Advanced Data Mining and Clustering Techniques for Large-Scale Text Embedding Analysis Using Python and Dask

Maciej Złotorowicz, Ryszard Błażej, Jakub Zaręba
Akademia Górniczo-Hutnicza

23.12.2024

Abstract

This project explores advanced data mining and clustering techniques applied to large-scale text embedding analysis. Using Python and Dask, the project aims to efficiently process and analyze vast amounts of code data scraped from popular Python repositories, leveraging distributed computing to handle large datasets. The primary focus is on embedding generation, clustering, visualization of high-dimensional text data, and training specialized transformer that will summarize code snippets.

Contents

1 Motivation **4**

2 Technology **5**
- 2.1 Dask . . . 5
- 2.2 Requests, BeautifulSoup4 and ast . . . 5
- 2.3 Postgres . . . 5
- 2.4 h5 (HDF5) . . . 5
- 2.5 Safetensors . . . 6
- 2.6 Sklearn, Torch and Transformers . . . 6

3 Data scrapping and Dataset Description **7**
- 3.1 Scrapping techniques . . . 7
- 3.2 Resulting Dataset . . . 7
- 3.3 Tags analysis . . . 9
- 3.4 Docstring analysis . . . 12

4 Features extraction and clustering **13**
- 4.1 Other tested techniques of feature extraction . . . 14
  - 4.1.1 TF-IDF Encoding . . . 14
- 4.2 Embeddings . . . 14
- 4.3 Clustering . . . 15
- 4.4 Visualization . . . 16
- 4.5 Dim reduction . . . 17
  - 4.5.1 PCA . . . 17
  - 4.5.2 TSNE . . . 19
  - 4.5.3 LocallyLinearEmbedding . . . 19
  - 4.5.4 Variational Autoencoder . . . 20
- 4.6 Results . . . 23

5 Training the Transformer **24**
- 5.1 Dataset . . . 24
- 5.2 Training . . . 27
- 5.3 Results . . . 27

6 Possible farther research **29**

Literature **30**

A Samples from clusters **31**

The project demonstrates the integration of various Python libraries and tools to perform scalable data analysis, providing insights into the structure and patterns within large text corpora. The methodologies and results presented in this project can be applied to various domains, including natural language processing, information retrieval, and big data analytics.

1 Motivation

Text data offers a wealth of information that can be unlocked using modern computational tools. However, the challenge lies in processing such data efficiently, particularly when dealing with large-scale, unstructured datasets. Among the various types of text data available, code bases stand out as a rich source of information. Freely available repositories like GitHub contain massive amounts of code. The project seeks to address the following challenges and opportunities:

1. Scalable Processing - Code repositories often consist of millions of lines of text. Efficiently processing such large-scale datasets requires the use of distributed frameworks, such as Dask, which allow for parallelized workflows and scalable analysis.
2. Semantic Understanding - Extracting meaningful patterns from code requires sophisticated techniques, such as embedding generation, which can capture both the syntax and semantics of the code snippets.
3. Uncovering Patterns - Clustering code snippets can reveal underlying structures, common patterns, or even latent relationships between different programming constructs. These insights can have practical applications, such as improving search engines or aiding in software documentation.
4. Visualization and Interpretability - Understanding high-dimensional representations of code snippets, such as embeddings, is made more accessible through visualization techniques. These visualizations can help identify trends, groupings, and other insights within the data.
5. Practical Impact - The generated dataset contains a vast amount of code snippets and their associated descriptions, which can be further utilized for training large language models (LLMs). These models could be tailored for specific tasks, such as code generation, summarization, or completion, making this project highly relevant in the growing field of AI-driven software development tools.

2 Technology

With this project, we faced significant challenges due to the volume of text data involved. Processing and analyzing that amount of code snippets and descriptions required tools. The implementation required the integration of several Python libraries, each selected for its ability to address specific aspects and issues with processing data, analysis, and storage. Overall, this project can be easily scaled up for running on clusters on even bigger datasets (We only used 300 repositories out of 10,000 so we could scrape more examples).

2.1 Dask

The volume of data required a solution that could scale computations beyond the limitations of pandas.[1] Dask allows for distributed computing and batch processing and has single-handedly made this project possible. Bigger arrays or dataframes can lead to python kernel crash coming from python not being able to allocate memory.

	Array	Chunk
Bytes	42.09 MiB	78.12 kiB
Shape	(55166, 200)	(100, 200)
Dask graph	552 chunks in 66 graph layers	
Data type	float32 numpy.ndarray	

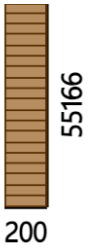


Figure 1: Example of dask array spited into small chunks.

2.2 Requests, BeautifulSoup4 and ast

BeautifulSoup4 and Requests are a very popular choice for scrapping data from the web. `ast` was used to parse and process python code because python parser probably knows best what is going on.

2.3 Postgres

PostgreSQL was chosen for flexibility. Data could be easily stored and exchanged between multiple users. It is also well integrated with Dask allowing for querying data in batches automatically.[1]

2.4 h5 (HDF5)

HDF5 data format is well suited for storing large arrays and structured data and outperforms Postgres. Its speed, compression capabilities, and ability to handle large datasets

made it a perfect choice for caching hard to compute embeddings during processing. It sometimes was used for storing text data and serializing dataframes.

2.5 Safetensors

Safetensors is a new lightweight, safe and efficient library for storing tensor data (models).

2.6 Sklearn, Torch and Transformers

Scikit-learn provides set of tools required for this project[9]. Multiple algorithms and tools such as k-means clustering and PCA were used for analyzing and visualizing high-dimensional embeddings. Torch - It is time to go beyond tensorflow Transformers - Transformers is a new free library that allows us to use state-of-the-art technology in NLP, namely LLM's[10]. It was crucial for generating high-quality text embeddings and performing other seq2seq tasks such as summarization or transformation of text data. It was also used for finetuneing of the model.

3 Data scrapping and Dataset Description

Efficient and accurate data collection was a critical component of this project, given the focus on large-scale analysis of Python code snippets. This section details the scraping techniques employed to gather the data, the resulting dataset structure, its size and characteristics, and any preprocessing steps undertaken.

3.1 Scrapping techniques

The dataset was compiled by scraping publicly available Python code from popular open-source repositories hosted on GitHub. The scraping process was distributed across multiple computers to maximize efficiency and parallelize workload. Each system handled a subset of repositories to ensure redundancy and balanced resource utilization.

3.2 Resulting Dataset

The data set collected consisted of the most frequently used Python libraries according to the PyPi index. The ones with the largest size are represented on the plot 2

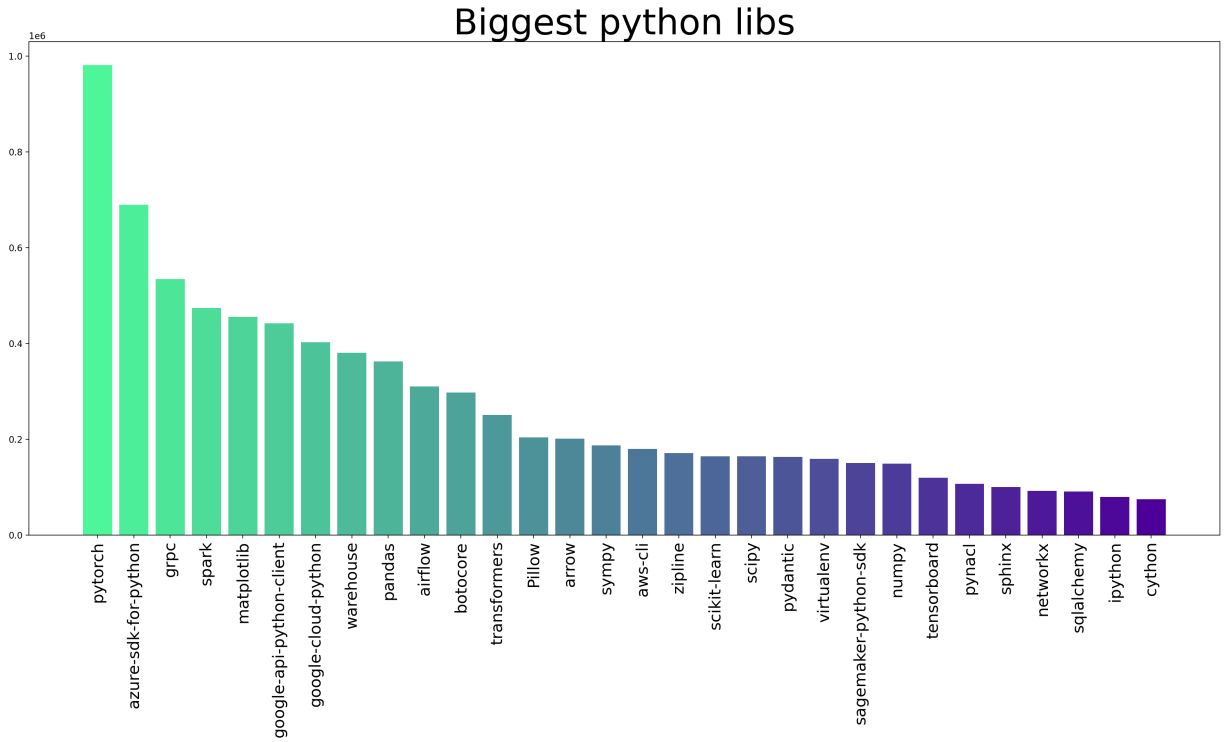


Figure 2: Biggest python libraries by size on disk

However, plain disk size is not the best representation for our dataset, because we focus only on Python files with function declarations, and a lot of libraries include many

