1 Module 18: Natural Language Processing

Contents

There are two sub-fields of NLP:

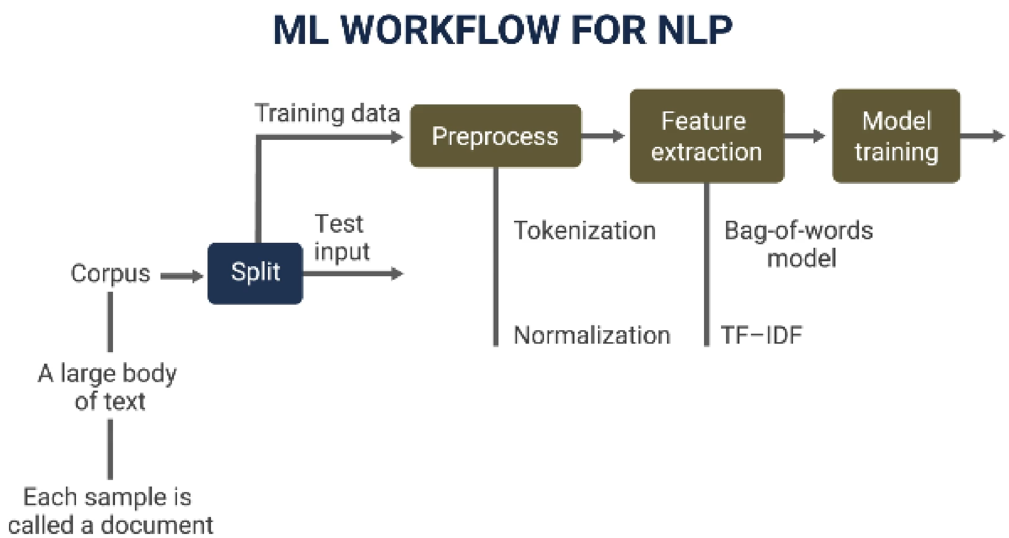
- Natural Language Understanding (**NLU**): analyze sentence meaning from syntactic/semantic elements
- Natural Language Generation (**NLG**): create human language from data input (or convert text to voice TTS)

This module focuses only on NLU.

1.1 NLP Pipeline

Main hurdles:

- Preprocessing: Text tokenization (splitting into grammatical units, aka tokens); Normalization (reduces tokens to core set cpaturing important information)
- Feature Extraction: turning words into a dataset more amenable to model training (bag of words, TF-IDF)



Industry standard is the Natural Language Toolkit.
Download datasets with `nltk.download()`

1.2 Preprocessing

Goals: convert text to numbers, data dimensionality reduction

Tokenization: Splitting a stringtext into an array of words. `nltk.word_tokenize(str)`

Normalization: Standardize text

- Lower casing `[word.lower() for word in words]`
- Convert numbers to words/ remove numbers
- Remove punctuation, accents, special characters
- Remove whitespace
- Expand abbreviations
- Remove words (stop words: am, is, are, etc.)
- Autocorrecting words ('fask' → 'task', etc.)

.
Grammatization: Speech designation i.e. "The" → Determiner, "movie" → Noun, "was" → Past-tense verb, etc.

`nltk.pos_tag(list[str])` and decode tagnames with `nltk.help.upenn_tagset(str)`
e.g. `str = 'PRP\$'` → 'pronoun, possessive'

Named entities: Nouns that denote particular items i.e. Organization, People, Location, Date, Time, etc.

Identifying named entities.

```
named_entities = []
for t in nltk.ne_chunk(words_pos):
    if hasattr(t, 'label'):
        e_name = ' '.join(c[0] for c in t.leaves())
        e_type = t.label()
        named_entities.append((e_name, e_type))

print(named_entities)
-----> [('Report', 'ORGANIZATION'),
        ('Tom Cruise', 'PERSON'),
        ('Steven Spielberg', 'PERSON'), etc.]
```

Removing PERSON specifically.

```
words_nonames = words.copy()
for ne in named_entities:
    if ne[1]=="PERSON":
        for name in nltk.word_tokenize(ne[0]):
            words_nonames.remove(name)
```

Stop words

```
from nltk.corpus import stopwords
stop_words = stopwords.words('english')

words = [w for w in words if not w in stop_words]
```

Now we've distilled text without losing core words.

