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

What is n-gram in 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.

Important parameters of Sklearn tfidfVectorizer() class





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.



**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.



**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.



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.



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.

**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.

Which is the best for faster compression and decompression

Among the commonly used compression algorithms for Parquet files, snappy is often favoured 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:



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 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.



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.



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>