Chatbots

Codecademy

Rule-Based Chatbots

Introduction to Natural Language Processing and Rule-Based Chatbots

Getting Started with Natural Language Processing

Intro to NLP

NLP, also known as *natural language processing*, is the field is at the intersection of linguistics, artificial intelligence, and computer science. The goal is to enable computers to interpret, analyze, and approximate the generation of human languages.

NLP can be conducted in several programming languages. However, Python has some of the most extensive open-source NLP libraries, including the Natural Language Toolkit or NLTK.

Text Preprocessing

Cleaning or preparation are crucial for many tasks, and NLP is no exception. ***Text preprocessing*** is usually the first step when faced with an NLP task.

Without preprocessing, your computer interprets "the", "The", and "<p>The" as entirely different words. There is a LOT that can be done here. Luckily Regex and NTLK will do most of it for us. Common tasks include:

- **Noise removal**: stripping text of formatting
- **Tokenization**: breaking text into individual words
- **Normalization**: cleaning text data in any other way

- **Stemming** is a blunt ace to chop off word prefixes and suffixes.
- **Lemmatization** is a scalpel to bring words down to their root forms.

```
# regex for removing punctuation!
import re
# nltk preprocessing magic
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
# grabbing a part of speech function:
from part_of_speech import get_part_of_speech

text = "So many squids are jumping out of suitcases these days that you can barely go anywhere without seeing one burst forth from a tightly packed valise. I went to the dentist the other day, and sure enough I saw an angry one jump out of my dentist's bag within minutes of arriving. She hardly even noticed."

cleaned = re.sub('\W+', ' ', text)
tokenized = word_tokenize(cleaned)

stemmer = PorterStemmer()
stemmed = [stemmer.stem(token) for token in tokenized]

## -- CHANGE these -- ##
lemmatizer = WordNetLemmatizer()
lemmatized                                                    =
[lemmatizer.lemmatize(token,get_part_of_speech(token))      for
token in tokenized]

print("Stemmed text:")
print(stemmed)
print("\nLemmatized text:")
print(lemmatized)
```

Parsing Text

Parsing is an NLP process concerned with segmenting text based on syntax.

NLTK tools for parsing:

- ***Part-of-speech tagging (POS tagging)*** identifies parts of speech
- ***Named entity recognition (NER)*** helps identify the proper nouns in a text
- ***Dependency grammar*** trees help you understand the relationship between the words in a sentence. Python library spaCy is helpful even though it isn't always perfect.
- ***Regex parsing***, using Python's re library, allows for a bit more nuance. When coupled with POS tagging, you can identify specific phrase chunks.

```
import spacy
from nltk import Tree
from squids import squids_text

dependency_parser = spacy.load('en')

parsed_squids = dependency_parser(squids_text)

# Assign my_sentence a new value:
my_sentence = "Hi my name is Prashant and I am learning
Chatbot building in Python."
my_parsed_sentence = dependency_parser(my_sentence)

def to_nltk_tree(node):
  if node.n_lefts + node.n_rights > 0:
      parsed_child_nodes = [to_nltk_tree(child) for child in
node.children]
    return Tree(node.orth_, parsed_child_nodes)
  else:
    return node.orth_

for sent in parsed_squids.sents:
  to_nltk_tree(sent.root).pretty_print()

for sent in my_parsed_sentence.sents:
 to_nltk_tree(sent.root).pretty_print()
```

Language Models: Bag-of-Words

How can we help a machine make sense of a bunch of word tokens? We can help computers make predictions about language by training a language model on a corpus (a bunch of example text).

Language models are probabilistic computer models of language. We build and use these models to figure out the likelihood that a given sound, letter, word, or phrase will be used. Once a model has been trained, it can be tested out on new texts.

One of the most common language models is the unigram model, a statistical language model commonly known as ***bag-of-words***. As name suggest, bag-of-words does not have much order to its chaos! What it does have is a tally count of each instance for each word.

