NLP

into to nlp

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human languages.

- Real world application
  - contextual advertisement
  - email spam
  - social media filteration
  - chatbot
- Common task
  - Text document classification
  - Sentiment analysis
  - info retrieval
  - parts of speech tagging
  - Language detection and machine translation
  - Conversational agents
  - knowledge graph and QA system
  - Text summarization
  - Topic modelling
  - Text generation
  - Spell checking and grammer correction
  - speech to text

Approaches to nlp

- Heuristic approach's
  - Regular Expressions
  - Wordnet
  - open mind common sense
- ML approach's
  - Naive bayes
  - Logistic Regression
  - SVM
  - LDA
  - Hidden markov model
- DL approach's
  - RNN
  - LSTM
  - GRU
  - Transformer
  - Auto encoders

Challenges in nlp

- Ambiguity — i saw the on the beach with my binoculars
- Contextual words — i ran to the store because we ran out of milk -> word ran
- Slang — pulling your leg
- irony, sarcasm
- spelling error
- creativity
- Diversity

NLP Pipeline

- **Basics**
  - Set of steps followed to build an end to end NLP software
- **steps**
  - Data acquisition
  - Text preparation
  - Feature engineering
  - Modellling
  - Deployment
- **Points to remember**
  - its not universal
  - dl pipelines are slightly different
  - pipelines is not linear
- **Data acquisition**
  - There could be 3 scenario
    - you have the data
      - we have data in csv file
      - we have data in database
        - ask data engineer
      - we have very less data
        - use data augmentataion
    - other have the data
      - public data set
      - web scraping
      - fetch data from api
    - No one have the data
- **Text preparation**
  - Text cleanup
    - Remove spelling mistakes
    - like html tag cleaning
    - change emojis to machine understandable text
  - Basic pre processing
    - basics
      - Tokenization
        - sentence tokenizations
        - word tokenizations
    - optionals
      - stop word removal
      - stemming
      - Removing digits , punctuations
      - changing text to upper case or lower case
      - language detection(used if we are making translator)
  - Advance pre-processing
    - Parts of speech taging
    - Parsing
    - core reference resolution
- **Feature engineering**
  - encoding the words to numbers
- **Modellling**
  - Model building
    - Heuristic approaches
      - used if
        - data is very less
    - ML algos
      - used if
        - data is less
      - sometime ml algo used with heuristic based approach
    - DL algos
      - used if
        - data is large
      - here tansfer learning also used
    - Cloud api
  - Evaluation
    - intrinsic evalution
    - extrinsic evalution
- **Deployment**
  - Deployment
  - monitoring
  - model update

Text Pre-processing

Lowercasing

- column: `df['cl'] = df['cl'].str.lower()`

Removing

To remove something we use regular expression

use this tool to make regular expression

- html tags

```
import re
def remove_html_tags(text):
    pattern = re.compile('<.*?>')
    return pattern.sub(r'', text)
```

```
df['cl'] = df['cl'].apply(remove_html_tags)
```

- urls

```
def remove_url(text):
    pattern = re.compile(r'https?://\S+|www\.\S+')
    return pattern.sub(r'', text)
```

```
df['review'] = df['review'].apply(remove_url)
```

Removing Punctuations

```
!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~
```

Punctuations

- code

```
exclude = string.punctuation
```

```
def remove_punc(text):
    return text.translate(str.maketrans('', '', exclude))
```

```
df['cl'] = df['cl'].apply(remove_punc)
```

Chat word treatment

spelling corrections

```
from textblob import TextBlob

def spelling_correction(text):
    textBlb = TextBlob(incorrect_text)
    return textBlb.correct().string
```

```
df['cl'] = df['cl'].apply(spelling_correction)
```

Removing stop words

If you want to do parts of speech tagging then do not remove stop words

- Code

```
from nltk.corpus import stopwords

def remove_stopwords(text):
    new_text = []

    for word in text.split():
        if word in stopwords.words('english'):
            new_text.append('')
        else:
            new_text.append(word)
    x = new_text[:]
    new_text.clear()
    return " ".join(x)
```

```
df['cl'] = df['cl'].apply(remove_stopwords)
```

Handling Emojis

- Ways
  - Remove

```
import re
def remove_emoji(text):
    emoji_pattern = re.compile("["
                               u"\U0001F600-\U0001F64F"
    # emoticons
                               u"\U0001F300-\U0001F5FF"
    # symbols & pictographs
                               u"\U0001F680-\U0001F6FF"
    # transport & map symbols
                               u"\U0001F1E0-\U0001F1FF"
    # flags (iOS)
                               u"\U00002702-\U000027B0"
                               u"\U000024C2-\U0001F251"
                               "]+", flags=re.UNICODE)
    return emoji_pattern.sub(r'', text)
```

```
df['cl'] = df['cl'].apply(remove_emoji)
```

