Data Science

Pre-processing text data

- Cleaning up the text data is necessary to highlight attributes that we are going to want our model to pick up on. Cleaning (or pre-processing) the data typically consists of number of steps:

  1. **Remove punctuation**
  2. **Converting text to lowercase**
  3. **Tokenization**
  4. **Remove stop-words**
  5. **Lemmatization /Stemming**
  6. **Vectorization**
  7. **Feature Engineering**

Pre-processing text data

- Cleaning up the text data is necessary to highlight attributes that we are going to want our model to pick up on. Cleaning (or pre-processing) the data typically consists of number of steps:

1. Remove punctuation
2. Converting text to lowercase
3. Tokenization
4. Remove stop-words
5. Lemmatization /Stemming
6. **Vectorization**
7. Feature Engineering

Vectorizing

- **Vectorizing** : The process that we use to convert text to a form that Python and a machine learning model can understand .
- It is defined as the process of **encoding text as integers to create feature vectors**.
- **A feature vector** is an **n-dimensional vector of numerical features that represent some object**.
- So in our context, that means we'll be taking

  an individual text message and converting it to a numeric

  vector that represents that text message.

Vectorizing

- There are many vectorization techniques, we will focus on the three widely used vectorization techniques:
  - Count vectorization
  - N-Grams.
  - Term frequency - inverse document frequency (TF-IDF)

- These methods will generate very **similar document-term matrices** where there's one line per document(**SMS in our case**,then the columns will represent each word or potentially a combination of words,
- The main difference between the three is what's in the **actual cells of the matrix**.

Vectorizing

Tf-Idf example

```python
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
texts = [

    "NLP is intresting field ", "NLP is not intresting field",
  "did not like NLP", "i like it", "good one"
]
# using default tokenizer in TfidfVectorizer
tfidf = TfidfVectorizer()
#tfidf = TfidfVectorizer(min_df=2, max_df=0.5, ngram_range=(1, 2))
features = tfidf.fit_transform(texts)
pd.DataFrame(
    features.todense(),
    columns=tfidf.get_feature_names()
)
```

	did	field	good	intresting	is	it	like	nlp	not	one
0	0.000000	0.520646	0.000000	0.520646	0.520646	0.000000	0.000000	0.432183	0.000000	0.000000
1	0.000000	0.461804	0.000000	0.461804	0.461804	0.000000	0.000000	0.383339	0.461804	0.000000
2	0.602985	0.000000	0.000000	0.000000	0.000000	0.000000	0.486484	0.403826	0.486484	0.000000
3	0.000000	0.000000	0.000000	0.000000	0.000000	0.778283	0.627914	0.000000	0.000000	0.000000
4	0.000000	0.000000	0.707107	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.707107

1- Count vectorization

- **Count Vectorizer**: The most straightforward one, it counts the number of times a token shows up in the document and uses this value as its weight.
- We will **create a matrix that only has numeric entries** counting how many times **each word appears in each text message**. The machine learning algorithm understands these counts. So if it sees a one or a two or a three in a cell, then **that model can start to correlate that with whatever we're trying to predict**
- A document term matrix is generated where each cell is the count corresponding to the message type indicating the number of times a word appears in a document, also known as the **term frequency**.
- The document term matrix is a set of dummy variables that indicates if a particular word appears in the document. **A column is dedicated to each word in the corpus.**
- This means, if a particular word appears many times in **spam or ham message** ,then the particular word has a high predictive power of determining if the message is a spam or ham .

Count vectorization- Document term matrix

- NLP is interesting , NLP is good
- Don’t like NLP
- good subject

NLP	is	interesting	Don’t	like	good	subject
2	2	1	0	0	1	0
1	0	0	1	1	0	0
0	0	0	0	0	1	0

Count vectorization- Document term matrix

- If we select only 10 messages and assume that we have 200 unique words
- The final result will be like this
- **10 rows with 200 columns**
- In our case , we have **5568 message * unique words**

Msg_id	Free	Meeting	…….	Label
1	5	0		Spam
2	1	2		Ham
3	0	5		Ham
4	2	0		Spam
5	1	3		Ham
6	0	3		Ham
7	0	2		Ham
8	4	0		Spam
9	0	3		Ham
10	2	0		Spam

Count vectorization- Document term matrix

Msg_id	Free	Meeting	....	Label
1	5	0		Spam
4	2	0		Spam
8	4	0		Spam
10	2	0		Spam

Msg_id	Free	Meeting	....	Label
2	1	2		Ham
3	0	5		Ham
5	1	3		Ham
6	0	3		Ham
7	0	2		Ham
9	0	3		Ham

Count vectorization- Document term matrix

- If we select only 10 messages and assume that we have 200 unique words
- The final result will be like this
- **10 rows with 200 columns**
- In our case , we have **5568 message * unique words**

