurant', 'in', 'albany', 'how', 'about', 'chezpanisse', 'that
      ', 'was', 'cheap']
```

It is a subjective thing, but this looks a lot more like a coherent piece of text than the bigram-generated text. In fact, the higher we put the n, the more realistic the text becomes, because we are more likely to sample larger pieces of the original text at once! Set `n_param` large enough, and you will be sampling entire sentences from the original text, making the language model a lot less useful!

3 Template-Based Natural Language Generation

Template-based natural language generation (NLG) refers to the process of producing textual output based on predefined patterns or templates. In this approach, the specific placeholders

[2] `http://www.nltk.org/api/nltk.lm.html`

in the template are filled in with the appropriate information to generate coherent sentences or paragraphs that accomplish a specific informational purpose. It is a rule-driven method where the structure of the output is largely predetermined, as opposed to more dynamic, data-driven approaches. While template-based NLG can be quick and effective for predictable and structured data, it lacks the adaptive flair of machine learning-based NLG techniques.

3.1 A simple gap-filling system

A gap-filling system is the most basic form of template-based NLG. Define the structure of your sentence, define the gaps to be filled, plug in the relevant data and you are done. It is very versatile, provided you fill in the correct data and it allows for generating simple text.

```
name = input("What is your name?")
country_of_origin = input("Where are you from?")

output = "Hello {v1}, I see that you are from {v2}.".format(v1=name, v2=
    country_of_origin)

print(output)
```

It does the job most of the time, but it is not necessarily the most interesting way of doing things, and it leads to a rather robotic-sounding output.

3.2 More complex templates

Let us go back to the essential steps of a template-based NLG system: content determination, document structuring, aggregation, lexical choice, referring expression generation, and realisation.

3.2.1 Content determination

What do we put in the text? If we want to talk about the weather, do we put every piece of weather-related information in it? Obviously not, some selection is required depending on the communicative goal. We provide a few ways you could implement a content determination system, with some toy examples in Python. Some of those examples will make more sense later in the course, when we will have seen text classification, information retrieval, etc.

Rule and Heuristics-Based Approaches For those types of approaches, we set explicit rules and heuristics to decide which pieces of data or facts should be included. This method often relies on if-then-else conditions that are predefined based on the application's requirements.

```
def rule_based_description(car):
    description = ""

    if car['type'] == 'sedan':
        description += "This is a family-friendly car."
    elif car['type'] == 'sports':
        description += "This car is designed for speed and performance."

    return description

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(rule_based_description(car_data))  # Output: This car is designed for speed
     and performance.
```

We can also apply heuristic rules derived from domain expertise to make slightly more complex decisions. For example, when generating weather reports, always including temperature and precipitation details could be a heuristic. Another heuristic could be, when describing a car, to a threshold for horsepower at 400 to differentiate decent cars from higher-quality cars.

```
def heuristic_description(car):
    description = "This is a {} car.".format(car['colour'])

    if car['horsepower'] > 400:
        description += " It has high horsepower, making it powerful."
    else:
        description += " It has decent performance for its range."

    return description

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(heuristic_description(car_data))  # Output: This is a red car. It has high
    horsepower, making it powerful.
```

Data-Driven Methods Using statistical or machine learning models that have been trained on large datasets to learn the significance of the pieces of content. These models can prioritise and select content based on patterns in the training data. Note that the following example will not run; it is just for illustration purposes.

```
from sklearn.dummy import DummyClassifier

# Example dataset: type, colour, horsepower mapped to a description label
X_train = [['sedan', 'blue', 100], ['sports', 'red', 500], ['SUV', 'black', 300]]
y_train = ['family', 'performance', 'utility']

# Train a dummy classifier for demonstration
model = DummyClassifier(strategy="most_frequent")
model.fit(X_train, y_train)

def data_driven_description(car):
    prediction = model.predict([[car['type'], car['colour'], car['horsepower']]])
    if prediction[0] == 'family':
        return "This is a family-friendly car."
    elif prediction[0] == 'performance':
        return "This car is designed for speed and performance."
    else:
        return "This is a versatile vehicle."

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(data_driven_description(car_data))  # Might output: This car is designed
    for speed and performance.
```

User preferences In this case, we adapt content determination based on user profiles or feedback. Over time, the system can learn to present information that is more closely aligned with the interests or preferences of the individual user.

```
user_preferences = {
    'colour_interest': 'red',
    'type_preference': 'sports'
}

def user_pref_description(car, preferences):
    description = ""
    if car['colour'] == preferences['colour_interest']:
        description += "This car's colour matches your preference!"
    if car['type'] == preferences['type_preference']:
        description += " It's the type of car you like!"
    return description

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(user_pref_description(car_data, user_preferences))  # Output: This car's
    colour matches your preference! It's the type of car you like!
```

Ontologies In the context of content determination and natural language generation, ontologies are structured frameworks that represent knowledge about a specific domain, categorising entities, and defining relationships between them. They facilitate the understanding and generation of content by providing a semantic hierarchy and context, ensuring that the generated language aligns with the structured knowledge of a particular subject or field. For example, in the medical domain, if discussing a particular disease, an ontology might suggest including symptoms, causes, and treatments.