Bag-of-words can be an excellent way of looking at language when you want to make predictions concerning topic or sentiment of a text. When grammar and word order are irrelevant, this is probably a good model to use.

```
# importing regex and nltk
import re, nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
# importing Counter to get word counts for bag of words
from collections import Counter
# importing a passage from Through the Looking Glass
from looking_glass import looking_glass_text
# importing part-of-speech function for lemmatization
from part_of_speech import get_part_of_speech

# Change text to another string:
# text = looking_glass_text
text = "Let's see what it does to this sample input. Will it
do something amazing or just some silly showoff!"

cleaned = re.sub('\W+', ' ', text).lower()
tokenized = word_tokenize(cleaned)

stop_words = stopwords.words('english')
filtered = [word for word in tokenized if word not in
stop_words]
```

```
normalizer = WordNetLemmatizer()
normalized = [normalizer.lemmatize(token,
get_part_of_speech(token)) for token in filtered]
# Comment out the print statement below
print(normalized)

# Define bag_of_looking_glass_words & print:
bag_of_looking_glass_words = Counter(normalized)
print(bag_of_looking_glass_words)
```

Language Models: N-Gram and NLM

Unlike bag-of-words, the ***n-gram*** model considers a sequence of some number (n) units and calculates the probability of each unit in a body of language given the preceding sequence of length n. Because of this, n-gram probabilities with larger n values can be impressive at language prediction.

Problems:

- How can your language model make sense of test words that it has never encountered before. A tactic known as language smoothing can help adjust probabilities for unknown words, but it isn't always ideal.
- For a model that more accurately predicts human language patterns, you want n to be as large as possible. That way, you will have more natural sounding language, right? Well, as the sequence length grows, the number of examples of each sequence within your training corpus shrinks. With too few examples, there won't be enough data to make many predictions.

Enter ***neural language models (NLMs)!*** Much recent work within NLP has involved developing and training neural networks to approximate the approach our human brain take towards language.This deep learning approach allows computers a much more adaptive tack to processing human language. Common NLMs include LSTMs and transfer models.

```
import nltk, re
from nltk.tokenize import word_tokenize
# importing ngrams module from nltk
from nltk.util import ngrams
from collections import Counter
```

```
from looking_glass import looking_glass_full_text

cleaned = re.sub('\W+', ' ', looking_glass_full_text).lower()
tokenized = word_tokenize(cleaned)

# Change the n value to 2:
looking_glass_bigrams = ngrams(tokenized, 2)
looking_glass_bigrams_frequency                                    =
Counter(looking_glass_bigrams)

# Change the n value to 3:
looking_glass_trigrams = ngrams(tokenized, 3)
looking_glass_trigrams_frequency                                   =
Counter(looking_glass_trigrams)

# Change the n value to a number greater than 3:
looking_glass_ngrams = ngrams(tokenized, 15)
looking_glass_ngrams_frequency = Counter(looking_glass_ngrams)

print("Looking Glass Bigrams:")
print(looking_glass_bigrams_frequency.most_common(10))

print("\nLooking Glass Trigrams:")
print(looking_glass_trigrams_frequency.most_common(10))

print("\nLooking Glass n-grams:")
print(looking_glass_ngrams_frequency.most_common(10))
```

Topic Models

Topic modeling is an area of NLP dedicated to uncovering latent, or hidden, topics within a body of language.

A common technique is to deprioritize the most common words and prioritize less frequently used terms as topics in a process known as ***term frequent-inverse document frequency (tf-idf)***. The python libraries *gensim* and *sklearn* have modules to handle tf-idf.

The next step in topic modeling is often ***latent Dirichelt allocation (LDA)***. LDA is a statistical model that takes your documents and determines which words keep popping up together in the same context.