different things, for example, our first by size Python includes a ton of C++ code meant for its optimization.

To further enable analysis and increase usability of our dataset, we selected all Python files and later separated them into single functions. After that we transferred those blocks of code into representation that allows accessing all the parts necessary:

- **file_id** - representation of the file from which the function is taken.
- **repo** - the number of repository to which the code belongs.
- **name** - name of the function.
- **args** - arguments taken by the function (if there are any)
- **args_types** - types for the arguments (if given)
- **args_defaults** - default values for arguments (if given)
- **body** - body of the function
- **dostring** - documentation of the function in a form of a string (if present)

As for the resulting dataset we can again measure which libraries make up the biggest part of our collection, by measuring how many functions have we taken from each of it. The results are presented in plot 3.

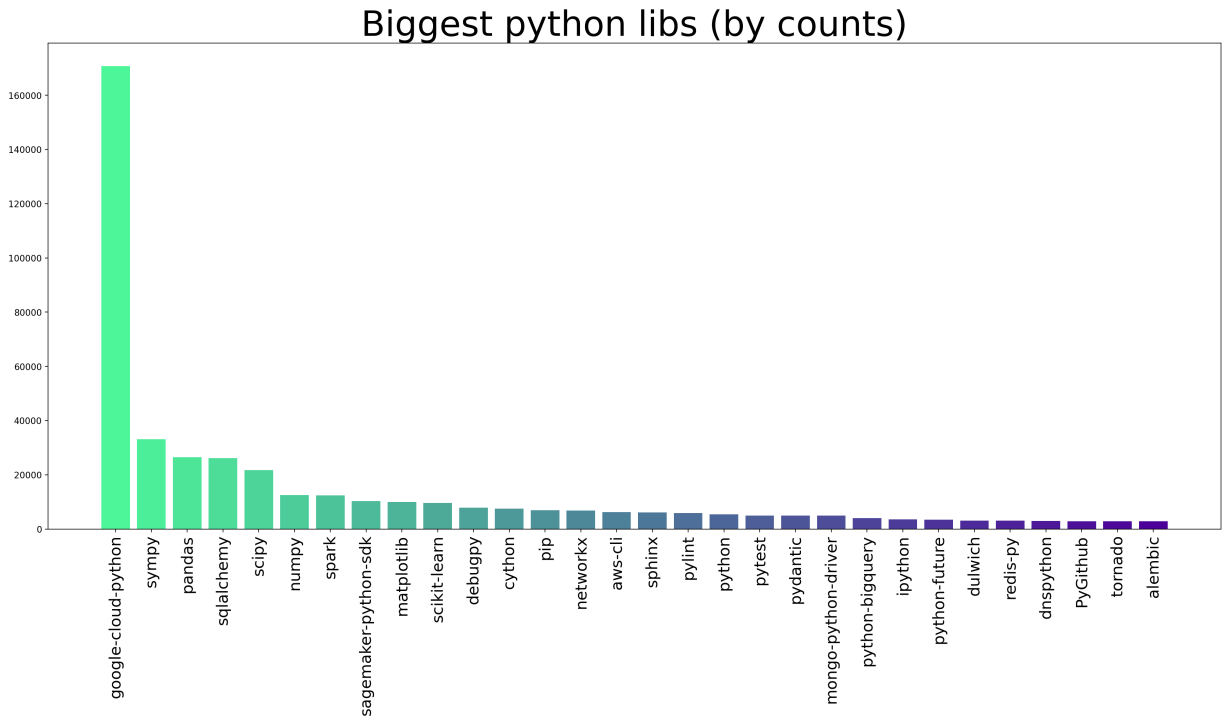


Figure 3: Biggest python libraries by function counts

3.3 Tags analysis

On Github, each repository can be labeled with some tags that generally represent what it does. To better know our gathered data, we visualized data about tags that have been assigned to repositories gathered by us, and we created a custom packed bubble chart with code as follows:

```
import random
import circlify

# compute circle positions
circles = circlify.circlify(
    df_xdict['value'].tolist(),
    show_enclosure=False,
    target_enclosure=circlify.Circle(x=0, y=0, r=1)
)

max_r = max(x_dict.values())

# reverse the order of the circles to match the order of data
circles = circles[::-1]

# Create just a figure and only one subplot
fig, ax = plt.subplots(figsize=(10, 10))

# Remove axes
ax.axis('off')

# Find axis boundaries
lim = max(
    max(
        abs(circle.x) + circle.r,
        abs(circle.y) + circle.r,
    )
    for circle in circles
)
plt.xlim(-lim, lim)
plt.ylim(-lim, lim)

# print circles
for circle, label in zip(circles, labels):
    x, y, r = circle
    ax.add_patch(plt.Circle((x, y), r, alpha=0.6, linewidth=2, color = ((random.uniform(0.2, 0.8)),
(random.uniform(0.2, 0.8)), (random.uniform(0.2, 0.8)))))
    plt.annotate(
        label,
        (x, y),
        va='center',
        ha='center',
        fontsize= 90 * r
    )
```

Code Snippet 1: Code for creating packed bubble chart

The resulting plot is shown in figure 4:

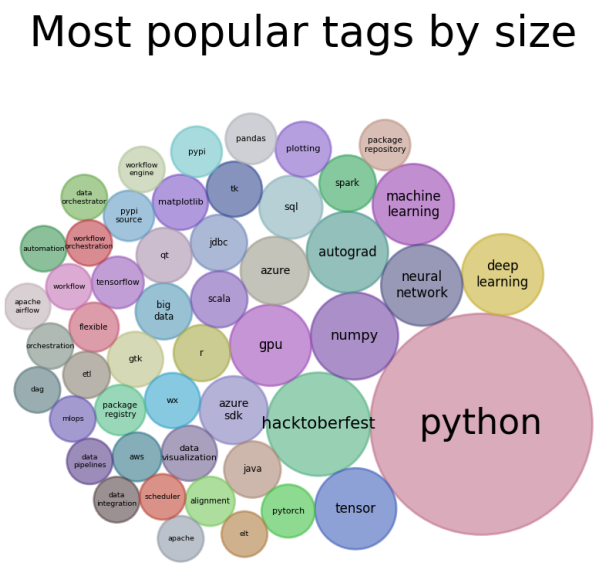


Figure 4: Tags most prevalent in our dataset

To no surprise, python is the most common tag that occurs; however, other ones can give us some insight about what topics are covered by libraries gathered in our dataset.

We also found an unexpected anomaly. The histogram in Figure 6 revealed a surprising outlier in the dataset. One specific function stands out, measuring nearly 500k characters in length.