Msg_id	Free	Meeting	…….	Label
1	5	0		Spam
2	1	2		Ham
3	0	5		Ham
4	2	0		Spam
5	1	3		Ham
6	0	3		Ham
7	0	2		Ham
8	4	0		Spam
9	0	3		Ham
10	2	0		Spam

Sparse Matrix

- when you have a matrix in which a very high percent of the **entries are zero**, instead of storing all **these zeros in the full matrix**, which would make it extremely inefficient, it'll just be converted to **only storing the locations and the values of the non-zero elements,** which is much more efficient for storage.
- **Sparse Matrix**: A matrix in which most entries are 0. In the interest of efficient storage, a sparse matrix will be stored by only storing the locations of the non-zero elements._

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

N-gram vectorizing

- The n-grams process creates a **document-term matrix** like we saw before. Now we still have **one row per text message** and we still have counts that occupy the individual cells but instead of the columns representing single terms ,here ;**all combinations of adjacent words of length** and in your text.
- As an example, let's use the string **NLP is an interesting topic**.

n	Name	Tokens
2	bigram	["nlp is", "is an", "an interesting", "interesting topic"]
3	trigram	["nlp is an", "is an interesting", "an interesting topic"]
4	four-gram	["nlp is an interesting", "is an interesting topic"]

N-gram vectorizing(2,2)

```
sentences = ["good movie", "not a good movie", "did not like", "i like it"]
ngram_vect = CountVectorizer(ngram_range=(2,2))
X_counts = ngram_vect.fit_transform(sentences)
print(X_counts.shape)
X_counts_df = pd.DataFrame(X_counts.toarray())
X_counts_df.columns = ngram_vect.get_feature_names()
X_counts_df
```

```
(4, 5)
```

	did not	good movie	like it	not good	not like
0	0	1	0	0	0
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	1	0	0

N-gram vectorizing(1,2)

```
sentences = ["good movie", "not a good movie", "did not like", "i like it"]
ngram_vect = CountVectorizer(ngram_range=(1,2))
X_counts = ngram_vect.fit_transform(sentences)
print(X_counts.shape)
X_counts_df = pd.DataFrame(X_counts.toarray())
X_counts_df.columns = ngram_vect.get_feature_names()
X_counts_df
```

```
(4, 11)
```

	did	did not	good	good movie	it	like	like it	movie	not	not good	not like
0	0	0	1	1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0	1	1	1	0
2	1	1	0	0	0	1	0	0	1	0	1
3	0	0	0	0	1	1	1	0	0	0	0

N-gram vectorizing(1,3)

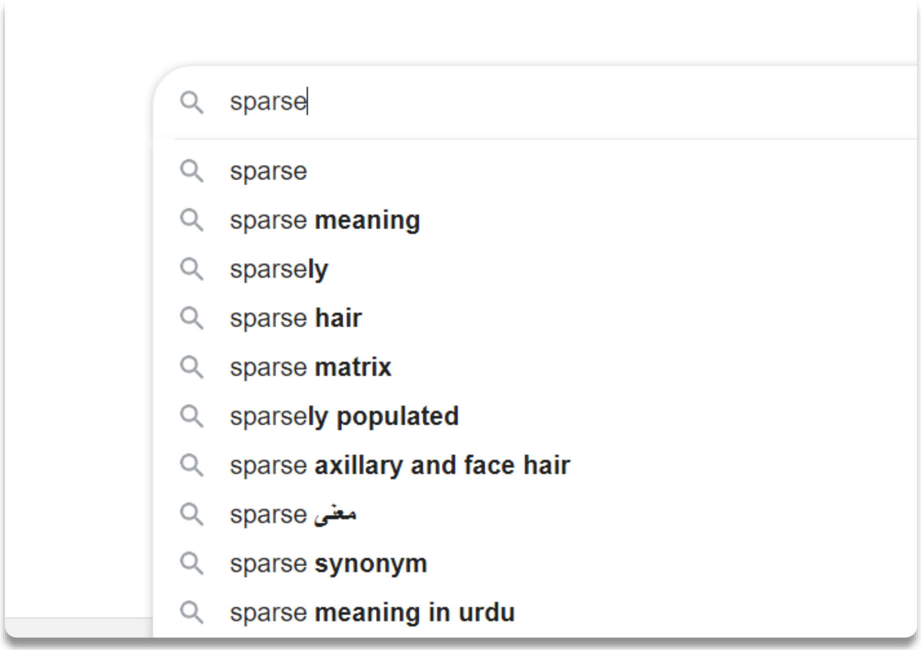
```
sentences = ["good movie", "not a good movie", "did not like", "i like it"]
ngram_vect = CountVectorizer(ngram_range=(1,3))
X_counts = ngram_vect.fit_transform(sentences)
print(X_counts.shape)
X_counts_df = pd.DataFrame(X_counts.toarray())
X_counts_df.columns = ngram_vect.get_feature_names()
X_counts_df
```

```
(4, 13)
```

	did	did not	did not like	good	good movie	it	like	like it	movie	not	not good	not good movie	not lik
0	0	0	0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	1	1	0	0	0	1	1	1	1	0
2	1	1	1	0	0	0	1	0	0	1	0	0	1
3	0	0	0	0	0	1	1	1	0	0	0	0	0