```
# Let us assume a simplified ontology represented as a Python dictionary
car_ontology = {
    'sports': {'related_terms': ['speed', 'race', 'performance']},
    'SUV': {'related_terms': ['family', 'travel', 'space']}
}

def ontology_description(car, ontology):
    car_type = car['type']
    if car_type in ontology:
        related_terms = ', '.join(ontology[car_type]['related_terms'])
        return f"This {car_type} car is associated with {related_terms}."
    return "Unknown car type."

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(ontology_description(car_data, car_ontology))  # Output: This sports car is
    associated with speed, race, performance.
```

Query-Driven Determination When the NLG system is tied to a query system (e.g., a search engine, a personal assistant, or a chatbot), content determination can be directly influenced by the user's query. Only information directly responding or related to the query would be selected in order to make your text as concise as possible while still meeting the communicative goal.

```
user_query = "red powerful cars"

def query_driven_description(car, query):
    description = ""
    if 'red' in query and car['colour'] == 'red':
        description += "The car is red,"
    if 'powerful' in query and car['horsepower'] > 400:
        description += " and it's indeed powerful."
    return description

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(query_driven_description(car_data, user_query))  # Output: The car is red,
    and it's indeed powerful.
```

Interactive Systems Allow users to interactively guide the content determination process. This can be done by allowing users to specify constraints, ask follow-up questions, or fine-tune the scope of information.

```
def interactive_description(car):
    print("What do you want to know about the car?")
    attribute = input("Choose: type, colour, or horsepower: ")
    if attribute in car:
        return f"The car's {attribute} is {car[attribute]}."
    return "That's not a feature I know about."

car_data = {'type': 'sports', 'colour': 'red', 'horsepower': 500}
print(interactive_description(car_data))  # The user input will guide the
    description.
```

3.2.2 Document structuring

Document structuring focusses on how to organise the output of our system. Like the other steps, there are several ways to do so.

Rule, heuristics and templates We can use predefined rules or schemas as well as domain-specific heuristics that dictate the sequence or hierarchy of information presentation, based on the nature and relationship of the content. For example, in a research paper, starting with an introduction, followed by methods, results, and then discussion.

```
def rule_based_structure(city_data):
    report = ""

    if "historical_landmarks" in city_data:
        report += f"Historical Landmarks:\n{city_data['historical_landmarks']}\n\
    n"

    if "tourist_attractions" in city_data:
        report += f"Tourist Attractions:\n{city_data['tourist_attractions']}\n\n"

    if "restaurants" in city_data:
        report += f"Recommended Restaurants:\n{city_data['restaurants']}"

    return report

city_info = {
    'tourist_attractions': 'Central Park, City Museum',
    'restaurants': 'Bistro Cafe, Riverside Grill',
    'historical_landmarks': 'Old Clock Tower, Heritage Building'
}
print(rule_based_structure(city_info))
```

Alternatively, focusing more on domain heuristics.

```
def heuristic_structure(city_data):
    report = ""

    # Always start with an introduction if the city name is provided
    if "name" in city_data:
        report += f"Discover the wonders of {city_data['name']}!\n\n"

    # Always list tourist attractions before restaurants
    if "tourist_attractions" in city_data:
        report += f"Tourist Attractions:\n{city_data['tourist_attractions']}\n\n"

    if "restaurants" in city_data:
        report += f"Dine at:\n{city_data['restaurants']}."

    return report

print(heuristic_structure({'name': 'SampleCity', 'tourist_attractions': 'City
    Museum', 'restaurants': 'Bistro Cafe'}))
```

A similar approach would be to use preset templates with designated slots for different types of information. By filling these slots, the generated content adheres to a predictable structure.

```
TEMPLATE = """
Historical Landmarks:
{historical_landmarks}

Tourist Attractions:
{tourist_attractions}

Recommended Restaurants:
{restaurants}
```

```
"""

def template_structure(city_data):
    return TEMPLATE.format(**city_data)