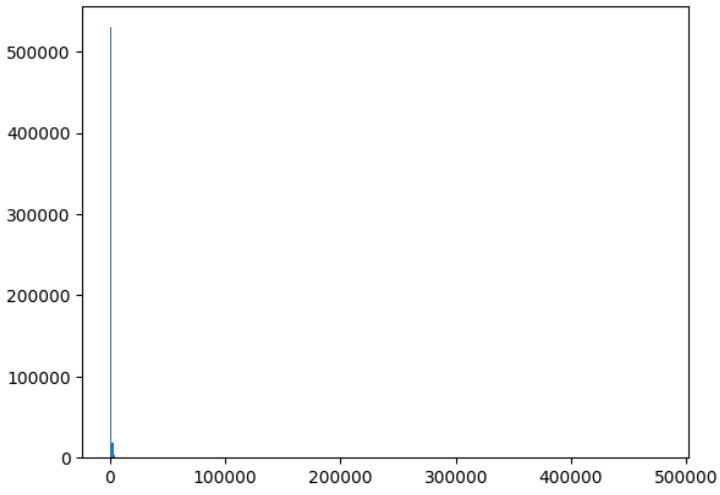


Figure 5: **That** one function is quite lengthy

That function is named `eqs_165x165` and it is coming from sympy library from file `sympy/polys/benchmarks/bench_solvers.py`, it is part of a benchmark test.

```
return [uk_0 + 50719 * uk_1 + 2789545 * uk_10 + 411400 * uk_100 + 1683000 * uk_101 + 166375 * uk_103 +
    680625 * uk_104 + 2784375 * uk_106 + 729 * uk_109 + ...
```

After removing **that** outlier histogram is much cleaner.

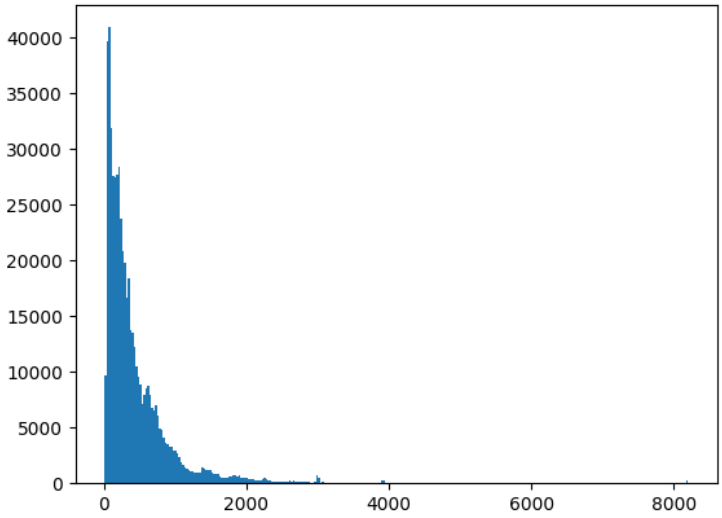


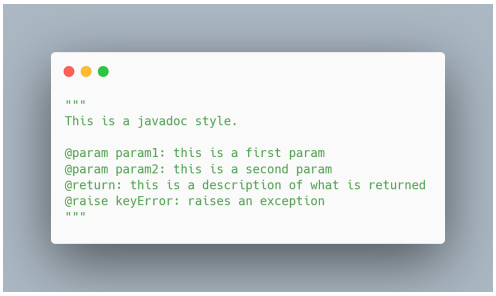
Figure 6: Histogram of function lengths

3.4 Docstring analysis

As docstrings are our point of interest for further use, we looked into them to get more insight. Generally in Python, there are four commonly used conventions of writing docstrings;

- **Epytext** - similar to javadoc, used by the automatic documentation generation code Epydoc [3].
- **reST** - reStructuredText, used by another automatic documentation plugin, namely Sphinx [8].
- **Google** - style used by the company in their own libraries [5].
- **Numpydoc** - a more elaborate version of the Google style, recommended by numpy [7].

The examples of each are shown in figure 7.



```
"""
This is a javadoc style.

@param param1: this is a first param
@param param2: this is a second param
@return: this is a description of what is returned
@raise keyError: raises an exception
"""
```

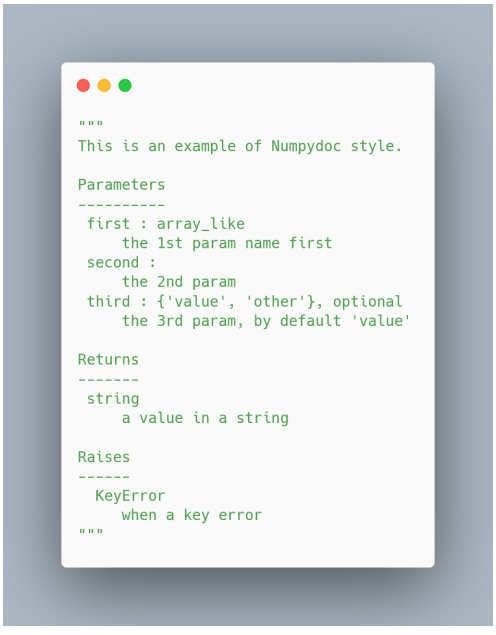
(a) Epytext

```
"""
This is a reST style.

:param param1: this is a first param
:param param2: this is a second param
:returns: this is a description of what is returned
:raises keyError: raises an exception
"""
```

(b) reST

```
"""
This is an example of Google style.

Args:
    param1: This is the first param.
    param2: This is a second param.

Returns:
    This is a description of what is returned.

Raises:
    KeyError: Raises an exception.
"""
```

(c) Google



```
"""
This is an example of Numpydoc style.

Parameters
----------
 first : array_like
     the 1st param name first
 second :
     the 2nd param
 third : {'value', 'other'}, optional
     the 3rd param, by default 'value'

Returns
-------
 string
     a value in a string

Raises
------
  KeyError
     when a key error
"""
```

(d) Numpydoc

Figure 7: Exemplary docstrings

Our data set was investigated in terms of the docstring conventions used, the results are shown in the plot 8.

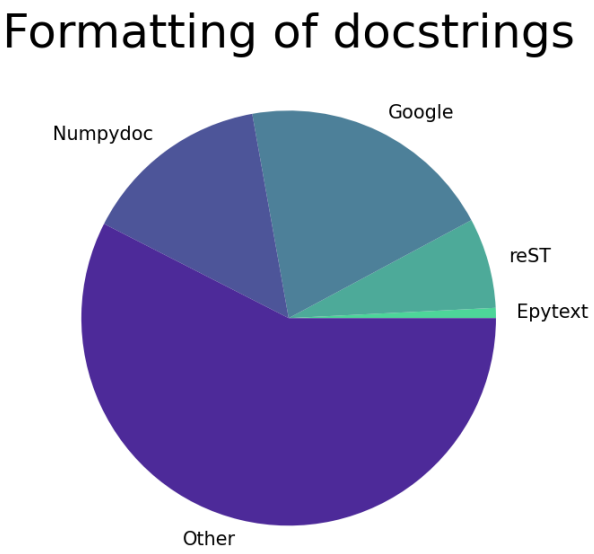


Figure 8: Used docstring conventions

Sadly, most of the documentation does not adhere to any of the conventions mentioned above and is often largely simplified, making it harder to gather sensible data from them.

4 Features extraction and clustering

The second milestone of this project was to gain real insights into the primary data source, the functions body. To achieve this, feature extraction and clustering techniques were applied to analyze the data in a meaningful way. Before settling on the chosen techniques, multiple alternative approaches were explored and evaluated. The goal was to transform the text into a vector that will represent the meaning of the snipped.

When analyzing code snippets, the goal was to convert the raw text into dense low-dimensional embedding vectors. Each direction in the embedding vector was designed to capture a distinct semantic or structural property of the input. For example - one direction can represent the meaning of testing the code (correlated with asserts, and test keywords, but also certain structure of the code) and another direction can represent dense mathematical notation. This representation enables us for more meaningful representation for downstream tasks like clustering.

4.1 Other tested techniques of feature extraction

4.1.1 TF-IDF Encoding

In the initial approach, we encode all keywords from the code snippets into a single vector. This method is conceptually simple but comes with significant drawbacks, particularly in handling the high dimensionality of the resulting vector.[9]

The resulting feature vector from this approach is quite large, with a shape of `(551,657, 443,641)`. This means that each code snippet is represented by a vector with over 443,000 dimensions, one for each unique keyword or feature extracted from the entire code corpus.

We might be able to improve this feature vector by replacing variable names (with phrases like var1, var2, etc.). Or, dividing them into smaller tokens. As default the tokenizer will treat phrases like `varible_one` as one token.

- PCA - is effective for reducing dimensions in datasets with linear relationships. It is also implemented in DASK for distributed computing.[9]
- TruncatedSVD - A more flexible dimensionality reduction method, especially effective for sparse datasets (like text or code). With 500 dims it can keep 40% of variance.

The TF-IDF approach, while commonly used in text processing[10], has certain limitations when it comes to capturing the richness of code semantics and relationships between terms in the context. Even other tested techniques like Word2vec have issues with understanding the deep relationships between tokens.

