|  |  |
| --- | --- |
| Sl. No. | Topic Name |
| 1 | N-Grams NLP |
| 2 | Sklearn tfidfVectorizer() |
| 3 | Data Import Export and Compression |
| 4 | Pandas |
| 5 | Data Filtering |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |

N-Grams NLP

Notes 1

N-Grams are phrases cut out of a sentence with N consecutive words. Thus a Unigram takes a sentence and gives us all the words in that we fence. A Bigram takes a sentence and gives us sets of two consecutive words in the sentence. A Trigram gives sets of three consecutive words in a sentence.

Let me explain with an example.

Unigram - [Let] [me] [explain] [with] [an] [example.]

Bigram [let me] [me explain] [explain with] [with an] [an example]

Trigram [let me explain] [me explain with] [explain with an] [with an example]

Notes 2

N-grams of texts are extensively used in text mining and natural language processing tasks. An n-gram is a contiguous sequence of n items from a given sample of text or speech. an n-gram of size 1 is referred to as a "unigram"; size 2 is a "bigram"; size 3 is a "trigram". When N>3 this is usually referred to as four grams or five grams and so on.

Formula to calculate number of N-grams in a sentence.

If X=Number of words in a given sentence K, the number of n-grams for sentence K would be:

Ngramk = X - (N - 1)

Example:

Sentence: I want to learn Machine Learning

Unigram: now calculate number of unigrams in sentence using formula

here, X = 6 and N = 1 (for unigram)

Ngramk = X - (N - 1)

Ngramk = 6 - (1–1) = 6 (i.e. unigram is equal to number of words in a sentence)

[I][want][to][learn][Machine][Learning]

Biagram:

here, X = 6 and N = 2 (for biagram)

Ngramk = X - (N - 1)

Ngramk = 6 - (2–1) = 5

[I want][want to][to learn][learn Machine][Machine Learning]

Trigram:

here, X = 6 and N = 3 (for trigram)

Ngramk = X - (N - 1)

Ngramk = 6 - (3–1) = 4

[I want to][want to learn][to learn Machine][learn Machine Learning]

You can also generate for N=4,5,6 and so on.

Sklearn tfidfVectorizer()

ngram\_rangetuple:

ngram\_rangetuple (min\_n, max\_n), default=(1, 1)

The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min\_n <= n <= max\_n will be used. For example an ngram\_range of (1, 1) means only unigrams, (1, 2) means unigrams and bigrams, and (2, 2) means only bigrams. Only applies if analyzer is not callable.

max\_dffloat:

max\_dffloat or int, default=1.0

When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float in range [0.0, 1.0], the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

min\_dffloat:

min\_dffloat or int, default=1

When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float in range of [0.0, 1.0], the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

Using stop words:

Stop words are words like “and”, “the”, “him”, which are presumed to be uninformative in representing the content of a text, and which may be removed to avoid them being construed as signal for prediction. Sometimes, however, similar words are useful for prediction, such as in classifying writing style or personality.

There are several known issues in our provided ‘english’ stop word list. It does not aim to be a general, ‘one-size-fits-all’ solution as some tasks may require a more custom solution. See [NQY18] for more details.

Please take care in choosing a stop word list. Popular stop word lists may include words that are highly informative to some tasks, such as computer.

You should also make sure that the stop word list has had the same preprocessing and tokenization applied as the one used in the vectorizer. The word we’ve is split into we and ve by CountVectorizer’s default tokenizer, so if we’ve is in stop\_words, but ve is not, ve will be retained from we’ve in transformed text. Our vectorizers will try to identify and warn about some kinds of inconsistencies.

Data Import Export and Compression

What is the most efficient way of exporting DataFrame for fast read write process?

When it comes to efficient read and write processes for large datasets in pandas, the choice of file format and optimization techniques becomes crucial. The following are some popular formats and methods for achieving fast read and write operations:

Parquet Format:

Parquet is a columnar storage format that is highly efficient for both reading and writing.

It supports compression and is designed to be a high-performance format for analytics.

Pandas provides a to\_parquet method for writing DataFrames to Parquet format.

import pandas as pd

# Write DataFrame to Parquet

df.to\_parquet('data.parquet', index=False)

# Read DataFrame from Parquet

df = pd.read\_parquet('data.parquet')

Feather Format:

Feather is a binary columnar data format designed for use with analytics.

It is lightweight and provides fast read and write operations.