print(template_structure(city_info))
```

Machine Learning We can also use machine learning and train models on structured documents to learn the typical patterns and sequences of this kind of content. Once trained, the model can predict the best structure for new information.

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Mock dataset: tourist attractions, restaurants, historical landmarks mapped to a sequence
X_train = np.array([
    [1, 0, 1],  # 1 if present, 0 if absent
    [1, 1, 0],
    [0, 1, 1]
])
y_train = [1, 2, 3]  # 1: historical-tourist-restaurant, 2: tourist-restaurant-historical, 3: restaurant-historical-tourist

model = DecisionTreeClassifier()
model.fit(X_train, y_train)

def ml_structure(city_data):
    sequence = model.predict([[
        1 if 'tourist_attractions' in city_data else 0,
        1 if 'restaurants' in city_data else 0,
        1 if 'historical_landmarks' in city_data else 0
    ]])[0]

    sections = {
        1: f"Historical Landmarks:\n{city_data.get('historical_landmarks', '')}",
        2: f"Tourist Attractions:\n{city_data.get('tourist_attractions', '')}",
        3: f"Recommended Restaurants:\n{city_data.get('restaurants', '')}"
    }

    return "\n\n".join([sections[i] for i in range(1, 4) if sequence == i])

print(ml_structure(city_info))
```

Discourse Models We can make use of models that understand the conventions of textual coherence, ensuring that the document follows recognised patterns of discourse, such as introducing a topic before delving into details.

```
# For simplicity, let us assume a basic discourse model that always starts with an introduction, followed by details, and ends with recommendations.
def discourse_structure(city_data):
    report = f"Welcome to the report on {city_data['name']}.\n\n"

    if "historical_landmarks" in city_data:
        report += f"Historical Landmarks:\n{city_data['historical_landmarks']}\n\n"

    if "tourist_attractions" in city_data:
        report += f"Tourist Attractions:\n{city_data['tourist_attractions']}\n\n"

    report += f"Lastly, when you're hungry, consider visiting:\n{city_data['restaurants']}."
```

```
    return report

print(discourse_structure({'name': 'SampleCity', 'historical_landmarks': '
    Heritage Building', 'tourist_attractions': 'City Museum', 'restaurants': '
    Bistro Cafe'}))
```

User-guided Structuring We could also allow users to define or tweak the structure according to their preferences, making the generated content customisable (and potentially better suited to user needs). This makes sense if we have a reason to think that users will have large differences in how they consume the information you provide.

```
def user_guided_structure(city_data):
    order = input("What order do you prefer? (options: historical_landmarks,
    tourist_attractions, restaurants, separated by commas): ").split(',')

    report = ""
    for section in order:
        if section in city_data:
            report += f"{section.replace('_', ' ').title()}:\n{city_data[section
    ]}\n\n"
    return report

# Note: This function is interactive and waits for user input.
# print(user_guided_structure(city_info))
```

Graph-based Methods We can represent content as nodes in a graph, where edges indicate relationships. Algorithms can then traverse the graph to determine an optimal structure to present the information.

```
# Using a simple adjacency list to represent a graph, where each section points
    to the next section.
graph_structure = {
    'intro': 'historical_landmarks',
    'historical_landmarks': 'tourist_attractions',
    'tourist_attractions': 'restaurants',
    'restaurants': None  # end
}

def graph_based_structure(city_data, graph):
    report = "Welcome to the city report.\n\n"
    section = 'intro'

    while graph[section]:
        report += f"{graph[section].replace('_', ' ').title()}:\n{city_data[graph
    [section]]}\n\n"
        section = graph[section]

    return report

print(graph_based_structure(city_info, graph_structure))
```

3.2.3 Aggregation

Aggregation in NLG involves combining separate pieces of information into coherent, grammatical, and often more concise sentences. This step helps ensure that the generated content is not overly verbose and is easy to understand.

Syntactic Aggregation This method merges pieces of information based on the syntactic structures of sentences, without necessarily considering deeper semantics or conceptual meanings. It focusses mainly on grammar and syntax to form coherent sentences. For example:

- "John plays football."
- "John plays basketball."

Those two statements would yield the following syntactic aggregation: "John plays football and basketball." In the above example, the aggregation is performed based on the common subject "John" and the verb "plays", combining the two objects "football" and "basketball" with the conjunction "and".

```
def syntactic_aggregate(sentence1, sentence2):
    # Split sentences to extract the subject, verb, and object
    subj1, verb1, obj1 = sentence1.split()
    subj2, verb2, obj2 = sentence2.split()

    # Check for common subject and verb
    if subj1 == subj2 and verb1 == verb2:
        return f"{subj1} {verb1} {obj1} and {obj2}."
    else:
        return None

sentence1 = "John plays football."
sentence2 = "John plays basketball."

print(syntactic_aggregate(sentence1[:-1], sentence2[:-1]))  # Removing trailing
    periods for simplicity
```

Conceptual Aggregation This method merges information on the basis of their underlying concepts or meanings. It takes into account the semantics of the sentences and can potentially reformulate the sentences to convey combined concepts more effectively. For example:

- "John is a software developer."
- "John creates mobile applications."