4.2 Embeddings

For the final approach, we moved on and embraced embeddings generated using transformer-based models. This approach allows for unconstrained embeddings that can react to any complexity within the text.

For this project, we utilized the `JinaAI/Jina-embeddings-v2-base-code` model[6], a specialized transformer-based model finetuned for generating code embeddings. Embeddings have 768 dimensions that capture the full meaning of the snippet.

	0	1	2	3	4	5	6	7	8	9	...	758	759	760	761	762	763	764	765	766	767
0	0.242721	0.416596	-0.247983	-0.269344	-0.045771	0.003983	-0.643093	0.709781	0.536804	-0.393591	...	-0.278954	-0.444132	0.322713	-0.328574	-0.341612	0.522995	-0.866399	0.181326	0.082756	0.199519
1	0.050176	0.300104	0.043600	0.085409	0.293300	0.054499	-0.394082	0.624031	0.046269	-0.905075	...	-0.129070	-0.091093	0.291247	-0.680475	-0.360574	0.428429	-0.698832	0.035107	0.150640	0.537492
2	-0.073493	-0.554840	-0.154205	1.224608	-0.021853	0.167990	-0.546410	0.434660	0.372562	0.506185	...	0.341457	0.355590	0.062959	0.187084	0.297098	0.605960	-0.244980	0.410118	0.038704	0.505363
3	-0.376492	0.100939	0.185906	-0.195942	-0.133067	-0.308304	-0.164811	-0.006943	-0.158194	-0.275358	...	-0.267996	-0.301235	0.391637	0.086361	0.163313	0.592806	0.013871	0.141489	0.321801	-0.102912
4	-0.497712	0.457880	0.340294	0.525480	-0.370195	-0.243208	0.085796	0.192543	-0.146180	-0.187930	...	-0.331745	-0.292625	0.762267	0.540826	-0.029412	0.565210	-0.056860	0.214326	0.237813	0.392732

5 rows × 768 columns

Figure 9: Sample of the embeddings

We can calculate cosine similarity to find snippets that align well with a given one. In Figure 10, you can see that the snippets with high similarity share the same structure and meaning. Conversely, you could attempt to find vectors that are most unaligned and point in opposite directions. However, it is typically challenging to find snippets that have perfectly opposite meanings. A cosine similarity of 0 indicates that the vectors

are perpendicular, meaning they have no relation or shared features. For vectors with a cosine similarity of < 0, this implies they are pointing in opposite directions, representing dissimilar or contrasting meanings.[9] For given snippet the least similar snippets had cosine similarity equal to -0.21143962. And these snippets were absolutely unrelated to datetime.

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\|\|\vec{B}\|}$$

	0	idx = TimedeltaIndex([timedelta(days=1), timed
1	1.000000	idx = TimedeltaIndex([100000000000000.0, 'NaT'...
299873	0.934841	idx = PeriodIndex(['2016-05-16', 'NaT', NaT, n...
161990	0.877274	idx = DatetimeIndex(['2016-05-16', 'NaT', NaT,...
544855	0.843352	idx = DatetimeIndex(['2016-05-16', 'NaT', NaT,...
0	0.842460	idx = TimedeltaIndex([timedelta(days=1), timed...
480566	0.827040	idx = TimedeltaIndex([100000000000000.0, 'NaT'...
111992	0.826613	idx = TimedeltaIndex([100000000000000.0, 'NaT'...
335844	0.813452	idx = timedelta_range(start='1 days', periods=...
252351	0.811320	idx = Index([NaT.asm8] * 2, dtype=object) msg ...
150931	0.807264	idx = DatetimeIndex([datetime(2013, 1, 1), dat...

Figure 10: Snippets that have highest cosine similarity to `dx = TimedeltaIndex([timedelta(days=1), timed ...`

4.3 Clustering

For clustering, we were limited to KMeans, it is a fast and scalable clustering algorithm that can be applied via `DASK`. Alternative approach - AglomerativeClusteering is a lot more compute intense and for us, always yield worse results. Probably because it does not scale well with this size of data.

Every clustering was finetuned using `silhouette_score`[9]. Because of the compute complexity of this metric and time required for clustering we scored and cluster them on subset of data.

- Clustering - 200000 (Around 35% of dataset)
- Scoring - 25000 (Around 5% of dataset)

Scoring and clustering are noisy, but *kolanko* can be easily seen[9]. This technique was applied every time clustering have been done. We often did that in few batches when we spotted places that requires fine tuning as clustering is taking a long time we were not able to test all cluster counts. In Figure 13 we see two plots: one with step 10 and another with step 5

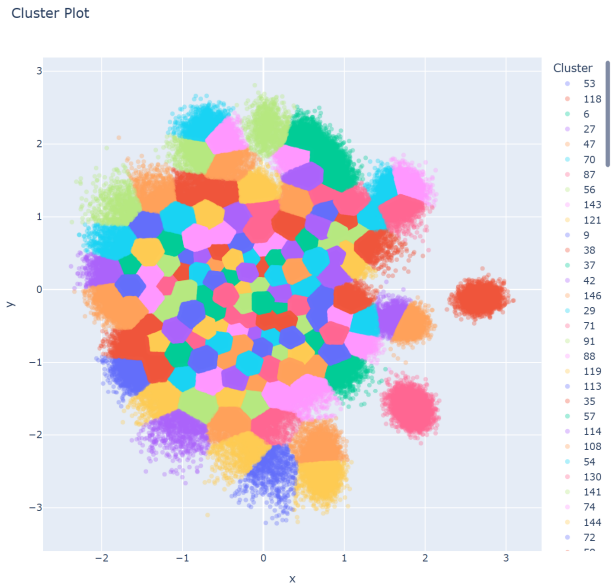


Figure 11: Example of too many clusters in clustering

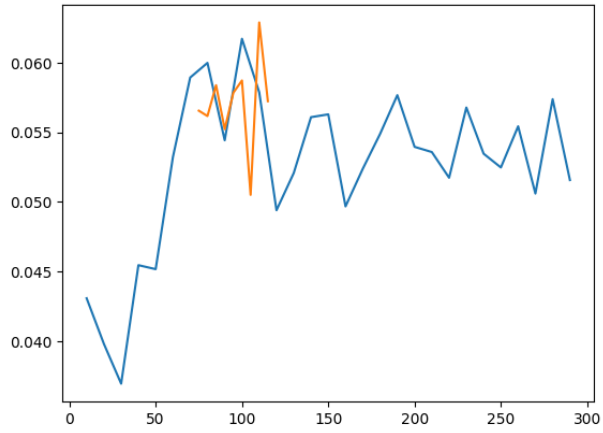


Figure 12: `silhouette_score` for different amounts of clusters

4.4 Visulization

After reducing dimensionality to 2 or 3, the whole dataset becomes manageable by graphing libraries like `plotly` (or `dash`). However, weaker devices require reducing the number of points to a few hundred thousand. We used `dash` as backend for our. The app allows for searching the latent space by embedding query and displaying subset that is closest by given the metric (Minkowsky, with variable p)