Stemming & Lemmatization replace groups of words with their root forms e.g. 'joyful' → 'joy'

Stemming

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()

A = ['joy', 'joyful', 'joyfully', 'joyous', 'gees']
[stemmer.stem(w) for w in A]
----> ['joy', 'joy', 'joy', 'joyou', 'gee']
```

All of these processes can always produce nonwords such as 'joyou' or 'gee'.

Lemmatization: when grammatical coherence must be preserved

```python
from nlk.stem import WordNetLemmatizer
lemma = WordNetLemmatizer()

A = ['joy', 'joyful', 'joyous', 'geese']
[lemma.lemmatize(w) for w in A]
----> ['joy', 'joyful', 'joyous', 'goose']
```

1.3 Feature Extraction

In order of complexity: Bag of Words → TDF-IDF → Full-word vectorization (not covered)

Bag of Words: each word is a feature; count number of occurrences

		features				output
	Tokens	'enjoy'	'disappoint'	'bore'	'great'	sentiment
documents	['enjoy','disappoint','bore','great']	1	1	1	1	1
	['bore','disappoint','bore','bore']	0	1	3	0	0
	['great','great','enjoy']	1	0	0	2	1

Note that Bag of Words does not track informativeness of words e.g. ”I had my car cleaned” and ”I had cleaned my car” are functionally the same.

TDF-IDF: quantifies the usefulness of each token

$$tfidf(t,d) = td(t,d) \times idf(t)$$

Term frequency: $tf(t,d) = \frac{\text{number of times that t occurs in d}}{numberofwordsind}$ (higher weight given to words with more frequency in the document)

Inverse Document Frequency: $idf(t) = -log\left(\frac{\text{number of documents that contain t}}{\text{total number of documents}}\right)$ (suppresses words that are present in every document; amplifies ”rare” words)

1.4 Naive Bayes

For input features x of length M, the outputs y are integers of a class $\in 1, K$.

For the multinomial logistic regression $h(x)$, there are $K-1$ tuning parameters β with $M+1$ entries.

$$h_p(x) = \arg_K \max \hat{P}_\beta(Y=K|X=x) \tag{1}$$

$$\hat{P}_\beta(Y=K|X=x) = \begin{cases} \frac{1}{1+\sum_{k=1}^{K-1}\exp(-\beta_k \cdot x)} & k=k \\ \frac{\exp(-\beta_k \cdot x)}{1+\sum_{k=1}^{K-1}\exp(-\beta_k \cdot x)} & \text{otherwise} \end{cases}$$

Naive Bayes similarly solves Eqn. 1, but it estimates $P(Y=K|X=x)$ differently by using Bayes' rule. This also means that features X are assumed to be independent of one another.

$$\begin{aligned} h(x) &= \arg_k \max P(Y=K|X=x) \\ &= \arg_k \max P(Y=K|X=x) \frac{P(X=x|Y=K)P(Y=K)}{P(X=x)} \\ &= \arg_k \max P(Y=K|X=x) P(X=x|Y=K)P(Y=K) \end{aligned} \tag{2}$$

Shorthand $P(X=x|Y=K) \rightarrow P(x,k)$

$$\begin{aligned} P(x,k) &= P(x_1, x_2, x_3, \dots, x_m|k) \\ &= P(x_1, x_2, x_3, \dots, x_{m-1}|x_m, k)P(x_m|k) \\ &= P(x_1, x_2, x_3, \dots, x_{m-2}|x_{m-1}x_m, k)P(x_{m-1}|x_m, k)P(x_m|k) \\ &= P(x_1|x_1 \dots x_m, k)P(x_2|x_3 \dots x_m, k) \dots P(x_m|k) \end{aligned}$$

And assuming all x_i are independent given class K, $P(x_i|x_j,k) = P(x_i|k)$ and therefore

$$P(x|k) = P(k)\prod_{i=1}^{M} P(x_i|k) \tag{3}$$

If we didn't do this, we would need to sample each x_i distribution some $N \geq 1$ times; by assuming independence and using Eqn. e, we simplify to estimating M separate 1-D distributions.

From Eqn. 2, apply logarithm to convert product to sum.

$$\begin{aligned} h(x) &= \arg_k \max P(k) \prod_{i=1}^{M} P(x_i|k) \\ &= \arg_k \max \left[\log P(k) + \sum_{i=1}^{M} \log P(x_i|k) \right] \\ &= \arg_k \max \left[\log P_k + \sum_{i=1}^{M} x_i \log P_{k_i} \right] \\ &= \arg_k \max \left[\beta_{k_0} + \beta_k^T x \right] \end{aligned} \tag{4}$$

Which is simply a linear regression.

$$\begin{aligned} \hat{p}_{k_i} &= \frac{N_{i_k}}{\sum_{i=1}^{M} N_{i_k}} \\ \hat{p}_k &= \frac{\sum_{i=1}^{M} N_{i_k}}{N} = \frac{\sum_{i=1}^{M} N_{i_k}}{\sum_{k=1}^{k} \sum_{i=1}^{M} N_{i_k}} \end{aligned} \tag{5}$$

If there exist any words present in the training data but not in the test data, then $N_{i_k} = 0$ and $\log 0 = -\infty$. Therefore amend to $\log \frac{\sum_{i=1}^{M} N_{i_k} + \alpha}{N + \alpha_M}$ with the denominator term added to keep values adding to 1. This is called Laplace smoothing.

1.5 Advanced NLP methods

Named-entity recognition (NER)

Uses unstructured data to extract entities (people, places, objects, monetary value, etc.) and restrict ML tasks (text/sentiment analysis) to the entities assigned as important. Each industry comain has its own NER capability to maximize precision.

Semantic Search

Uses ML to understand *intent* behind a query, search data for the answer & respond. THe unique feature is that *intent* is not dependent on keywords. The algorithm uses users' search history, past purchases, online behavior, location, etc. to identify relevant information. Therefore, larger the knowledge graph == more accurate.

Sentiment Analysis

Associate sentiment with parts of categorized data (entities, topics, aspects) and return aggregate positive, negative or neutral score. Widespread in consumer/employee insights & social media sentiment analysis.

Text summarizations
Decompose large documents into a dictionary of commonly occuring words, sort & categorize, select & aggregate most-common words.
Aspect-based granularity
Identify relevant entities from gathered data for sentiment analysis & extract relevant information.
Question-answering systems
Think customer-service. Extract info from big data to answer queries.

1.6 Training Evaluation

Intrinsic eval: intermediate objectives e.g. performance of NLP on specific subtask
Extrinsic eval: review of performance on final objective

Intrinsic eval is important for guiding efforts:
- Confusion matrix, RMSE, F1 Score
- Area under the curve (AUC)
- Perplexity
- Metric for evaluation of translation with explicit ordering (METEOR)
- Recall-oriented understudy for gisting evaluation (ROUGE)

1.7 Twitter example