Those two statements would yield the following conceptual aggregation: "John is a software developer specialising in mobile applications." Here, the aggregation goes beyond just the syntactic structure. It recognises that "creating mobile applications" is a specialisation of being a "software developer", and it merges the two ideas into a single, more descriptive sentence. For this, let us use a simple heuristic approach where we have predefined patterns for conceptual relationships.

```
def conceptual_aggregate(sentence1, sentence2):
    # Recognise patterns for software development specialisations if "software
    developer" in sentence1 and "creates mobile applications" in sentence2:
        return "John is a software developer specialising in mobile applications.
    "
    return None

sentence1 = "John is a software developer."
sentence2 = "John creates mobile applications."

print(conceptual_aggregate(sentence1, sentence2))
```

Of course, this is an overly simplistic example. In practice, special data structures such as knowledge graphs can be used to keep track of conceptual relationships and aggregate them appropriately.

```python
class KnowledgeGraph:
    def __init__(self):
        # Basic graph structure: {entity: {relation: value}}
        self.graph = {}

    def add_relation(self, entity, relation, value):
        if entity not in self.graph:
            self.graph[entity] = {}
        self.graph[entity][relation] = value

    def get_relations(self, entity):
        return self.graph.get(entity, {})

def conceptual_aggregate(sentences):
    kg = KnowledgeGraph()

    # Extract relations and populate the knowledge graph
    for sentence in sentences:
        words = sentence.split()
        entity = words[0]
        if "software developer" in sentence:
            kg.add_relation(entity, 'profession', 'software developer')
        elif "specialises in" in sentence:
            specialisation = words[-2] + " " + words[-1].strip('.')
            kg.add_relation(entity, 'specialises', specialisation)

    # Aggregation Logic
    aggregated_statements = []
    for entity, relations in kg.graph.items():
        if relations.get('profession') == 'software developer' and 'mobile applications' in relations.get('specialises', ''):
            aggregated_statements.append(f"{entity} is a software developer specialising in mobile applications.")
        # ... add more rules as needed

    return aggregated_statements

sentences = ["John is a software developer.", "John specialises in mobile applications."]
print(conceptual_aggregate(sentences))
```

A more complex version of that would use an ontology to keep track of different software development specialisations (e.g., games, websites, AI, etc.) and produce appropriate aggregates. In practice, effective aggregation often combines both syntactic and conceptual methods to ensure grammatical correctness while preserving or even improving the conveyed meaning.

Approaches for aggregation There are different methods and strategies for implementing aggregation, depending on the nature of the data and the desired result.

- **Rule-based Aggregation** Define explicit rules to determine how and when to aggregate information. For example, if two sentences have the same subject, they might be combined using a conjunction.
- **Statistical Methods** Use statistical measures or patterns derived from large datasets to decide on aggregation. For example, certain sentence structures might be more common and can be preferred during aggregation.
- **Template-based Aggregation** Define templates where multiple pieces of information can be inserted into predefined slots in the template, creating a single aggregated sentence.
- **Machine Learning Approaches** Train models on examples of aggregated content, allowing the model to learn and predict how new content should be aggregated.

- **Semantic Methods** Understand the semantic relationships between pieces of information and use this understanding to meaningfully aggregate content.
- **User-driven Aggregation** Provide interfaces for users to manually or semiautomatically guide the aggregation process according to their preferences.

3.2.4 Lexical choice

Lexical Choice is one of the steps in natural language generation (NLG), where appropriate words or phrases are chosen to convey a specific meaning, sentiment, style, or tone. It deals with selecting the best words or word sequences to use in a generated sentence, considering the context and the intended message.

Factors influencing lexical choice:

- **Specificity**: Some information requires a more specific term, while others can be more generic. For instance, "canine" vs. "dog".
- **Formality**: The level of formality required can influence the choice of words. For example, "purchase" (formal) vs. "buy" (casual).
- **Tone** and **Sentiment**: Depending on whether the content is meant to be positive, negative, or neutral, the word choice can vary, e.g., "challenging" vs. "problematic".
- **Variability**: To avoid repetition or to engage the reader, different lexical items might be chosen for variety, even if they mean the same thing.
- **Audience Understanding**: Consideration of the audience's familiarity with terms is vital. For example, technical jargon might be appropriate for an expert audience, but not for laypeople.

Much like the other steps, there are several possible ways of building lexical choice into an NLG system.