Both pandas and Arrow (a separate library) provide support for Feather.

import pandas as pd

# Write DataFrame to Feather

df.to\_feather('data.feather')

# Read DataFrame from Feather

df = pd.read\_feather('data.feather')

HDF5 Format:

HDF5 is a versatile format that supports compression and efficient chunked read and write operations.

Pandas provides support for HDF5 through the HDFStore interface.

import pandas as pd

# Write DataFrame to HDF5

df.to\_hdf('data.h5', key='mydata', mode='w', complevel=9, complib='blosc')

# Read DataFrame from HDF5

df = pd.read\_hdf('data.h5', key='mydata')

Adjust the complevel and complib parameters for compression based on your requirements.

CSV with Compression:

If you prefer a human-readable format like CSV, you can still achieve some level of compression using a compressed format such as gzip.

import pandas as pd

# Write DataFrame to CSV with gzip compression

df.to\_csv('data.csv.gz', index=False, compression='gzip')

# Read DataFrame from CSV with gzip compression

df = pd.read\_csv('data.csv.gz', compression='gzip')

Choose the format that best suits your use case and requirements. Consider the trade-offs between file size, readability, and speed based on your specific needs. Additionally, adjust compression options and chunk sizes based on the characteristics of your data and the resources available on your system.

Perquet vs Feather

Parquet and Feather are both columnar storage formats designed for efficient analytics and fast read/write operations. They share several similarities but also have some differences. Here's a comparison between Parquet and Feather:

Parquet:

Flexibility:

Parquet is a standard open-source columnar storage format that is part of the Apache Arrow project.

It is widely used in the Hadoop ecosystem and supported by many big data processing frameworks.

Compression:

Parquet supports various compression algorithms, including Snappy, Gzip, and LZO.

It is highly efficient in terms of both storage and processing due to its columnar nature, which allows for better compression and predicate pushdown.

Schema Evolution:

Parquet supports schema evolution, allowing you to add, remove, or modify columns without breaking compatibility with existing data.

Performance:

Parquet is optimized for performance in analytical workloads, particularly when dealing with large datasets and complex queries.

Ecosystem Integration:

Parquet is integrated with many big data tools and frameworks, including Apache Spark, Apache Hive, and Apache Impala.

Feather:

Simplicity:

Feather is a simpler and lightweight binary columnar data format designed for easy interoperability between languages.

It is part of the Apache Arrow project, which focuses on in-memory analytics.

Compression:

Feather supports compression, but the options are more limited compared to Parquet. It typically uses LZ4 compression by default.

Schema Evolution:

Feather supports schema evolution, making it possible to read older Feather files with newer schema versions.

Performance:

Feather is designed for high-performance in-memory analytics, and it is optimized for reading and writing data quickly in a cross-language context.

Ecosystem Integration:

Feather is not as deeply integrated into big data ecosystems as Parquet. It is often used for sharing data between different programming languages (e.g., Python and R) and for in-memory analytics.

Considerations:

Use Case: If you are working within a big data ecosystem, Parquet might be a better fit. If you need a lightweight format for quick inter-language data exchange, Feather could be more suitable.

Ecosystem Compatibility: Consider the tools and frameworks you are using. Parquet may be a better choice if you are working with Apache Spark, for example.

Compression Requirements: If you have specific compression requirements or preferences, check the compression options supported by each format.

In summary, both Parquet and Feather have their strengths, and the choice between them depends on your specific use case, ecosystem, and performance requirements.

Parquet Index

The index parameter in the to\_parquet method of a pandas DataFrame controls whether the DataFrame's index should be included in the Parquet file or not.

Here's an explanation of the two options:

If index=True (default):

The DataFrame's index will be included in the Parquet file as a separate column.

When reading the Parquet file back into a DataFrame, the index will be restored.

This is useful if you want to preserve and use the index information later.

import pandas as pd

# Create a sample DataFrame

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['X', 'Y', 'Z'])

# Write DataFrame to Parquet with index

df.to\_parquet('data\_with\_index.parquet', index=True)

If index=False:

The DataFrame's index will not be included in the Parquet file.

When reading the Parquet file back into a DataFrame, a default integer index will be created.

This can be useful if you don't need to preserve the original index or if you want to save space in the file.

import pandas as pd

# Create a sample DataFrame

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['X', 'Y', 'Z'])

# Write DataFrame to Parquet without index

df.to\_parquet('data\_without\_index.parquet', index=False)

In summary, the choice of whether to include the index in the Parquet file depends on your specific use case. If the index is important for your analysis or if you want to preserve the original structure of the DataFrame, you may choose index=True. If the index is not crucial or if you want to save space, you may choose index=False.

