ial-nlp-feature-extraction-methods

July 8, 2023

1 Essential_NLP_Feature_Extraction_Methods.ipynb

NLP feature extraction methods are techniques used to convert raw text data into numerical representations that can be processed by machine learning models. These methods aim to capture the meaningful information and patterns in text data.

Here are some essential NLP feature extraction methods:

1. Label Encoding
2. One Hot Encoding
3. Count Vectorization

- TF-IDF Vectorizer

- Bag Of Words (BOW)

4. `Word Embedding`

- Word2Vec
- GloVe
- FastText

5. N-gram Features

2 1-Label Encoding

Label Encoding is a technique used to convert categorical variables(texts) into numerical representations. Each unique category is assigned a unique integer value.

It can be quickly and easily integrated, but it does not understand the relationship between categories, for example, it does not recognize that nurses and doctors are closer to each other compared to others.

[ ]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Example categorical data
categories = ['teacher', 'nurse', 'police', 'doctor']

# Initializing the LabelEncoder
encoder = LabelEncoder()
```

```python
# Fitting and transforming the categories
encoded_labels = encoder.fit_transform(categories)

# Creating a DataFrame
df = pd.DataFrame({'Category': categories, 'Encoded_Labels': encoded_labels})

# Printing the DataFrame
df.head()
```

[ ]:

	Category	Encoded_Labels
0	teacher	3
1	nurse	1
2	police	2
3	doctor	0

3 2-One Hot Encoding

One Hot Encoding is a technique used to convert categorical variables into binary vectors. Each category is represented by a binary vector where only one element is "hot" (1) and the others are "cold" (0).

If the number of categories is low, it is feasible to use One Hot Encoding to convert texts into numerical values.If the number of categories is large, adding a significant number of columns can lead to unnecessary data expansion, resulting in increased computational cost and time.

[ ]:

```python
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Example categorical data
categories = ['teacher', 'nurse', 'police', 'doctor']

# Convert categorical data into a DataFrame
data = pd.DataFrame({'Category': categories})

# Initialize the OneHotEncoder
encoder = OneHotEncoder(sparse_output=False, dtype=int)

# Fit and transform the categorical data
encoded_data = encoder.fit_transform(data)

# Convert the encoded data to a DataFrame
encoded_df = pd.DataFrame(encoded_data, columns=categories)

# Print the encoded DataFrame
encoded_df.head()
```

[ ]:
```
   teacher  nurse  police  doctor
0        0      0       0       1
1        0      1       0       0
2        0      0       1       0
3        1      0       0       0
```

4 3-Count Vectorization

Count Vectorization is a technique used to convert text documents into numerical vectors based on the frequency of words in the documents. calculates according to the frequency of the word in the sentence

a) TF-IDF Vectorizer: It combines the concepts of "TF" (Term Frequency) and "IDF" (Inverse Document Frequency).

[ ]:
```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

# Example text data
documents = ["This is the first document.",
             "This document is the second document.",
             "And this is the third one.",
             "Is this the first document?"]

# Convert text data into a DataFrame
data = pd.DataFrame({'Text': documents})

# Initialize the TF-IDF Vectorizer
vectorizer = TfidfVectorizer()

# Fit and transform the text data
tfidf_vectors = vectorizer.fit_transform(data['Text'])

# Convert the TF-IDF vectors to a DataFrame
tfidf_df = pd.DataFrame(tfidf_vectors.toarray(), columns=vectorizer.
 ↪get_feature_names_out())

# Print the TF-IDF DataFrame
tfidf_df.head()
```

[ ]:
```
        and  document     first        is       one    second       the  \
0  0.000000  0.469791  0.580286  0.384085  0.000000  0.000000  0.384085
1  0.000000  0.687624  0.000000  0.281089  0.000000  0.538648  0.281089
2  0.511849  0.000000  0.000000  0.267104  0.511849  0.000000  0.267104
3  0.000000  0.469791  0.580286  0.384085  0.000000  0.000000  0.384085

      third      this
```

0	0.000000	0.384085
1	0.000000	0.281089
2	0.511849	0.267104
3	0.000000	0.384085

a)Bag Of Words (BOW):

It creates a vocabulary of unique words from the corpus and represents each document as a vector of word frequencies.

[ ]:
```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

# Example text data
documents = ["This is the first document.",
             "This document is the second document.",
             "And this is the third one.",
             "Is this the first document?"]