Rule-based Methods Predefined rules or templates dictate which terms or phrases to use based on specific conditions or contexts.

```
def rule_based_choice(context):
    rules = {
        "greeting": "Hello",
        "farewell": "Goodbye",
        "thank": "Thank you"
    }
    return rules.get(context, "Unknown")

print(rule_based_choice("greeting"))  # Outputs: Hello
```

Thesaurus-based Methods Use a thesaurus or lexicon to find synonyms and select them according to context. In this example, we use the library PySynonyms, which you might need to install with `pip install py_synonyms`.

```
# pip install py_synonyms

from py_synonyms import Synonyms

def thesaurus_based_choice(word):
    synonyms = Synonyms(word).get_synonyms()
    return synonyms[0] if synonyms else word

print(thesaurus_based_choice("happy"))  # Might output: joyful
```

Statistical Methods Statistical language models can be used to determine the likelihood of the appropriateness of a word or phrase in a particular context. Feel free to go back to the previous section of this lab to have a more in-depth treatment of language models.

```
corpus = "I love programming. I love Python. I hate errors."

def bigram_choice(word):
    words = corpus.split()
    bigrams = [(words[i], words[i+1]) for i in range(len(words)-1)]
    choices = [b[1] for b in bigrams if b[0] == word]
    return max(set(choices), key=choices.count) if choices else None

print(bigram_choice("I"))  # Outputs: love
```

Machine Learning Approaches Deep learning models, like transformers, can be trained on vast amounts of text to make contextually appropriate lexical choices. For example, using the transformers library from Huggingface (warning: this involves installing a rather large library with `pip install transformers`, which might take a long time or not work at all on some computers). You might recognise the name of this language model, GPT-2, as being the ancestor of GPT-3, which is the basis of a famous chatbot.

```
# You might need to start by running pip install transformers
from transformers import GPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2-medium")
model = GPT2LMHeadModel.from_pretrained("gpt2-medium")

def ml_choice(partial_text):
    inputs = tokenizer.encode(partial_text, return_tensors="pt")
    outputs = model.generate(inputs, max_length=inputs.shape[1]+1, do_sample=
    False)
    decoded = tokenizer.decode(outputs[0])
    return decoded.split()[-1]  # Return the last word

print(ml_choice("Machine learning is a"))  # Might output: process
```

User Feedback We can also incorporate user feedback to refine lexical choices over time, especially in interactive NLG systems. This is particularly useful if you do not know enough about the user and want to leave them the choice to personalise their experience based on their preferences and level of expertise.

```
suggestions = {
    "food": ["apple", "banana", "cherry"],
    "vehicle": ["car", "bike", "bus"]
}

def feedback_choice(topic):
    print(f"Suggested word for {topic}: {suggestions[topic][0]}")
    feedback = input("Is this okay? (yes/no) ")
    if feedback == "yes":
        return suggestions[topic][0]
    else:
        # Just moving to the next suggestion for simplicity
        return suggestions[topic][1]

print(feedback_choice("food"))  # Outputs apple or banana based on feedback
```

3.2.5 Referring expression generation

Referring Expression Generation (REG) is a subtask of Natural Language Generation (NLG) that focusses on determining how to refer to entities in a way that they can be unambiguously

recognised by the listener or reader. When generating natural language, it is crucial to decide how to refer to objects or entities so that the intended audience can easily identify and distinguish them. REG concerns itself with selecting the most appropriate noun phrases to accomplish this.

For instance, if there are two apples on a table, one green and one red, instead of just saying "the apple," a proper referring expression would specify "the green apple" or "the red apple" to disambiguate between the two.

Here are some techniques and considerations for implementing REG:

- **Attribute Selection** Determine which attributes (like color, size, type, etc.) are essential to distinguish the entity from others in its context.
- **Definiteness** Decide whether to use definite ('the cat') or indefinite ('a cat') articles based on the familiarity or uniqueness of the entity.
- **Pronominal Reference** Decide whether it is appropriate to use a pronoun ("he," "she," "it") based on prior mentions and clarity.
- **Salience** Entities that have been recently mentioned or play a significant role in the discourse are more salient and might be referred to differently.

One of the most popular approaches to REG is the Incremental Algorithm (1) which attempts to find the shortest candidate description that makes an entity distinguishable from the others. Revisiting our previous example with software engineers would yield the following toy example.

```
developers = [
    {"name": "Alice", "specialization": "mobile apps", "years_of_experience": 5,
    "known_languages": ["Java", "Kotlin"], "company": "TechCorp"},
    {"name": "Bob", "specialization": "web development", "years_of_experience":
    3, "known_languages": ["JavaScript", "React"], "company": "WebSoft"},
    {"name": "Charlie", "specialization": "mobile apps", "years_of_experience":
    7, "known_languages": ["Swift"], "company": "TechCorp"},
    {"name": "Dana", "specialization": "embedded systems", "years_of_experience":
     10, "known_languages": ["C", "C++"], "company": "HardwareTech"},
]

def discriminate(entity, candidates, attribute):
    """Return true if the attribute discriminates the entity from the candidates.
    """
    value = entity[attribute]
    return sum(1 for e in candidates if e[attribute] == value) == 1

def incremental_reg(entity, developers):
    # Initially, all developers are candidates except the target entity
    candidates = [e for e in developers if e != entity]

    # Order of attributes to try (can be optimised based on context)
    attributes_order = ["specialisation", "years_of_experience", "known_languages
    ", "company"]

    description = []

    for attribute in attributes_order:
        if discriminate(entity, candidates, attribute):
            description.append(str(entity[attribute]))
            break  # The entity is now uniquely identified
        else:
            # If not discriminative, add to the description and reduce the
    candidate list
            description.append(str(entity[attribute]))
            candidates = [e for e in candidates if e[attribute] == entity[
    attribute]]