$$D(x, y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}}$$

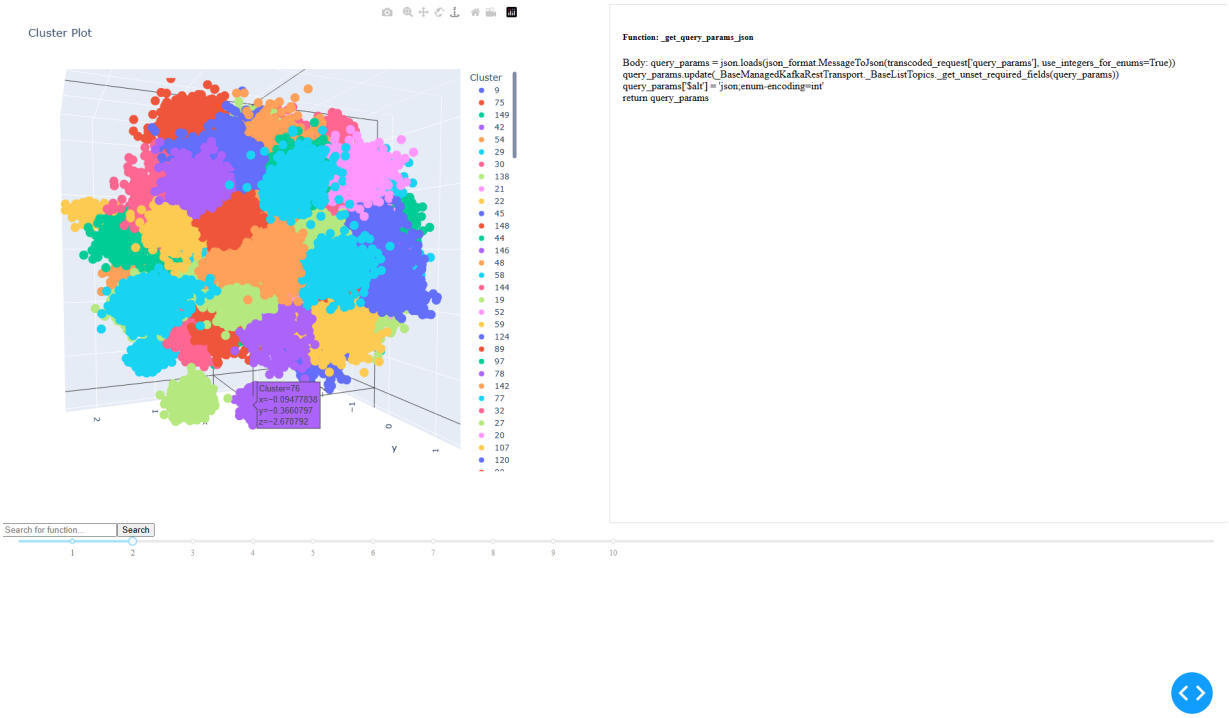


Figure 13: dash app with highlighted sample and corresponding code

4.5 Dim reduction

Dimensionality reduction is essential for achieving clear and readable plots. Unfortunately, embeddings often contain very dense information across all dimensions, and any attempt to reduce dimensionality can quickly lead to a loss of variance.

One approach involves first reducing the dimensions to a relatively small number for clustering while preserving as much variance as possible, followed by further reduction for display. While this may result in overlapping clusters due to alignment in reduced dimensionality, it can still yield meaningful results.

Other approach is to use nonlinear dimensionality reduction like TNSE or VAE and reduce dimensions as far as possible onto 2 or 3 and cluster on that.

The big issue with these algorithms was to find ones that can be applied to large dataset or applied only to subset. Because size of dataset prevents us from using any technique we want. And if we would like to scale this project -to include more data and get better insight we need to choose algorithms that can scale.

4.5.1 PCA

PCA is a simple yet effective algorithm for dim reduction and is also supported in DASK, so it can be applied in batches for whole dataset.

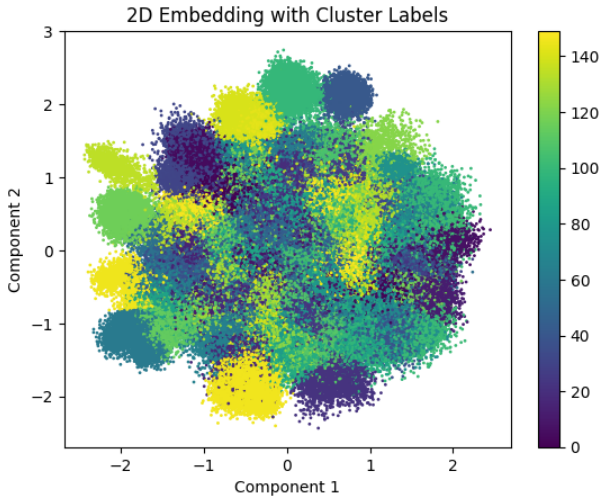


Figure 14: Clustering on embeddings reduced using PCA to 100 and then to 2 for display

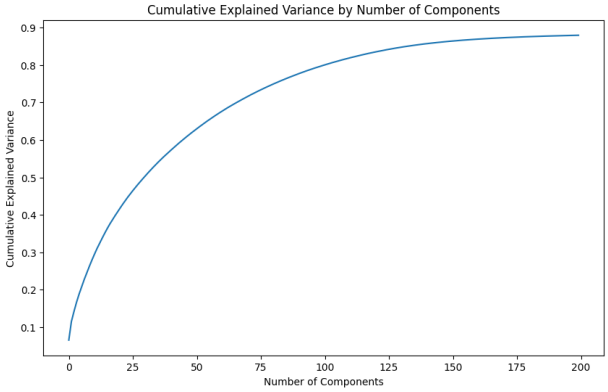
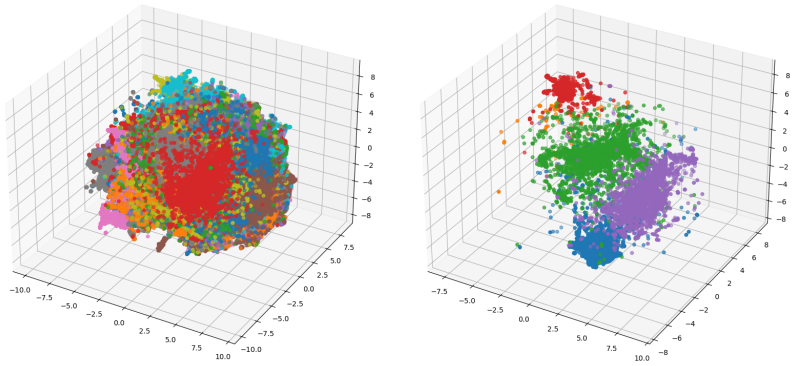


Figure 15: Variance preservation of PCA

‘

4.5.2 TSNE

t-SNE is a non-linear method for unwinding high-dimensional data, it builds graphs of samples and flattens out multidimensional relations between them.[2] It is computationally intensive and has to be fit on whole dataset and it makes it not ideal for this task. [9]

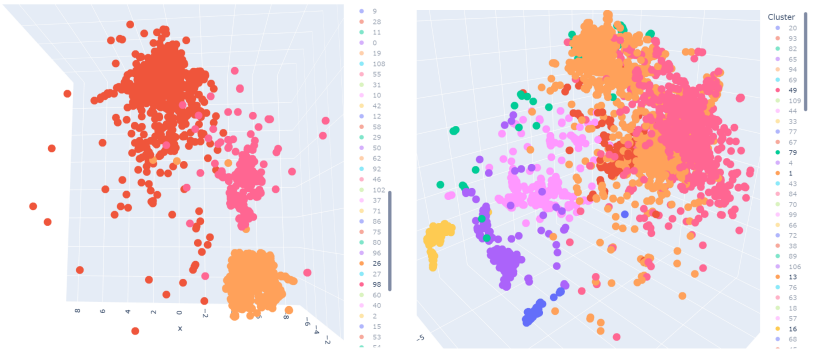


(a) All clusters reduced with TNSE

(b) Subset of visualization from Image a

Figure 16: 100 dimensions reduced with TNSE to 3

The clusters generated by this method overlap significantly and are strangely shaped.



(a) All clusters reduced with TNSE

(b) Subset of visualization from Image a

Figure 17: 100 dimensions reduced with TNSE to 3

4.5.3 LocallyLinearEmbedding

LLE preserves local structures well and works best on small to medium datasets. However, it is slow and less practical for large-scale applications, making it more suitable for subsets

or smaller data. Clusters separation generated by LLE was bad and had strange shapes.

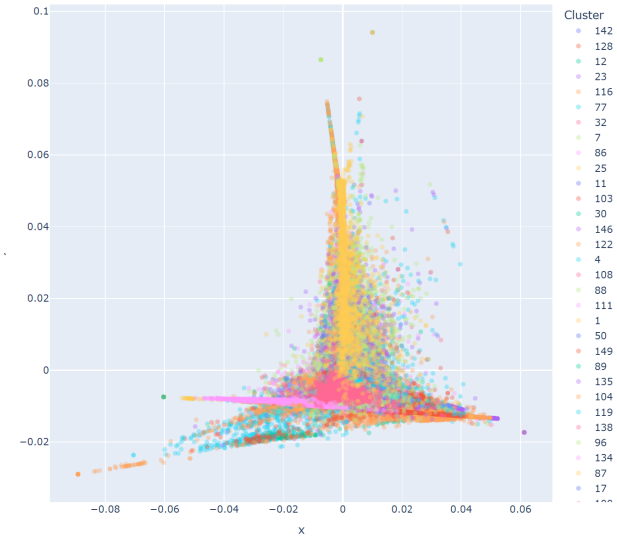


Figure 18: Clusters after LLE (not great)

4.5.4 Variational Autoencoder

VAEs is an autoencoder neural network that has some constrains on the encoding layer to enforce continuity and smoothens[4], excelling in capturing non-linear patterns and preserving structure of high-dimensional clusters. They **easily scalable**, and **can handle arbitrary large datasets**. We can also decode and encode information with ease.