# Convert text data into a DataFrame
data = pd.DataFrame({'Text': documents})

# Initialize the CountVectorizer
vectorizer = CountVectorizer()

# Fit and transform the text data
bow_vectors = vectorizer.fit_transform(data['Text'])

# Convert the BOW vectors to a DataFrame
bow_df = pd.DataFrame(bow_vectors.toarray(), columns=vectorizer.
 ↪get_feature_names_out())

# Print the BOW DataFrame
bow_df.head()
```

[ ]:

	and	document	first	is	one	second	the	third	this
0	0	1	1	1	0	0	1	0	1
1	0	2	0	1	0	1	1	0	1
2	1	0	0	1	1	0	1	1	1
3	0	1	1	1	0	0	1	0	1

5 4) Word Embedding

Word Embedding is a technique in NLP that represents words as dense vectors in a high-dimensional space. It captures semantic meaning and word relationships, allowing for better understanding and processing of natural language. Word embeddings are learned from large text data using neural network models and provide dense representations that improve NLP model performance compared to sparse representations.

a) Word2Vec:

It is a neural network-based model that learns continuous vector representations (embeddings) of words from large text corpora. These embeddings capture semantic and syntactic relationships between words, allowing for more meaningful and context-aware word representations.

- CBOW (Continuous Bag of Words): predicts the target word based on the surrounding context words. Given the context words, CBOW tries to predict the target word in the center.

- Skip-gram: predicts the surrounding context words given a target word. Given a target word in the center, Skip-gram aims to predict the context words that typically appear around it.

- CBOW (Continuous Bag of Words):

[ ]:
```python
# CBOW (Continuous Bag of Words)

import pandas as pd
from gensim.models import Word2Vec

# Training data
sentences = [["I", "like", "apples"],
             ["I", "enjoy", "eating", "fruits"],
             ["Apples", "are", "delicious"],
             ["Fruits", "provide", "vitamins"]]

# Training the CBOW model with sg=0
model_cbow_sg0 = Word2Vec(sentences, min_count=1, window=2, sg=0)

# Accessing word vectors for CBOW (sg=0)
word_vectors_sg0 = model_cbow_sg0.wv

# Creating a DataFrame for word vectors with CBOW (sg=0)
word_vectors_df_sg0 = pd.DataFrame(word_vectors_sg0.vectors,
 ↪index=word_vectors_sg0.index_to_key)



# Displaying the word vectors DataFrame
word_vectors_df_sg0.head(10)
```

[ ]:
```
                  0         1         2         3         4         5   \
I         -0.000536  0.000236  0.005103  0.009009 -0.009303 -0.007117
vitamins  -0.008620  0.003666  0.005190  0.005742  0.007467 -0.006168
provide    0.000095  0.003077 -0.006813 -0.001375  0.007669  0.007346
Fruits    -0.008243  0.009299 -0.000198 -0.001967  0.004604 -0.004095
delicious -0.007139  0.001241 -0.007177 -0.002245  0.003719  0.005833
are       -0.008728  0.002130 -0.000874 -0.009320 -0.009429 -0.001411
Apples     0.008134 -0.004458 -0.001068  0.001007 -0.000191  0.001148
fruits     0.008168 -0.004443  0.008985  0.008254 -0.004435  0.000303
```