Which is the best for faster compression and decompression

Among the commonly used compression algorithms for Parquet files, snappy is often favored for its faster decompression speed compared to other algorithms like gzip. Snappy is designed for fast compression and decompression, making it a good choice when read performance is a priority.

Here's an example of using snappy compression with the to\_parquet method:

import pandas as pd

# Create a sample DataFrame

df = pd.DataFrame({'A': range(1, 11), 'B': range(11, 21)})

# Write DataFrame to Parquet with snappy compression

df.to\_parquet('data\_snappy\_compressed.parquet', compression='snappy')

When using snappy compression, you can expect faster decompression speeds compared to other algorithms like gzip. However, keep in mind that the choice of compression algorithm might also depend on other factors, such as the compression ratio and the amount of disk space savings.

If storage space is a concern, you might need to find a balance between compression ratio and decompression speed. For example, gzip generally provides better compression ratios but may be slower to decompress compared to snappy. It's recommended to test different compression options on your specific data and workload to determine the best trade-off for your use case.

Additionally, if read performance is crucial, you might also want to consider using efficient file formats like Parquet itself, which is designed for high-performance columnar storage and query processing. The choice of compression algorithm, in combination with the file format, can have a significant impact on the overall read and decompression speed.

Pandas

Pandas Joint

I have two data frame df1 and df2. I want to keep only those records in df1 which matches in df2 and add df2 columns in df1

If you want to keep only the records from df1 that have matching values in df2, you can perform a merge with an inner join and keep only the rows from the left DataFrame (df1). This is effectively filtering df1 to only include rows with matching values in df2.

import pandas as pd

# Create two sample DataFrames

df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})

df2 = pd.DataFrame({'ID': [2, 3, 4], 'Age': [25, 30, 22]})

# Perform an inner join and keep only rows from df1

result\_df1\_filtered = pd.merge(df1, df2, on='ID', how='inner')

# Display the result

print("DataFrame 1:")

print(df1)

print("\nDataFrame 2:")

print(df2)

print("\nFiltered DataFrame 1 (keeping only matching rows):")

print(result\_df1\_filtered)

Data Filtering

How to check characters other than English in pandas?

In pandas, you can check for non-English characters in a DataFrame column using various methods. One common approach is to use regular expressions to filter or identify rows that contain non-English characters.

```

import pandas as pd

import re

# Sample DataFrame

data = {'text\_column': ['Hello', 'Bonjour', '你好', 'مرحبا', 'こんにちは']}

df = pd.DataFrame(data)

# Function to check if a string contains non-English characters

def has\_non\_english\_chars(text):

# Using regular expression to match non-English characters

return bool(re.search('[^\x00-\x7F]', text))

# Applying the function to the DataFrame

df['has\_non\_english\_chars'] = df['text\_column'].apply(has\_non\_english\_chars)

# Displaying the DataFrame

print(df)

```

In this example, the has\_non\_english\_chars function checks if a given string contains any characters outside the ASCII range (0 to 127), which generally corresponds to English characters. The regular expression [^\x00-\x7F] matches any character that is not within this range.

You can customize the regular expression based on your specific requirements or the character set you want to consider as "non-English."

Keep in mind that this approach may not be foolproof, as it depends on the definition of "English" characters and the specific character set you are considering as non-English. Different languages and character sets may have unique characters that fall outside the ASCII range. Adjust the regular expression accordingly based on your use case.

Reference

How to Use Tfidftransformer & Tfidfvectorizer?

<https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/>

TF-IDF Vectorizer scikit-learn

<https://medium.com/@cmukesh8688/tf-idf-vectorizer-scikit-learn-dbc0244a911a>

AttributeError: 'TfidfVectorizer' object has no attribute 'get\_feature\_names\_out'

<https://stackoverflow.com/questions/70215049/attributeerror-tfidfvectorizer-object-has-no-attribute-get-feature-names-out>

Understanding Cosine Similarity in Python with Scikit-Learn

<https://memgraph.com/blog/cosine-similarity-python-scikit-learn>

Scikit-learn Count Vectorizers

<https://medium.com/@cmukesh8688/scikit-learn-count-vectorizers-32b58dee0541>