```

```
    return ' '.join(description)

developer_to_refer = {"name": "Charlie", "specialisation": "mobile apps", "
    years_of_experience": 7, "known_languages": ["Swift"], "company": "TechCorp"}
print(incremental_reg(developer_to_refer, developers))
# Might output: mobile apps 7 Swift, depending on the order of attributes and
    their distinctiveness.
```

3.2.6 Realisation

The realisation stage is the final step in the Natural Language Generation (NLG) process, where structured intermediate representations or abstract specifications are transformed into coherent, grammatically correct, and fluent natural language text.

Here are the main activities in the realisation stage:

- **Morphological Inflection** At this stage, the correct morphological forms of words are chosen based on the context. For instance, ensuring verbs agree with their subjects in number and tense.
- **Sentence Aggregation** This is about combining smaller sentences or clauses into more complex and fluent ones. For example, "John is a teacher." and "John teaches maths." might be aggregated as "John is a teacher who teaches maths."
- **Word Order Determination** In languages with flexible word order, the best order is chosen based on fluency and emphasis. For example, in English, we prefer "green big ball" over "big green ball."
- **Insertion of Function Words** Words like 'of,' 'is,' 'the,' etc., which might not be present in the abstract representation, are inserted for grammatical correctness.
- **Punctuation and Formatting** Proper punctuation marks are inserted, and the text is correctly formatted.

Now if you have paid attention, you might be thinking that Sentence Aggregation sounds a lot like the Aggregation stage we saw earlier. And that would not be completely incorrect. Although there is a subtle overlap, particularly around sentence aggregation, the two stages have different primary objectives: **Aggregation** is about **content synthesis and elimination of redundancy**, while **Realisation** is about **linguistic correctness and fluency**. In a well-structured NLG system, aggregation occurs before realisation. By the time you reach the realisation stage, the content should be well-organised, nonredundant, and ready to be transformed into fluent text.

In practice, how distinct these stages are might vary depending on the specific NLG system or framework. Some systems might blend these stages, especially when working with simpler tasks or when the intermediate representations are close to natural language. But in more sophisticated NLG systems, especially those dealing with complex data-to-text tasks, the distinction is clearer and helps in producing high-quality text.

Let us return to our running example by building an abstract representation of our software developers and their various attributes, and write a small program to generate a short bio for each of them.

```
developers = [
    {"subject": "Alice", "verb": "work", "duration": 5, "field": "mobile apps", "
    languages": ["Java", "Kotlin"], "company": "TechCorp", "location": "at"},
    {"subject": "Bob", "verb": "specialise", "duration": 3, "field": "web
    development", "languages": ["JavaScript", "React"], "company": "WebSoft", "
    location": "with"},
```

```
    {"subject": "Charlie", "verb": "work", "duration": 7, "field": "mobile apps",
     "languages": ["Swift"], "company": "TechCorp", "location": "at"},
]

def realise(developer):
    # Morphological Inflection
    if developer["verb"] == "work" and developer["duration"] > 1:
        verb_form = "has worked"
    elif developer["verb"] == "specialise" and developer["duration"] > 1:
        verb_form = "has specialised"
    else:
        verb_form = developer["verb"]

    # Handling list of languages
    if len(developer["languages"]) > 1:
        languages = ', '.join(developer["languages"][:-1]) + ' and ' + developer[
    "languages"][-1]
    else:
        languages = developer["languages"][0]

    # Constructing the sentence based on word choice and structure
    if developer["location"] == "at":
        sentence = f"{developer['subject']} {verb_form} for {developer['duration
    ']} years in {developer['field']} at {developer['company']} and is proficient
     in {languages}."
    else:
        sentence = f"{developer['subject']} {verb_form} for {developer['duration
    ']} years in {developer['field']} with {developer['company']} and knows {
    languages}."

    return sentence

for developer in developers:
    bio = realise(developer)
    print(bio)
```

The output of this program should be the following:

```
Alice has worked for 5 years in mobile apps at TechCorp and is proficient in Java
     and Kotlin.
Bob has specialised for 3 years in web development with WebSoft and knows
    JavaScript and React.
Charlie has worked for 7 years in mobile apps at TechCorp and is proficient in
    Swift.
```

As you can see, the realisation process ensures that verbs are inflected correctly based on duration (work to has worked, specialise to has specialised), and chooses between "proficient in" and "knows" based on the context.