Our AutoEncoder (snippet 2 and 3) is built from few dense layers, which gradually decreases from 768 into `latent_dim` that depends on task. Usually `latent_dim` is set to 2 or 3. On encoding layer there are standard constraints that are put onto the VAE model to enforce their properties of latent space.

```python
class VariationalEncoder(nn.Module):
    def __init__(self, latent_dims):
        super(VariationalEncoder, self).__init__()
        self.linear1  = nn.Linear(768, 512)
        self.linear11 = nn.Linear(512, 512)
        self.linear2  = nn.Linear(512, latent_dims)
        self.linear3  = nn.Linear(512, latent_dims)

        self.N = torch.distributions.Normal(0, 1)
        self.N.loc = self.N.loc.cuda() # hack to get sampling on the GPU
        self.N.scale = self.N.scale.cuda()
        self.kl = 0

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = F.elu(self.linear1(x))
        x = F.elu(self.linear11(x))
        mu =  self.linear2(x)
        sigma = torch.exp(self.linear3(x))
        z = mu + sigma*self.N.sample(mu.shape)
        self.kl = (sigma**2 + mu**2 - torch.log(sigma) - 1/2).sum()
        return z

class Decoder(nn.Module):
    def __init__(self, latent_dims):
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(latent_dims, 512)
        self.linear11 = nn.Linear(512, 512)
        self.linear2 = nn.Linear(512, 768)

    def forward(self, z):
        z = F.elu(self.linear1(z))
        z = F.elu(self.linear11(z))
        z = F.elu(self.linear2(z))
        return z
```

Code Snippet 2: Vae

```python
class VariationalAutoencoder(nn.Module):
    def __init__(self, latent_dims):
        super(VariationalAutoencoder, self).__init__()
        self.encoder = VariationalEncoder(latent_dims)
        self.decoder = Decoder(latent_dims)

    def forward(self, x):
        z = self.encoder(x)
        return self.decoder(z)
```

Code Snippet 3: Assembled VAE

Our training parameters were as follows:

- `latent_dim` - 3 or 2
- `batch_size` - 10000
- `lr` - decreased from 0.03 to $1e-6$ in epoches 3, 10 and 20
- `epoch` - 30

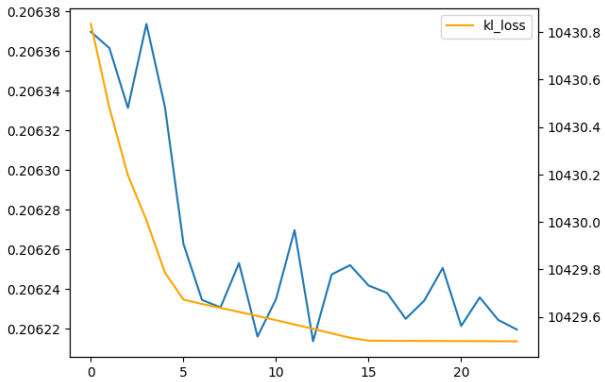


Figure 19: Example training curve of VAE autoencoder - kl loss and reconstruction loss)

VAE was used as the final approach. It creates very clear clusters with almost spherical shapes and can "draw" clear boundaries between them if possible even when amount of dimensions is low.

4.6 Results

As a result, we chose 150 clusters in the 3-dimensional latent space generated by the VAE. Clustering clearly separates similar code snippets into distinct clusters. Additionally, our visualization application allows for searching the space for specific keywords or sentences to find subsets that match a given query.

Examples of clustering samples are provided in Appendix A. Overall, clusters group code snippets with similar structure, meaning, and purpose. Many clusters have a very distinct shape and they consists of similar snippets usually with with same code structure differing in variable names or values.

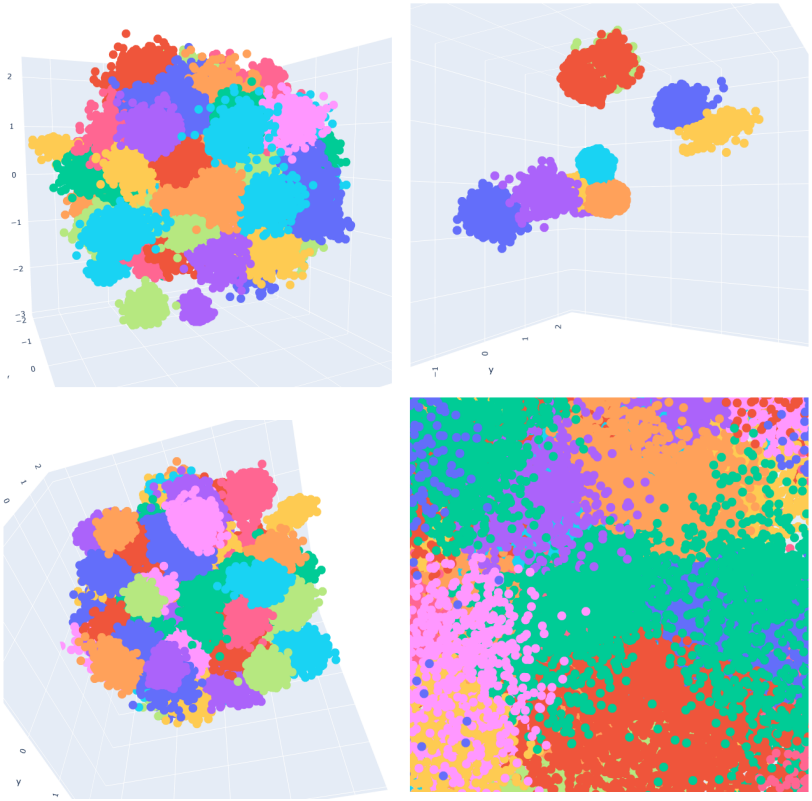
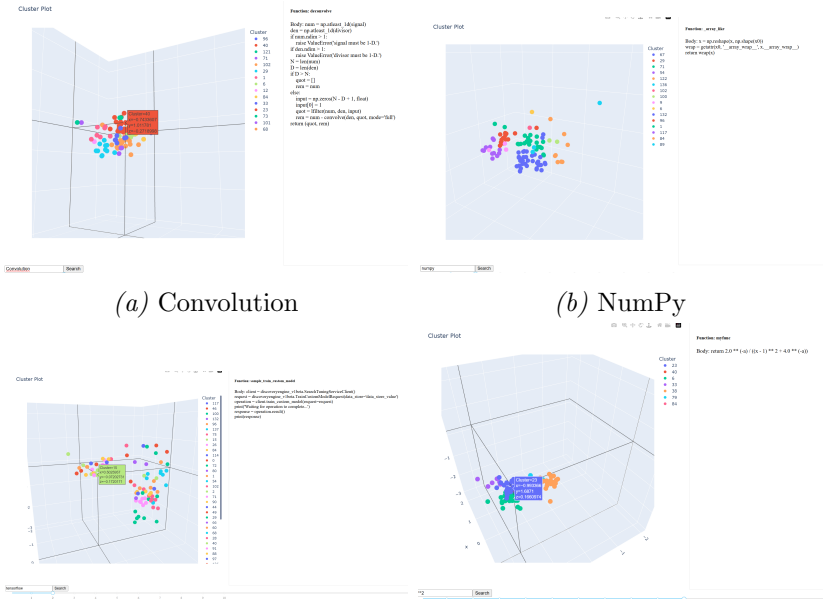


Figure 20: Final Clustering

By default, searching returns 100 samples closest to the embedded query using the **Minkowski metric** with $p = 2$ (Euclidean distance). The search is performed on the original embedding, so the resulting subset does not always resemble a sphere in latent space (although it sometimes does, especially for narrow topics). The parameter p can be adjusted to modify the search results.



(a) Convolution *(b)* NumPy

(c) TensorFlow *(d)* x^{**2}

Figure 21: Different search queries and resulting subsets

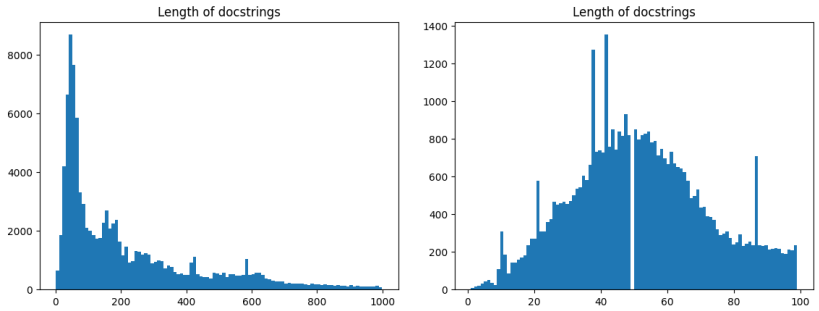
5 Training the Transformer

The last goal of the project was to analyze the docstrings. However, the original dataset contains only 120k samples with documentation included. To address this limitation, we generated docstrings using a pretrained transformer model. Unfortunately, these small models perform poorly on general tasks and do not yield fully usable results.