```
eating    -0.009579  0.008943  0.004165  0.009235  0.006644  0.002925
enjoy     -0.005156 -0.006668 -0.007777  0.008311 -0.001982 -0.006855

                  6         7         8         9  …        90        91  \
I          0.006459  0.008973 -0.005015 -0.003763  …  0.001631  0.000190
vitamins   0.001106  0.006047 -0.002840 -0.006174  …  0.001088 -0.001576
provide   -0.003673  0.002643 -0.008317  0.006205  … -0.004509  0.005702
Fruits     0.002743  0.006940  0.006065 -0.007511  … -0.007426 -0.001064
delicious  0.001198  0.002103 -0.004110  0.007225  …  0.003137 -0.004713
are        0.004433  0.003704 -0.006499 -0.006873  …  0.009072  0.008939
Apples     0.006115 -0.000020 -0.003246 -0.001511  … -0.002702  0.000444
fruits     0.004274 -0.003926 -0.005560 -0.006512  …  0.002058 -0.004004
eating     0.009804 -0.004425 -0.006803  0.004227  … -0.005085  0.001131
enjoy     -0.004154  0.005144 -0.002869 -0.003750  … -0.008977  0.008592

                 92        93        94        95        96        97  \
I          0.003474  0.000218  0.009619  0.005061 -0.008917 -0.007042
vitamins   0.002197 -0.007882 -0.002717  0.002663  0.005347 -0.002392
provide    0.009180 -0.004100  0.007965  0.005375  0.005879  0.000513
Fruits    -0.000795 -0.002563  0.009683 -0.000459  0.005874 -0.007448
delicious  0.005281 -0.004233  0.002642 -0.008046  0.006210  0.004819
are       -0.008209 -0.003012  0.009887  0.005105 -0.001588 -0.008692
Apples    -0.003538 -0.000419 -0.000709  0.000823  0.008196 -0.005737
fruits    -0.008241  0.006278 -0.001949 -0.000666 -0.001771 -0.004536
eating     0.002883 -0.001536  0.009932  0.008350  0.002416  0.007118
enjoy      0.004047  0.007470  0.009746 -0.007290 -0.009040  0.005836

                 98        99
I          0.000901  0.006393
vitamins  -0.009510  0.004506
provide    0.008213 -0.007019
Fruits    -0.002506 -0.005550
delicious  0.000787  0.003013
are        0.002962 -0.006676
Apples    -0.001660  0.005573
fruits     0.004062 -0.004270
eating     0.005891 -0.005581
enjoy      0.009391  0.003507

[10 rows x 100 columns]
```

Skip-gram:

[ ]:
```python
# Skip-gram

import pandas as pd
from gensim.models import Word2Vec
```

```python
# Training data
sentences = [["I", "like", "apples"],
             ["I", "enjoy", "eating", "fruits"],
             ["Apples", "are", "delicious"],
             ["Fruits", "provide", "vitamins"]]

# Training the Skip-gram model with sg=1
model_skip_gram_sg1 = Word2Vec(sentences, min_count=1, window=2, sg=1)

# Accessing word vectors for Skip-gram (sg=1)
word_vectors_sg1 = model_skip_gram_sg1.wv

# Creating a DataFrame for word vectors with Skip-gram (sg=1)
word_vectors_df_sg1 = pd.DataFrame(word_vectors_sg1.vectors,␣
 ↪index=word_vectors_sg1.index_to_key)

# Displaying the word vectors DataFrame
word_vectors_df_sg1.head(10)
```

[ ]:

```
                 0         1         2         3         4         5  \
I        -0.000536  0.000236  0.005103  0.009009 -0.009303 -0.007117
vitamins -0.008620  0.003666  0.005190  0.005742  0.007467 -0.006168
provide   0.000095  0.003077 -0.006813 -0.001375  0.007669  0.007346
Fruits   -0.008243  0.009299 -0.000198 -0.001967  0.004604 -0.004095
delicious -0.007139  0.001241 -0.007177 -0.002245  0.003719  0.005833
are      -0.008729  0.002131 -0.000874 -0.009321 -0.009430 -0.001411
Apples    0.008133 -0.004458 -0.001068  0.001006 -0.000191  0.001148
fruits    0.008168 -0.004443  0.008985  0.008254 -0.004435  0.000303
eating   -0.009579  0.008943  0.004165  0.009235  0.006644  0.002925
enjoy    -0.005156 -0.006668 -0.007777  0.008311 -0.001982 -0.006855