  - Replace it with its meaning

```
import emoji
import re

def replace_emojis_with_meanings(text):
    # First, demojize the text to get :emoji_name: format
    demojized_text = emoji.demojize(text)

    # Use regex to find all :emoji_name: patterns and replace them with readable text
    def convert_match(match):
        # Get the matched emoji name without colons
        emoji_name = match.group(1)
        # Replace underscores with spaces
        readable_name = emoji_name.replace('_', ' ')
        return readable_name

    # Replace all :emoji_name: with readable text
    readable_text = re.sub(r':([a-zA-Z0-9_]+):', convert_match, demojized_text)

    return readable_text
```

```
df['cl'] = df['cl'].apply(replace_emojis_with_meanings)
```

Stemming and lemmatization

changing walking , walks , walked --> walk

Stemming is the process of reducing inflection in words to their root form such as mapping a group of words to the same stem even if the stem itself is not a valid word in the language

that why when you need to show text to user we do not use stemming -> in that case we use lemmatization(lemmatization also get the root of word but it make sure the root word is valid in language)

lemmatization is slow as comapre to stemming

- Code

```
from nltk.stem.porter import PorterStemmer

ps = PorterStemmer()
def stem_words(text):
    return " ".join([ps.stem(word) for word in text.split()])
```

```
df['cl'] = df['cl'].apply(PorterStemmer)
```

- lemmatization code

```
import nltk
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

sentence = "He was running and eating at same time. He has bad habit of swimming after playing long hours in the Sun."
punctuations="?:!.,;"
sentence_words = nltk.word_tokenize(sentence)
for word in sentence_words:
    if word in punctuations:
        sentence_words.remove(word)

sentence_words
print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
    print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.lemmatize(word,pos='v')))
```

POS Tagging

Part-of-speech (POS) tagging is a process in natural language processing (NLP) that involves labeling the words in a sentence with their corresponding part-of-speech categories, such as nouns, verbs, adjectives, adverbs, etc.

- Theory
  - machine learning techniques used for POS tagging
    - Hidden Markov Models or Conditional Random Fields
- used where
  - name entity recognition
  - chatbots
  - word sense diaambiguation
    - Example: i ran to store because i ran out of milk-> here ran has 2 meaning
- code

```
!pip install spacy
```

```
import spacy
```

```
nlp = spacy.load('en_core_web_sm')
```

```
doc = nlp(u"I will google about facebook")
```

- doc.text: 'I will google about facebook'
- doc[-1]: facebook
- doc[2].pos_: 'VERB'
- doc[2].tag_: 'VB'
- spacy.explain('VB'): 'verb, base form'

Tokenization

breaking text document into smaller parts it can be word , sentence , etc (mostly words)

- Code

```
import spacy
nlp = spacy.load('en_core_web_sm')
```

```
doc4 = nlp("I am going to visit delhi!")
```

```
for token in doc4:
    print(token)
```

Text Representation

Techniques
- ohe
- bow
- ngrams
- tf-idf
- Uni-grams
- Bi-grams

Basic terminology
- Corpus (c) — when we combine all the words it became corpus
- vocabulary (v) — number of unique words present in the data set
- Document (d) — single row text data is known as document
- word (w)

- Example - data
  - then corpus
    - people watch campusx write comment
  - vocabulary (v) = 5

OHE
- Convert each word into V(vocabulary) dimension
- not used
- adv
  - easy to implement
  - sparsity
- div
  - Fix size of input is required
  - at prediction if some new word comes -> it will break the model= This problem is called oov (out of vocabulary)
  - No capturing of semantic meaning
- For example
  - D1 = people watch campusx
  - people watch campusx write comment
  - vocabulary (v) = 5

  - ohe

  - ohe representation of D1

Bag of words
- works best for text classification problems

- Working
  - First we create vocabulary

  - then for each row we store the occurance of words
- Core idea
  - A category of text -> have same words with same frequency
  - order of words does not matter here
  - words context does not matter but it still able to capture the semantic meaning
- code
  - from sklearn.feature_extraction.text import CountVectorizer
  - vectorizer = CountVectorizer()
  - X_tarin_bow = vectorizer.fit_transform(df['c1']).toarray()
- adv
  - simple and intuitive
  - oov resolved — it only look for the words present in vocabulary in test data set, so even if we encounter new words in test data set, bag of words just simply ignore it
  - able to capture semantic meaning(a little bit) not very much) — here each data point act as a vector , similar vector will closer to each other
  - no need of fix size of input and output
- div
  - sparsity
  - ordering is ignored
  - Semantic meaning is not reliable
    - Example — i like food / i dont like food
    - because of commonality of words these 2 text's vector's should be very close but they hold different meaning
- hyper parameters
  - binary — bool, default=False
    - If True, all non zero counts are set to 1.
    - binary = true works better for sentiment analysis
  - max_features — default=None , int
    - used to remove rare words
    - Ex -> max_features = 10 --> then it will take only 10 words which has the maximum frequency in the corpus