The **best solution is to fine-tune a transformer on a summarization task**. For this purpose, we use the model `google/pegasus-cnn_dailymail`[11], as the previously used model was not designed for sequence-to-sequence tasks. This is a smaller model designed for summarization out of the box but can perform other tasks as well.

5.1 Dataset

From our database, we extracted as many code snippets and corresponding docstrings as possible. Some docstrings are quite lengthy, **so we utilized the model to summarize them**, removing unnecessary parts such as parameter lists, examples, etc. The distribution of docstring lengths is shown in Figure 22. Long docstrings are result of including parameter names.



(a) Unbounded x-axis *(b)* (Zoom)

Figure 22: Histogram of docstring lengths

The query used for summarization is as follows:

```
Docstring: {doc}.
Function body: {body[:100] if len(body) > 100 else body};
Summary:
```

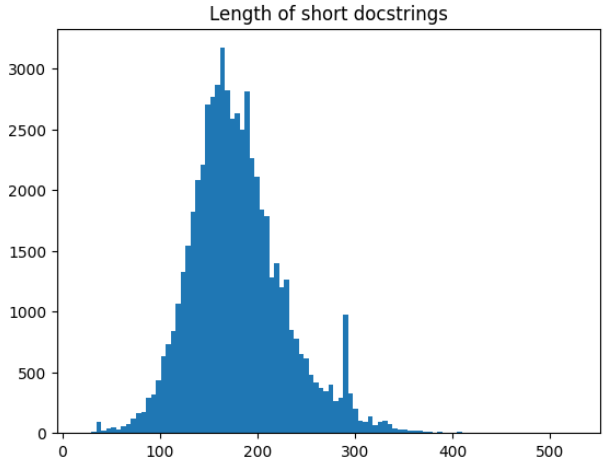


Figure 23: Lengths of short docstrings

Some of the generated data was meaningless, **so we removed some samples based on their length**. As we found, it is a good metric to determine if data were badly generated.

```
604                .<n>  .<n> .<n> .<n> .<n> .<n> .<n> .<n> .
950                      .<n> .<n> .<n> .<n> .<n> .<n> .<n> .
54948                    .<n> .<n> .<n> .<n> .<n> .<n> .<n> .
1508                     .<n> .<n> .<n> .<n> .<n> .<n> .<n> .
1715                      t t t t t t t t t t .<n>t t t t t .
8897                      K[x], K[x], K[x], K[x], K[x], K[x].
```

1968 K[x], K[x], K[x], K[x], K[x], K[x].
2009 .<n> .<n> .<n> .<n> .<n> .<n> .<n> .

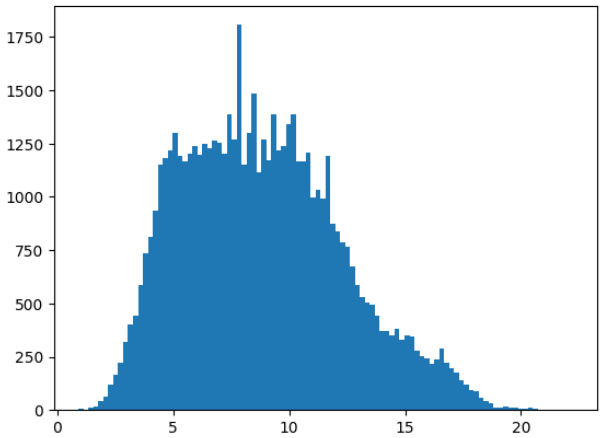


Figure 24: Distribution of similarity of original doc-string and summarized one. (mse). We discarded samples with error higher then 10.

We also checked the distance between the original and generated docstrings using mse and rejected samples with a distance below a specified threshold (we have selected a threshold equal to 10 based on histogram 24). The complete algorithm is shown in Snippet 4.

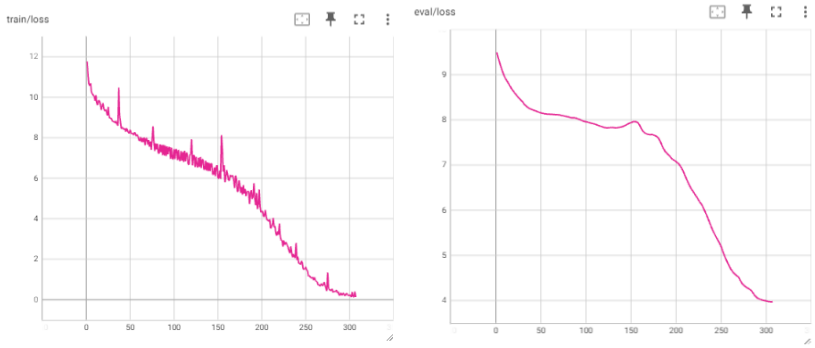
```python
SIMILLARITY_THRESHOLD = 10

short['docstring'][distances > SIMILLARITY_THRESHOLD]


def filter(docstring, short_docstring, distance):
    if len(docstring) < 85:
        return docstring

    if distance > SIMILLARITY_THRESHOLD and len(docstring) < 120:
        return docstring

    if distance < SIMILLARITY_THRESHOLD and len(short_docstring) > len(docstring):
        return None

    return short_docstring

short['filtered_docstring'] = short.apply(lambda x: filter(x['docstring'], x['short_docstring'], distances[x.name]), axis=1)

short
```

Code Snippet 4: Algorithm for choosing valid docstrings

5.2 Training

We finetuned the model in seq2seq task with methods described in Tunstall book[10]. We fed a function as the input and expected docstrings in the output.

1. learning_rate - `5e-05`
2. num_train_epochs - `500`
3. per_device_train_batch_size - `1`
4. per_device_eval_batch_size - `8`
5. seed - `42`
6. save_steps - `15`



(a) Train loss *(b)* Eval loss

Our tests reveal that models trained beyond episode 150 experience over-fitting, resulting in the generation of nonsensical outputs. So, our final model is checkpoint from episode 150 that performs best in embedding test and has the most meaningful results.

5.3 Results

The best way to show model performance is to proof it by generating a bunch of samples and comparing them to the reference points. With generated embeddings for:

1. Body of the function
2. Original docstrings included in function - it generally should well align with the body of the function.
3. Docstrings summarized by general auto regressive transformer (`google/pegasus-cnn_dailymail`[11]).
4. Docstrings generated based on body of the function (without docstring) by finetuned model

We should also note that Task 3 is significantly easier compared to Task 4, and Task 4 is in fact the objective of this stage. In Figure 26, we present three density plots illustrating

the cosine similarity between the embeddings of the docstrings and the embeddings of function bodies. These embeddings are generated using the `jinaai/jina-embeddings-v2-base-code`[6] model.