                 6         7         8         9  …        90        91  \
I         0.006459  0.008973 -0.005015 -0.003763  …  0.001631  0.000190
vitamins  0.001106  0.006047 -0.002840 -0.006174  …  0.001088 -0.001576
provide  -0.003673  0.002643 -0.008317  0.006205  … -0.004509  0.005702
Fruits    0.002743  0.006940  0.006065 -0.007511  … -0.007426 -0.001064
delicious  0.001198  0.002103 -0.004110  0.007225  …  0.003137 -0.004713
are       0.004433  0.003705 -0.006500 -0.006874  …  0.009073  0.008940
Apples    0.006114 -0.000020 -0.003246 -0.001511  … -0.002702  0.000444
fruits    0.004274 -0.003926 -0.005560 -0.006512  …  0.002058 -0.004004
eating    0.009804 -0.004425 -0.006803  0.004227  … -0.005085  0.001131
enjoy    -0.004154  0.005144 -0.002869 -0.003750  … -0.008977  0.008592

                92        93        94        95        96        97  \
I         0.003474  0.000218  0.009619  0.005061 -0.008917 -0.007042
vitamins  0.002197 -0.007882 -0.002717  0.002663  0.005347 -0.002392
```

```
provide    0.009180 -0.004100  0.007965  0.005375  0.005879  0.000513
Fruits    -0.000795 -0.002563  0.009683 -0.000459  0.005874 -0.007448
delicious  0.005281 -0.004233  0.002642 -0.008046  0.006210  0.004819
are       -0.008210 -0.003013  0.009888  0.005105 -0.001588 -0.008693
Apples    -0.003538 -0.000419 -0.000709  0.000823  0.008195 -0.005737
fruits    -0.008241  0.006278 -0.001949 -0.000666 -0.001771 -0.004536
eating     0.002883 -0.001536  0.009932  0.008350  0.002416  0.007118
enjoy      0.004047  0.007470  0.009746 -0.007290 -0.009040  0.005836

                 98        99
I          0.000901  0.006393
vitamins  -0.009510  0.004506
provide    0.008213 -0.007019
Fruits    -0.002506 -0.005550
delicious  0.000787  0.003013
are        0.002962 -0.006677
Apples    -0.001660  0.005572
fruits     0.004062 -0.004270
eating     0.005891 -0.005581
enjoy      0.009391  0.003507

[10 rows x 100 columns]
```

b) GloVe:

GloVe stands for Global Vectors for Word Representation. It is an unsupervised learning algorithm that aims to generate word embeddings by capturing global word co-occurrence patterns in a corpus.

[ ]:
```python
import numpy as np

# Kelimeler ve vektörler
words = ['apple', 'orange', 'banana', 'grape']
vectors = [
    [0.1, 0.2, 0.3, 0.4],
    [0.5, 0.6, 0.7, 0.8],
    [0.9, 1.0, 1.1, 1.2],
    [1.3, 1.4, 1.5, 1.6]
]

# GloVe dosyasına yazma
glove_file = 'glove_file.txt'  # Oluşturulacak GloVe dosyasının adı ve yolu

with open(glove_file, 'w', encoding='utf-8') as f:
    for word, vector in zip(words, vectors):
        vector_str = ' '.join(str(num) for num in vector)
        f.write(f"{word} {vector_str}\n")
```

[ ]:
```python
import pandas as pd
from gensim.models import KeyedVectors