```
import nltk
nltk.download('twitter_samples')
from nltk.corpus import twitter_samples

tweets_pos = twitter_samples.strings('positive_tweets.json') # 5000
tweets_neg = twitter_samples.strings('negative_tweets.json') # 5000
all_tweets = tweets_pos + tweets_neg # This contains labels

# Train/test splits
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(all_tweets, all_labels,
↪  test_size = .25)

def preprocess_tweets(X_train):
    """
    Preprocesses a tweet set. Always a good idea to collect into a single
↪   function so identical preprocessing can be applied later.
    """
```

```
        # Tokenize --  nltk provides one especially for twitter
        from nltk.tokenize import TweetTokenizer
        tokenizer = TweetTokenizer(preserve_case = False, \
        strip_handles = True, \ # Twitter handles removed
        reduce_len = True)
        X_train_tok = [tokenizer.tokenize(tweet) for tweet in X_train]

        # Normalization

        # Stop words
        from ntlk.corpus import stopwords
        swords = stopwords.words('english')

        X_train_tok_nostop = []
        for tweet in X_train_tok:
            words = [word for word in tweet if word not in swords]
            X_train_tok_nostop.append(words)

        # Stemming
        from nltk.stem import PorterStemmer
        stemmer = PorterStemmer()

        X_train_tok_nostop_stem = []
        for tweet in X_train_tok_nostop:
            words = [stemmer.stem(word) for word in tweet]
            X_train_tok_nostop_stem.append(words)

    return X_train_tok_nostop_stem

X_train_pp = preprocess_tweets(X_train)

# Bag of words model

# Check number of words
unique_words = set()
for tweet in X_train_pp:
    unique_words.update(tweet)
len(unique_words) # 10,255 unique words; number of features

# Build training set
bow_matrix_train = np.zeros( (len(X_train_pp), len(unique_words)) )
for i, tweet in enumerate(X_train_pp):
    for word in tweet:
        if word in unique_words:
            bow_matrix_train[i,unique_words==word] += 1
```

```
# Resulting matrix is very sparse; just .08% of values are non-zero (typical)
# Note: SciPy has sparse matrix class to handle these more efficiently

# Apply Naive Bayes method

# Manually
alpha = 1 # LaPlace smoothing factor

ind_pos = (y_train == 1) # indices of positive tweets

# Word count for each term i in positive/negative tweets
N_pos_i = bow_matrix_train[ ind_pos,:].sum(axis=0)
N_neg_i = bow_matrix_train[-ind_pos,:].sum(axis=0)

# Total word count in pos/neg tweets
N_pos = N_pos_i.sum()
N_neg = N_neg_i.sum()

# Coefficients for pos/neg classes
logp_pos_i = np.log( (N_pos_i + alpha) / (N_pos + M*alpha) )
logp_neg_i = np.log( (N_neg_i + alpha) / (N_neg + M*alpha) )

# Intercepts for pos/neg classes
logp_pos = np.log( N_pos / (N_pos + N_neg) )
logp_neg = np.log( N_neg / (N_pos + N_neg) )

# Sci-kit learn
from sklearn.naive_bayes import MulinomialNB
naive_bayes = MultinomialNB(alpha = 1., fit_prior=False).fit(bow_matrix_train,
↪  y_train)

# In which case
logp_pos_i == naive_bayes.feature_log_prob[1]

# Evaluating the model
X_test_pp = preprocess_tweets(X_test)
bow_matrix_test = np.zeros( (len(X_test_pp), len(unique_words) )
for i, tweet in enumerate(X_test_pp):
     for word in tweet:
        if word in unique_words:
            bow_matrix_test[i, unique_words==word] += 1

naive_bayes.score(bow_matrix_test, y_test) # 99.8%

# Can also test with logreg
```

```python
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression().fit(bow_matrix_train, y_train)
lr.score(bow_matrix_test, y_test) # 99.992%
```