The **orange** plot shows the best alignment with the code, representing the original unmodified docstrings. In contrast, the **green** plot corresponds to docstrings generated by a general transformer model, which demonstrates poor performance. The mean similarity is barely above zero, and many samples have a similarity below zero, indicating that the generated text often does not have a meaningful connection to the code. The central **blue plot represents our fine-tuned model**. Although the summarization is not perfect, almost all generated docstrings have a similarity above zero, with some achieving scores above 0.5. **This demonstrates that the fine-tuned model performs better than the original and is capable of successfully summarizing Python code into short docstrings.**

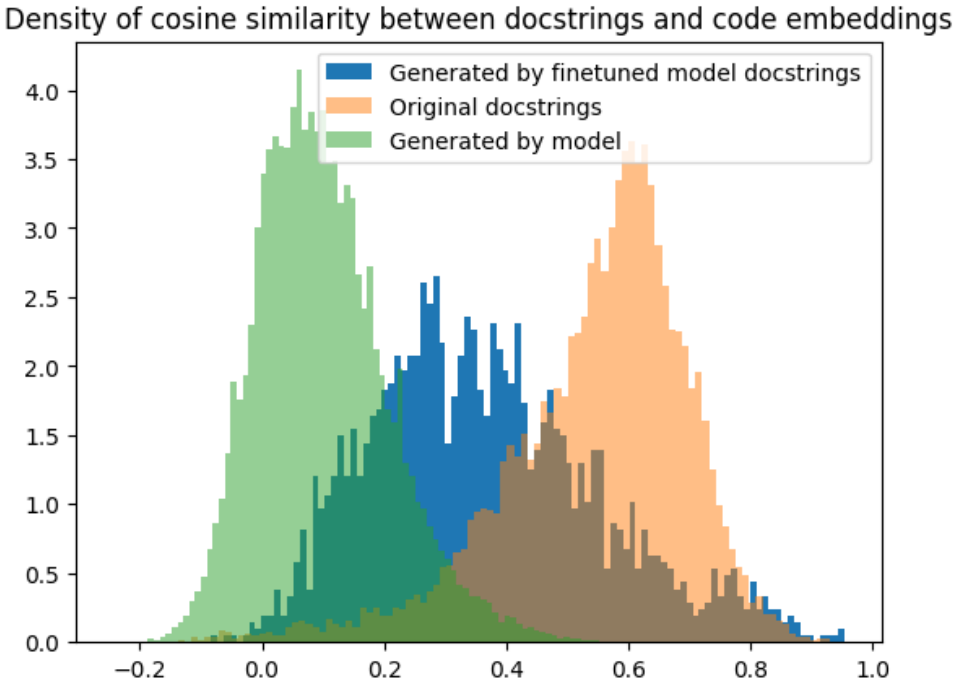


Figure 26: Distribution of cosine-similarity of docstring embeddings with embeddings of corresponding code

In Figure 27 you can see samples of docstrings generated by our model (150 checkpoint).

(a) The following code describes a multidimensional index using the Timedelta method .¡n¿ The standard deviations of the index are given for each day as a sequence .¡n¿ If 1 day or 2 days are provided, their length must match the number of days.

(b) The following code describes a multidimensional array using the standard multidimensional format .¡n¿ The standard deviations of the np.array are given for each label as a sequence .¡n¿ If " scalars" are provided, their length must match the number of labels.

(c) The following code describes a multidimensional frame using the standard Gaussian process .¡n¿ The standard deviations of the frame are given for each character as a sequence .¡n¿ If 'bar' is provided, the length of the frame must match the number of characters .

Figure 27: Code and generated descriptions. More in source code [12]

6 Possible farther research

1. Advanced Fine-tuning - In the current form, finetuneing is quite limited by method. Because we were using well responding embedding model **it is possible to perform neural network destilation** with respect to code-embedding transformer and dataset to seq2seq transformer used. In this case we would maximize the similarity of generated docstring embeddings with code embeddings. My intuition says that this method may lead to a very good summarization of the code. This task may require us to regularize the training to prevent model from generating plain code.
2. Two-step dimensionally reduction using neural networks - VAE can outperform other dim-reduction techniques, but if we would apply clustering on higher dimension latent space, we might be able to get better clustering. We could add a cluster preserving loss function, for example, incorporating centroid preservation loss or contrastive learning. Both techniques are differentiable if we precompute clusters.
3. Expanding Dataset - **This project has the potential to be run on a bigger scale** for even more advanced clustering and text generation. **Current size of dataset is approximately 3% of all available repositories**. It is only a small fraction of whole data that can be scrapped, and using a linear approximation the whole dataset may consist of 16-20 million code snippets.

References

[1] dask. *dask documentation*. URL: https://docs.dask.org/en/stable/ (visited on 12/15/2024).

[2] scikit-learn developers. *scikit-learn documentation*. URL: https://scikit-learn.org (visited on 12/15/2024).

[3] Epydoc. *Epydoc documentation*. URL: https://epydoc.sourceforge.net/ (visited on 12/22/2024).

[4] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd. O'Reilly Media, 2019. ISBN: 978-1-4920-3264-9.

[5] Google. *Google formating instructions*. URL: https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings (visited on 12/22/2024).

[6] Michael Günther et al. *Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents*. 2024. arXiv: 2310.19923 [cs.CL]. URL: https://arxiv.org/abs/2310.19923.

[7] Numpy. *Numpy style guide*. URL: https://numpydoc.readthedocs.io/en/latest/format.html (visited on 12/22/2024).

[8] Sphinx. *Sphinx documentation*. URL: https://www.sphinx-doc.org/en/master/ (visited on 12/22/2024).

[9] Marcin Szpyrka. *Advanced Data Mining*. Lecture. Katedra Informatyki Stosowanej, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, 2024/2025.

[10] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural Language Processing with Transformers, Revised Edition*. O'Reilly Media, 2022. ISBN: 978-1-0981-3679-6.

[11] Jingqing Zhang et al. *PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization*. 2019. arXiv: 1912.08777 [cs.CL].

[12] Złotorowicz Maciej, Ryszard Błażej. *Code repository*. URL: https://github.com/Lord225/datamining-on-pypl (visited on 12/15/2024).

A Samples from clusters

Cluster 41

```
Body: http_options: List[Dict[str, str]] = [{'method': 'patch',
  'uri': '/v2/{generator.name=projects/*/locations/*/generators/*}',
  'body': 'generator'}]
```

- -

```
Body: http_options: List[Dict[str, str]] = [{'method': 'post', 'uri': '/v1/{analysis_query.scope=*/*}:
    analyzeIamPolicyLongrunning', 'body': '*'}]
return http_options
```

- -

```
Body: http_options: List[Dict[str, str]] = [{'method': 'post', 'uri': '/v1beta1/{environment=projects
    /*/locations/*/environments/*}:stopAirflowCommand', 'body': '*'}]
return http_options
```

- -

```
Body: http_options: List[Dict[str, str]] = [{'method': 'patch', 'uri': '/v2/{generator.name=projects
    /*/locations/*/generators/*}', 'body': 'generator'}]
return http_options
```

Cluster 134

```
Body: self._wrapped_methods = {self.list_available_versions: self._wrap_method(self.
    list_available_versions, default_timeout=None, client_info=client_info), self.list_instances: self
    ._wrap_method(self.list_instances, default_timeout=None, client_info=client_info), self.
    get_instance: self._wrap_method(self.get_instance,
```

- -

```
Body: self._wrapped_methods = {self.list_available_versions: self._wrap_method(self.
    list_available_versions, default_timeout=None, client_info=client_info), self.list_instances: self
    ._wrap_method(self.list_instances, default_timeout=None, client_info=client_info), self.
    get_instance: self._wrap_method(self.get_instance, default_timeout=None, client_info=client_info),
     ...
```

- -

```
Body: self._wrapped_methods = {self.get_encryption_spec: self._wrap_method(self.get_encryption_spec,
    default_timeout=None, client_info=client_info), self.initialize_encryption_spec: self._wrap_method
    (self.initialize_encryption_spec, default_timeout=None, client_info=client_info), self.
    get_location: self._wrap_method(self.get_location, default_timeout=None, client_info=client_info),
     self.list_locations: self._wrap_method(self.list_locations, default_timeout=None, client_info=
    client_info), self.cancel_operation: self._wrap_method
```

- -

```
Body: self._wrapped_methods = {self.create_workload: gapic_v1.method.wrap_method(self.create_workload,
      default_timeout=None, client_info=client_info), self.update_workload:
```

Cluster 12

```
Body: np.random.seed(1234)
x = np.random.randn(5)
mean = np.random.randn(5)
cov = np.abs(np.random.randn(5))
d1 = multivariate_normal.logpdf(x, mean, cov)
d2 = multivariate_normal.pdf(x, mean, cov)
assert_allclose(d1, np.log(d2))
```

- -

```
Body: rng = np.random.RandomState(0)
(Y, _, _) = generate_toy_data(3, 65, (8, 8), random_state=rng)
spca_lars = SparsePCA(n_components=3, method='lars', random_state=rng)
U1 = spca_lars.fit_transform(Y)
spca_lasso = SparsePCA(n_components=3, method='cd', random_state=rng)
U2 = spca_lasso.fit(Y).transform(Y)
assert_array_almost_equal(U1, U2)
```

- -

```
Body: np.random.seed(12345)
vals = np.random.lognormal(size=100)
weights = np.random.choice([1.0, 10.0, 100], size=vals.size)
orig_weights = weights.copy()
stats.gaussian_kde(np.log10(vals), weights=weights)
assert_allclose(weights, orig_weights, atol=1e-14, rtol=1e-14)
```