N-gram vectorizing

- When you use n-grams there's usually an **optimal n value or range that will yield the best performance.**
- The intuition here is that bi-grams and tri-grams can capture **contextual information compared to just unigrams.** Rather than only seeing one word at a time,
- The trade-off is between the number of N values. Choosing a smaller N value, may not be sufficient enough to **provide the most useful information**. Whereas choosing a high N value, will yield a huge matrix with loads of features. N-gram may be powerful, but it needs a little more care.
- **Google's auto complete uses an n-grams like approach, try to type and test.**



Term frequency - inverse document frequency (TF-IDF)

- **TF-IDF** creates a **document term matrix**, where there's still **one row per text message** and the columns still represent single unique terms.
- But instead of the cells representing the **count**, the cells represent a **weighting** that's meant to identify how important a word is to an individual text message.
- This formula lays out how this weighting is determined.
- **weighting** = **TF*IDF**

$$W_{x,y} = tf_{x,y} * \log\left(\frac{N}{df_x}\right)$$

- **TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).**

- **IDF(t) = log_(Total number of documents / Number of documents with term t in it).**

$W_{x,y}$= Word x within document y

$tf_{x,y}$= frequency of x in y

df_x= number of documents containing x

N=total number of documents

TF-IDF- How to Compute:

- Typically, the **TF-IDF weight** is composed by two terms:
  - **Term Frequency (TF)**, the **number of times a word appears** in a document, **divided by the total number of words in that document**. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (the total number of terms in the document) as a way of normalization:
    - TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).
  - **Inverse Document Frequency (IDF)**, computed as the **logarithm of the number of the documents** in the corpus divided by the number of documents where the specific term appears. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:

    IDF(t) = log_(Total number of documents / Number of documents with term t in it).

TF-IDF- How to Compute:

- **Example:**

- Consider a document containing **10** words wherein the word "**NLP**" appears **3** times.
- Now, assume we have **1000** documents, and the word "**NLP**" appears in **10** of these documents

- **TF("NLP") = (Number of times term "NLP" appears in a document) /** (Total number of terms in the document).
  - **TF("NLP") =(3 / 10) = 0.3.**
- **IDF("NLP") = log(Total number of documents /** Number of documents with term "**NLP**" in it).
  - **IDF("NLP") = log(1000 /10)** = **= log(100)=2 .**

- Thus, the **Tf-idf** weight is the product of these quantities: **0.3 * 2 = 0. 6**.

TF-IDF- Vs N-Gram:

```
entences = ["good movie", "not a good movie", "did not like", "i like it"]
ngram_vect = CountVectorizer(ngram_range=(1,3))
X_counts = ngram_vect.fit_transform(sentences)
print(X_counts.shape)
X_counts_df = pd.DataFrame(X_counts.toarray())
X_counts_df.columns = ngram_vect.get_feature_names()
X_counts_df
```

```
(4, 13)
```

	did	did not	did not like	good	good movie	it	like	like it	movie	not	not good	not good movie	not like
0	0	0	0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	1	1	0	0	0	1	1	1	1	0
2	1	1	1	0	0	0	1	0	0	1	0	0	1

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
sentences = ["good movie", "not a good movie", "did not like", "i like it"]
#tfidf = TfidfVectorizer(min_df=1)
tfidf = TfidfVectorizer( ngram_range=(1, 3))
features = tfidf.fit_transform(sentences)
print(features.shape)
pd.DataFrame(
    features.todense(),
    columns=tfidf.get_feature_names()
)
```

```
(4, 13)
```

	did	did not	did not like	good	good movie	it	like	like it	movie	not	not good	not good movie	not like
0	0.000000	0.000000	0.000000	0.577350	0.577350	0.000000	0.000000	0.000000	0.577350	0.000000	0.00000	0.00000	0.000000
1	0.000000	0.000000	0.000000	0.372225	0.372225	0.000000	0.000000	0.000000	0.372225	0.372225	0.47212	0.47212	0.000000
2	0.436719	0.436719	0.436719	0.000000	0.000000	0.000000	0.344315	0.000000	0.000000	0.344315	0.00000	0.00000	0.436719
3	0.000000	0.000000	0.000000	0.000000	0.000000	0.617614	0.486934	0.617614	0.000000	0.000000	0.00000	0.00000	0.000000

Course Contents

https://dair.ai/notebooks/nlp/2020/03/19/nlp_basics_tokenization_segmentation.html