N-grams
- here we make vocabulary out of multiple continues words
- Rest will be similar to Bag of words
- Note Bag of words -> is a special case N-grams -> if we give ngram_range(1,1) = unigrams = bag of words
- Types of N-grams
  - Unigrams (1-gram) — Single words. Example: "I love NLP" -> ["I", "love", "NLP"]
  - Bigrams (2-gram) — Pairs of consecutive words. Example: "I love NLP" -> ["I love", "love NLP"]
  - Trigrams (3-gram) — Triplets of consecutive words. Example: "I love NLP" -> ["I love NLP"]
  - N-grams (n-gram) — Any sequence of n consecutive words.
- code
  - from sklearn.feature_extraction.text import CountVectorizer
  - vectorizer = CountVectorizer(ngram_range=(2, 2), max_features=5000)
  - X_train_ngm = vectorizer.fit_transform(df['c1']).toarray()
- Hyper parameter — ngram_range
  - default (1,1)
  - if we take (1,2) — this mean include bigram and unigrams
    - in vocabulary include single word and binary words pair also
- adv
  - capture semantic meaning better then one and bag of words
- div
  - sparsity
  - not efficient for large data set
  - semantic meaning not able to capture it properly

Tf-idf Term Frequency-Inverse Document Frequency
- Give different weightage to the words
- Here everything remain same as bag of words but instead of taking frequency here we calculate TF-IDF for each words.
- Components of TF-IDF
  - Term Frequency (TF) — Measures how frequently a term appears in a document.

  - Inverse Document Frequency (IDF) — Measures how important a term is

    - why we taking log here
      - For example we are using this formula (NLP) , now we have a very rare word
      - then IDF = (1000/1) = 1000
      - now tf will always in between 0 to 1
      - now even if we do TF * idf , because idf is very high tf have no significant impact
  - TF-IDF = TF * IDF
- div
  - sparsity
  - semantic meaning not able to capture it
- code
  - from sklearn.feature_extraction.text import TfidfVectorizer
  - vectorizer = TfidfVectorizer()
  - X_train_tfidf = vectorizer.fit_transform(df['c1']).toarray()

Word2vec
- deep learning based word embedding technique
- adv
  - here we can capture word semantic meaning
  - when we represent words as vector these vector will be dence vector and small in dimensions
- intuition
  - word2vec use shallow neural network to create feature(like 300) and then assign each word a numeric value for each feature and on the basis of these numeric value , it decide the similarity between word
  - once we get the vector of each words , when we are applying this method on sentence -> ie sum up the vector of each word in the sentence
- way to use
  - use a pre trained model — if your data and pre trained data have 80% same vocabulary then use it
  - train a word2vec model on your data set — if you have enough data , then this will out perform the pre trained models
- Types
  - CBOW (continuous Bag of word) — The model predicts the target word from its context words. For example, given the context "The cat sat on the ___", it predicts "mat".
    - used when data is small
  - Skip Gram — The model predicts the context words from a target word. For example, given the word "cat", it predicts words like "the", "sat", "on".
    - used when data is large
- own models
  - code
    - Link
    - pip install gensim
    - import gensim
    - from nltk import sent_tokenize from gensim.utils import simple_preprocess

    - data set
    - from gensim.models import Word2Vec
    - model creation
      - CBOW — model = Word2Vec(vector_size=300, window=5, min_count=1, sg=0)
      - Skip-gram — model = Word2Vec(vector_size=300, window=5, min_count=1, sg=1)
      - # `vector_size` is the size of the word vectors
      - # `window` is the maximum distance between the current and predicted word within a sentence
      - # `min_count` ignores all words with total frequency lower than this
      - # `sg` defines the training algorithm, 0 for CBOW, 1 for Skip-gram
    - tokenization

```
story = []
for doc in df['review']:
    raw_sent = sent_tokenize(doc)
    for sent in raw_sent:
        story.append(simple_preprocess(sent))
```

    - creating vocabulary out of data — model.build_vocab(story)
    - model.train(story, total_examples=model.corpus_count, epochs=model.epochs)
    - def document_vector(doc): # remove out-of-vocabulary words doc = [word for word in doc.split() if word in model.wv.index_to_key] return np.mean(model.wv[doc], axis=0)
    - X = [] for doc in tqdm(df['review'].values): X.append(document_vector(doc))
    - X=np.array(X)
- Pretrain models
  - code
    - import gensim
    - from gensim.models import word2vec,KeyedVectors
    - !pip install wget
    - !wget -c "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.gz"
    - model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin.gz',binary=True,limit=500000)