For visualizing your newly minted topics, ***word2vec*** is a great techniques to have up your sleeve. Word2vec can map out your topic model results spatially as vectors so that similarly used words are closer together. This word-to-vector mapping is known as a word embedding.

```
import nltk, re
from sherlock_holmes import bohemia_ch1, bohemia_ch2, bohemia_ch3, boscombe_ch1, boscombe_ch2, boscombe_ch3
from preprocessing import preprocess_text
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# preparing the text
corpus = [bohemia_ch1, bohemia_ch2, bohemia_ch3, boscombe_ch1, boscombe_ch2, boscombe_ch3]
preprocessed_corpus = [preprocess_text(chapter) for chapter in corpus]

# Update stop_list:
stop_list = ["say", "see", "holmes", "shall", "know", "quite", "well", "think", "look", "little"]
# filtering topics for stop words
def filter_out_stop_words(corpus):
  no_stops_corpus = []
  for chapter in corpus:
          no_stops_chapter = " ".join([word for word in chapter.split(" ") if word not in stop_list])
     no_stops_corpus.append(no_stops_chapter)
  return no_stops_corpus
filtered_for_stops = filter_out_stop_words(preprocessed_corpus)

# creating the bag of words model
bag_of_words_creator = CountVectorizer()
bag_of_words = bag_of_words_creator.fit_transform(filtered_for_stops)

# creating the tf-idf model
tfidf_creator = TfidfVectorizer(min_df = 0.2)
tfidf = tfidf_creator.fit_transform(preprocessed_corpus)
```

```
# creating the bag of words LDA model
lda_bag_of_words_creator                                    =
LatentDirichletAllocation(learning_method='online',
n_components=10)
lda_bag_of_words                                            =
lda_bag_of_words_creator.fit_transform(bag_of_words)

# creating the tf-idf LDA model
lda_tfidf_creator                                           =
LatentDirichletAllocation(learning_method='online',
n_components=10)
lda_tfidf = lda_tfidf_creator.fit_transform(tfidf)

print("~~~ Topics found by bag of words LDA ~~~")
for               topic_id,               topic               in
enumerate(lda_bag_of_words_creator.components_):
  message = "Topic #{}: ".format(topic_id + 1)
                            message             +=              "
".join([bag_of_words_creator.get_feature_names()[i]  for  i  in
topic.argsort()[:-5 :-1]])
  print(message)

print("\n\n~~~ Topics found by tf-idf LDA ~~~")
for               topic_id,               topic               in
enumerate(lda_tfidf_creator.components_):
  message = "Topic #{}: ".format(topic_id + 1)
    message  +=  "  ".join([tfidf_creator.get_feature_names()[i]
for i in topic.argsort()[:-5 :-1]])
  print(message)
```

Text Similarity

Addressing ***text similarity*** - including spelling correction - is a major challenge within natural language processing.

Addressing word similarity and misspelling for spellcheck or autocorrect often involves considering the ***Levenshtein distance*** or minimal edit distance between two words. The distance is calculated through the minimum number of insertions, deletions, and substitutions that would need to occur for one word to become another.

Phonetic similarity is also a major challenge within speech recognition. English - speaking humans can easily tell from context whether someone said “euthanasia” or “youth in Asia” but it’s a far more challenging task for a machine! More advanced autocorrect and spelling correction technology additionally considers key distance on a keyboard and ***phonetic similarity*** (how much two words or phrases sound the same).

It’s also helpful to find out if texts are the same to guard against plagiarism, which we can identify through ***lexical similarity*** (the degree to which texts use the same vocabulary and phrases).

Meanwhile, ***semantic similarity*** (the degree to which documents contain similar meaning or topics) is useful when you want to find an article or book similar to one you recently finished.

```
import nltk
# NLTK has a built-in function
# to check Levenshtein distance:
from nltk.metrics import edit_distance

def print_levenshtein(string1, string2):
    print("The Levenshtein distance from '{0}' to '{1}' is {2}!".format(string1, string2, edit_distance(string1, string2)))

# Check the distance between
# any two words here!
print_levenshtein("fart", "target")

# Assign passing strings here:
three_away_from_code = "barcode"

two_away_from_chunk = "bunk"

print_levenshtein("code", three_away_from_code)
print_levenshtein("chunk", two_away_from_chunk)
```

Language Prediction & Text Generation

Language prediction is an application of NLP concerned with predicting text given preceding text. Autosuggest, autocomplete, and suggest replies are common forms of language prediction.

First step to language prediction is picking a language model. Bag of words alone is generally not a great model for language prediction; no matter what the preceding word was, you will just get one of the most commonly used words from your training corpus.

If you go n-gram route, you will most likely rely on ***Markov chains*** to predict the statistical likelihood of each following word based on the training corpus. Markov chains are memory-less and make statistical predictions based entirely on the current n-gram on hand.