3.3 Putting it all together

Looking at the previous sections in isolation might seem daunting, so let us try to put everything together with a simple NLG system.

3.3.1 Our starting dataset

We will be working with the details of two of our star developers: Alice and Bob.

```
developers = [
    {
        "name": "Alice",
        "role": "Senior Developer",
        "experience": 5,
```

```
        "education": {"degree": "MSc in Computer Science", "university": "
    Imperial College London"},
        "expertise": "mobile apps",
        "tech_stack": [{"language": "Java", "proficiency": "expert"}, {"language"
    : "Kotlin", "proficiency": "intermediate"}],
        "past_employers": ["MobiTech", "AppStart"],
        "current_project": {"name": "MobileBank", "description": "A next-gen
    mobile banking solution."},
        "accolades": ["Best Mobile App Developer 2022", "Tech Innovator Award
    2021"],
        "hobbies": ["photography", "cycling"]
    },
    {
        "name": "Bob",
        "role": "Frontend Developer",
        "experience": 3,
        "education": {"degree": "BSc in Software Engineering", "university": "
    University of Manchester"},
        "expertise": "web development",
        "tech_stack": [{"language": "JavaScript", "proficiency": "expert"}, {"
    language": "React", "proficiency": "advanced"}],
        "past_employers": ["WebGenius", "SiteMakers"],
        "current_project": {"name": "WebStore", "description": "A comprehensive e
    -commerce platform."},
        "accolades": ["Web Developer of the Year 2020"],
        "hobbies": ["painting", "chess"]
    }
]
```

3.3.2 The NLG pipeline

First, we start by implementing content determination. To keep it simple, we will just take the entire dataset. If we wanted to be more subtle, we could focus on a specific subset of the data. For example, if we wanted to craft the bios of the developers for a grant proposal, we would focus on specific achievements and technical skills.

```
def content_determination(dev):
    # For simplicity, we will select all attributes. But this step may involve
    more complex logic in real-world applications.
    return dev
```

We then move on to document structuring. Once again, we do not need to get too complicated: we simply organise the information we have from our previous stage in a way that makes it easier for the following stages to transform into usable natural language text.

```
def document_structuring(dev):
    return {
        "introduction": [dev['name'], dev['role'], dev['experience'], dev['
    education']],
        "skills": [dev['expertise'], dev['tech_stack']],
        "experience": [dev['past_employers'], dev['accolades']],
        "current_work": [dev['current_project']],
        "personal": [dev['hobbies']]
    }
```

The next stage is aggregation. We simply merge the information into concise and complete sentences. If we wanted to improve this, we could add alternative templates and some form of randomness to pick one of several alternative ways of stating the same sentences.

```
def aggregation(structured_dev):
    aggregated_data = {}

```

```
    aggregated_data["introduction"] = f"{structured_dev['introduction'][0]}, a {structured_dev['introduction'][1]} with {structured_dev['introduction'][2]} years of experience, graduated with a {structured_dev['introduction'][3]['degree']} from {structured_dev['introduction'][3]['university']}."

    tech_stack = ', '.join([f"{t['language']} ({t['proficiency']})" for t in structured_dev["skills"][1]])
    aggregated_data["skills"] = f"Specialising in {structured_dev['skills'][0]}, they are proficient in {tech_stack}."

    past_employers = ', '.join(structured_dev['experience'][0])
    accolades = ', '.join(structured_dev['experience'][1])
    aggregated_data["experience"] = f"Having previously worked at {past_employers}, they have been awarded with {accolades}."

    project = structured_dev['current_work'][0]
    aggregated_data["current_work"] = f"Currently, they're involved in the project '{project['name']}', which is described as: {project['description']}."

    hobbies = ', '.join(structured_dev['personal'][0][:-1]) + ' and ' + structured_dev['personal'][0][-1]
    aggregated_data["personal"] = f"Outside of professional life, their interests include {hobbies}."

    return aggregated_data
```

After this, we move on to the lexical choice. For the sake of illustration, we will simply add a synonym. You can refer to the previous section to see alternative ways of dealing with lexical choice to vary parameters such as the tone or complexity of the result.

```
def lexical_choice(aggregated_dev):
    text = aggregated_dev["skills"].replace("proficient", "skilled")
    aggregated_dev["skills"] = text
    return aggregated_dev
```