# Load pre-trained GloVe embeddings
glove_file = 'glove_file.txt'  # Path to the GloVe file
# Reading the GloVe file
word_vectors_df = pd.read_csv(glove_file, sep=' ', header=None, index_col=0,␣
↪quoting=3)


# Displaying the word vectors DataFrame
word_vectors_df.head()
```

[ ]:
```
          1    2    3    4
0
apple   0.1  0.2  0.3  0.4
orange  0.5  0.6  0.7  0.8
banana  0.9  1.0  1.1  1.2
grape   1.3  1.4  1.5  1.6
```

c) FastText

It learns word embeddings using the Skip-gram or Continuous Bag-of-Words (CBOW) architecture, making it effective for various natural language processing tasks. FastText is particularly useful for languages with rich morphology and large-scale datasets

[ ]:
```python
import pandas as pd
from gensim.models import FastText

# Training data
sentences = [["I", "like", "apples"],
             ["I", "enjoy", "eating", "fruits"]]

# Training the FastText model
model_fasttext = FastText(sentences, min_count=1, window=5, vector_size=100)

# Accessing word vectors
word_vectors = model_fasttext.wv

# Creating a DataFrame for word vectors
word_vectors_df = pd.DataFrame(word_vectors.vectors, index=word_vectors.
↪index_to_key)

# Displaying the word vectors DataFrame
word_vectors_df.head(10)

similarity = model_fasttext.wv.similarity("apples", "fruits")
print("Similarity between 'apples' and 'fruits':", similarity)
```

```
analogies = model_fasttext.wv.most_similar(positive=["eating", "fruits"],␣
 ↪negative=["apples"])
print("Word analogy for 'eating' and 'fruits' - 'apples':", analogies)
```

```
Similarity between 'apples' and 'fruits': 0.5611198
Word analogy for 'eating' and 'fruits' - 'apples': [('enjoy',
0.09406167268753052), ('I', -0.022638631984591484), ('like',
-0.06936056911945343)]
```

[ ]:
```
import fasttext.util
fasttext.util.download_model('en', if_exists='ignore')  # English
ft = fasttext.load_model('cc.en.300.bin')
```

6 5) N-gram features

N-gram features are contiguous sequences of n words in a text document. They capture the contextual information and relationships between words, considering not just individual words but also the groups of words they form.

[ ]:
```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

# Example text data
documents = ["This is the first document.",
             "This document is the second document.",
             "And this is the third one.",
             "Is this the first document?"]

# Convert text data into a DataFrame
data = pd.DataFrame({'Text': documents})

# Initialize the CountVectorizer with desired n-gram range
ngram_vectorizer = CountVectorizer(ngram_range=(2,3))

# Fit and transform the text data
ngram_vectors = ngram_vectorizer.fit_transform(data['Text'])

# Convert the N-gram vectors to a DataFrame
ngram_df = pd.DataFrame(ngram_vectors.toarray(), columns=ngram_vectorizer.
 ↪get_feature_names_out())

# Print the N-gram DataFrame
ngram_df.head()
```

[ ]:

```
   and this  and this is  document is  document is the  first document  \
0         0            0            0                0               1
1         0            0            1                1               0
2         1            1            0                0               0
3         0            0            0                0               1

   is the  is the first  is the second  is the third  is this  …  \
0       1             1              0             0        0  …
1       1             0              1             0        0  …
2       1             0              0             1        0  …
3       0             0              0             0        1  …

   the second document  the third  the third one  third one  this document  \
0                    0          0              0          0              0
1                    1          0              0          0              1
2                    0          1              1          1              0
3                    0          0              0          0              0

   this document is  this is  this is the  this the  this the first
0                 0        1            1         0               0
1                 1        0            0         0               0
2                 0        1            1         0               0
3                 0        0            0         1               1

[4 rows x 25 columns]
```