A more advanced approach, using a neural language model, is the ***Long Short Term Memory (LSTM)*** model. LSTM uses deep learning with a network of artificial "cells" that manage memory, making them better suited for text prediction than traditional neural networks.

```
import nltk, re, random
from nltk.tokenize import word_tokenize
from collections import defaultdict, deque
from document1 import training_doc1
from document2 import training_doc2
from document3 import training_doc3

class MarkovChain:
  def __init__(self):
    self.lookup_dict = defaultdict(list)
    self._seeded = False
    self.__seed_me()

  def __seed_me(self, rand_seed=None):
    if self._seeded is not True:
      try:
        if rand_seed is not None:
          random.seed(rand_seed)
        else:
          random.seed()
        self._seeded = True
      except NotImplementedError:
```

```
        self._seeded = False

    def add_document(self, str):
        preprocessed_list = self._preprocess(str)
        pairs = self.__generate_tuple_keys(preprocessed_list)
        for pair in pairs:
            self.lookup_dict[pair[0]].append(pair[1])

    def _preprocess(self, str):
        cleaned = re.sub(r'\W+', ' ', str).lower()
        tokenized = word_tokenize(cleaned)
        return tokenized

    def __generate_tuple_keys(self, data):
        if len(data) < 1:
            return

        for i in range(len(data) - 1):
            yield [ data[i], data[i + 1] ]

    def generate_text(self, max_length=50):
        context = deque()
        output = []
        if len(self.lookup_dict) > 0:
            self.__seed_me(rand_seed=len(self.lookup_dict))
            chain_head = [list(self.lookup_dict)[0]]
            context.extend(chain_head)

            while len(output) < (max_length - 1):
                next_choices = self.lookup_dict[context[-1]]
                if len(next_choices) > 0:
                    next_word = random.choice(next_choices)
                    context.append(next_word)
                    output.append(context.popleft())
                else:
                    break
            output.extend(list(context))
        return " ".join(output)

my_markov = MarkovChain()
my_markov.add_document(training_doc1)
my_markov.add_document(training_doc2)
my_markov.add_document(training_doc3)
```

```
generated_text = my_markov.generate_text()
print(generated_text)
```

Advanced NLP Topics

- ***Naive Bayes classifiers*** are supervised machine learning algorithms that leverage a probabilistic theorem to make predictions and classifications. They are widely used for sentiment analysis (determining whether a given block of language expresses negative or positive feelings) and spam filtering.
- We've made enormous gains in ***machine translation***, but even the most advanced translation software using neural networks and LSTM still has far to go in accurately translating between languages.
- Some of the most life-altering applications of NLP are focused on improving ***language accessibility*** for people with disabilities. Text-to-speech functionality and speech recognition have improved rapidly thanks to neural language models, making digital spaces far more accessible places.
- NLP can also be used to detect bias in writing and speech.

```
from reviews import counter, training_counts
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

# Add your review:
review = "This lesson is great introudction to advanced NLP applications and the project cited is also wonderful."
review_counts = counter.transform([review])

classifier = MultinomialNB()
training_labels = [0] * 1000 + [1] * 1000

classifier.fit(training_counts, training_labels)

neg = (classifier.predict_proba(review_counts)[0][0] * 100).round()
pos = (classifier.predict_proba(review_counts)[0][1] * 100).round()

if pos > 50:
```

```
    print("Thank you for your positive review!")
elif neg > 50:
    print("We're sorry this hasn't been the best possible lesson
for you! We're always looking to improve.")
else:
    print("Naive Bayes cannot determine if this is negative or
positive. Thank you or we're sorry?")


print("\nAccording to our trained Naive Bayes classifier, the
probability that your review was negative was {0}% and the
probability it was positive was {1}%.".format(neg, pos))
```

Challenges and Considerations

When working with NLP, we have several important considerations to take into account:

- Different NLP tasks may be more or less difficult in different languages. Because so many NLP tools are built by and for English speakers, these tools may lag behind in processing other languages. The tools may also be programmed with cultural and linguistic biases specific to English speakers.
- English itself is not a homogeneous body. It varies by person, dialect, and by many sociolinguistic factors. When we build and train NLP tools, are we only building them for one type of English speaker?
- You can have the best intentions and still in advertently program a bigoted tool. While NLP can limit bias, it can also propagate bias. As an NLP developer, it’s important to consider biases, both within your code and within the training corpus. A machine will learn the same biases you teach it, whether intentionally or unintentionally.
- As you become someone who builds tools with NLP, it’s vital to take into account your user’s privacy. There are many powerful NLP tools that come head-to-head with privacy concerns. Who is collecting your data? How much data is being collected and what data do these companies plan to do with your data?

```
from reviews import counter, training_counts
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
```

```
# Add your review:
review = "It was lit!"
review_counts = counter.transform([review])

classifier = MultinomialNB()
training_labels = [0] * 1000 + [1] * 1000

classifier.fit(training_counts, training_labels)

neg    =    (classifier.predict_proba(review_counts)[0][0]    *
100).round()
pos    =    (classifier.predict_proba(review_counts)[0][1]    *
100).round()

if pos > 50:
  print("Naive Bayes classifies this as positive.")
elif neg > 50:
  print("Naive Bayes classifies this as negative.")
else:
   print("Naive Bayes cannot determine if this is negative or
positive.")


print("\nAccording to our trained Naive Bayes classifier, the
probability that your review was negative was {0}% and the
probability it was positive was {1}%.".format(neg, pos))
```