We then need to manage references within the text. Let us have some fun and deal with things like gender (defaulting to gender neutral if not provided), anaphoric references (ensuring that the developer's attributes are referred back to when relevant), and sensitivity to contextual details (past employers or specific hobbies).

```
def referring_expression(aggregated_dev):
    # Initial reference with full name
    intro = aggregated_dev["introduction"].replace(aggregated_dev['name'], f"{aggregated_dev['name']} (hereinafter referred to as the developer)")

    # Determine the pronoun based on gender
    pronoun = 'they'
    possessive_pronoun = 'their'
    if 'gender' in aggregated_dev:
        if aggregated_dev['gender'].lower() == 'male':
            pronoun = 'he'
            possessive_pronoun = 'his'
        elif aggregated_dev['gender'].lower() == 'female':
            pronoun = 'she'
            possessive_pronoun = 'her'

    # Replace subsequent direct name mentions with the pronoun
    skills = aggregated_dev["skills"].replace(aggregated_dev['name'], pronoun)
    experience = aggregated_dev["experience"].replace(aggregated_dev['name'], pronoun)
    current_work = aggregated_dev["current_work"].replace(aggregated_dev['name'], pronoun)

    # Adjust for verb conjugation with 'they'
```

```
    if pronoun == 'they':
        current_work = current_work.replace('has', 'have')

    # Referring back to past employers when discussing current projects, if
    relevant
    if any(employer in aggregated_dev["current_work"] for employer in
    aggregated_dev["experience"][0]):
        current_work = current_work.replace("Currently", f"Following {
    possessive_pronoun} tenure at {','.join(aggregated_dev['experience'][0])}")

    # Handling anaphoric references in personal info (e.g., hobbies)
    personal = aggregated_dev["personal"]
    if 'coding' in personal:
        personal = personal.replace('coding', 'this activity')  # Assuming a
    mention of coding as a hobby after introducing profession

    referred_content = {
        "introduction": intro,
        "skills": skills,
        "experience": experience,
        "current_work": current_work,
        "personal": personal
    }

    return referred_content
```

For the final stage, we simply join all the resulting words together and add a bit of flair to the descriptions. This is of course a lot simpler than the methods described in the previous section on realisation, but we are mostly trying to illustrate the different phases. Nowadays, complex NLG frameworks automate much of the details.

```
def realisation(lexical_dev):
    intro = lexical_dev["introduction"]

    # Transition to skills
    skills = "Delving into technicalities, " + lexical_dev["skills"]

    # Seamless connection between past experience and accolades
    experience = lexical_dev["experience"].replace(", they have", ". Additionally
    , they have")

    # Highlight the present role
    current_work = "On the present front, " + lexical_dev["current_work"]

    # Conclude with personal interests
    personal = "When away from the code, " + lexical_dev["personal"]

    return f"{intro} {skills} {experience} {current_work} {personal}"
```

We then need to call all the functions in the correct order, resulting in the following code.

```
for dev in developers:
    content = content_determination(dev)
    structured = document_structuring(content)
    aggregated = aggregation(structured)
    lexically_enriched = lexical_choice(aggregated)
    referred = referring_expression(lexically_enriched)
    final_output = realisation(referred)
    print(final_output, "\n")
```

4 Exercises

Exercises are basic tasks that can help you better understand the content. You are not expected to finish everything in the lab.

1. Print some statistics about your document: number of sentences, number of tokens, average number of tokens per sentence.
2. Print some statistics about your language models: 10 most common unigrams/bigrams/trigrams, 10 least common unigrams/bigrams/trigrams, and their probability of appearing in the text.
3. Using data from the Gutenberg library, build a language model that can write in the style of your favourite author.
4. Using data from the IMDB movie review dataset, build two language models: one to generate positive reviews, and one to generate negative reviews.

5 Tasks

Tasks are much more time-consuming than exercises and allow you to go much deeper into content mastery. We do not expect you to finish all of them in the lab, but they can be good toy problems to practice with.

1. [⋆] Build yourself a training and a test set, and use perplexity to compare the performance of a basic, non-smoothed language model with a language model using the smoothing function of your choice.
2. [⋆ ⋆] The LM module allows you to calculate the perplexity of your language model. One common use of language model perplexity is to find out the author of a text. Implement a plagiarism detector using language models (without googling "perplexity author detection").
3. [⋆ ⋆ ⋆] Can you make two language models talk to each other? Pair up with a friend (or with yourself) and train two language models on different datasets (there are several open-source books on the Gutenberg library, but you can find resources elsewhere if you want). Then, work together to make your language models speak to each other, in turns.
4. [⋆ ⋆ ⋆] Bootstrap your own dataset by building an object similar to the one described in 3.3.1, containing the details of your CV, and develop a basic NLG system which outputs your bio and adapts the language, structure, and amount of details based on the following criteria:
   - technical vs non-technical audience (vocabulary used);
   - complete bio vs shortened bio (amount of details);
   - purpose of the bio (job application, social event, dating profile).
5. [⋆ ⋆] Building on the previous task, embed your NLG system in an I/O loop as seen in the previous lab, and make it progressively build a profile of the user that can be used to generate the bio.

References

[1] Dale, R., and Reiter, E. Computational interpretations of the gricean maxims in the generation of referring expressions. *Cognitive science 19*, 2 (1995), 233–263.