

# Hugging Face Model Recommender System (HFMRs)

## Table of Contents

[Introduction](#)

[Literature Review / Background](#)

[Methodology](#)

[Dataset](#)

[Exploratory Data Analysis \(EDA\)](#)

[Implementation](#)

[Inference](#)

[System Architecture](#)

[Streamlit Application](#)

[FastAPI + Chromium Extension](#)

[Challenges](#)

[Developing Recommenders without User-Product Interactions](#)

[Lack of Universal Metrics and Gold Label Datasets](#)

[Future Work](#)

[Explore Different Word Embeddings](#)

[Consider Representation of Tasks](#)

[Model Updates](#)

[Cloud Deployment](#)

[Chromium Extension](#)

[Conclusion](#)

[References](#)

[Misc](#)

## Introduction

The discipline of Artificial Intelligence (AI) is expanding rapidly, and pre-trained models are increasingly being used to build and fine-tune AI models. HuggingFace is an AI community that hosts open-source models for Natural Language Processing (NLP), Computer Vision (CV), and other related fields that AI practitioners can leverage to build their own AI applications. As of

writing, HuggingFace boasts an impressive collection of over 120k models, 20k datasets, and 50k demos, making it an excellent resource for AI practitioners worldwide.

However, with such a vast number of models available, it can be overwhelming for AI practitioners to discover similar models to perform benchmarking. Currently, HuggingFace relies on a traditional search function to help users to find the models they need, which can be limiting since users may need an easier way to discover similar models that may be useful for their project.

In this article, we will discuss the implementation of a recommender system, Hugging Face Model Recommender System (HFMRs), that can suggest the most relevant models for their needs, improving the user experience and making it easier and faster to find similar models.

## Literature Review / Background

Recommender systems have become an integral part of modern society, serving a variety of purposes, such as personalized recommendations for online shopping, music and video streaming services, and social networks.

Some of the common techniques for building recommender systems include collaborative filtering and content-based filtering.

Collaborative filtering is one of the most widely used approaches to building recommender systems. This technique relies on the intuition that users with similar past preferences will have similar tastes in the future and uses this similarity to generate recommendations. Collaborative filtering can be further subdivided into two types: user-based and item-based. User-based collaborative filtering recommends items to a user based on the items similar users have liked. In contrast, item-based collaborative filtering recommends items to a user based on the items they have picked in the past.

Content-based filtering is another popular approach to building recommender systems. This technique relies on the idea that users with similar preferences will like items with similar attributes. Content-based filtering systems typically use information about the item's features, such as genre or artist, to generate recommendations.

For our project, we are limited to using content-based filtering as the dataset we have curated does not include user-specific data. Collaborative filtering, although more effective in many cases, requires user-based data and cannot be applied in our situation.

Graph neural networks (GNNs) have emerged as a promising approach for building recommender systems. GNNs can model complex relationships between users and items in a graph structure and can incorporate both explicit and implicit feedback to generate more accurate recommendations.

One of the key advantages of GNNs is their ability to capture high-order dependencies between items and users, which can be especially useful in situations where users have unique and complex preferences. Additionally, GNNs can be combined with other approaches, such as collaborative filtering, to improve the overall performance of the recommender system.

In our case, we used GNN for content-based recommendations. In a content-based recommendation system, the recommendations are made based on the similarity of items' attributes. A graph is constructed where the nodes represent items, and the edges represent the similarity between the items' attributes, where the similarity is measured using cosine similarity obtained from the pre-trained word embeddings.

The GNN can then be trained to capture complex relationships between the item features and identify similar items for recommendation.

## Methodology

### Dataset

We used the Hugging Face Model dataset that is readily available from HuggingFace's [Transformers Python Library](#). The library provides a [list\\_models\(\)](#) API that allows us to query the model metadata from their repository. The API allows us to query the data with different granularity by adding parameters such as sorting and filtering.

For this project, we will use the top 10 thousand models sorted by downloads for recommendations.

Below is the data dictionary of the model metadata used.

Field	Description	Example
modelId	Unique identifier for the model.	albert-large-v1

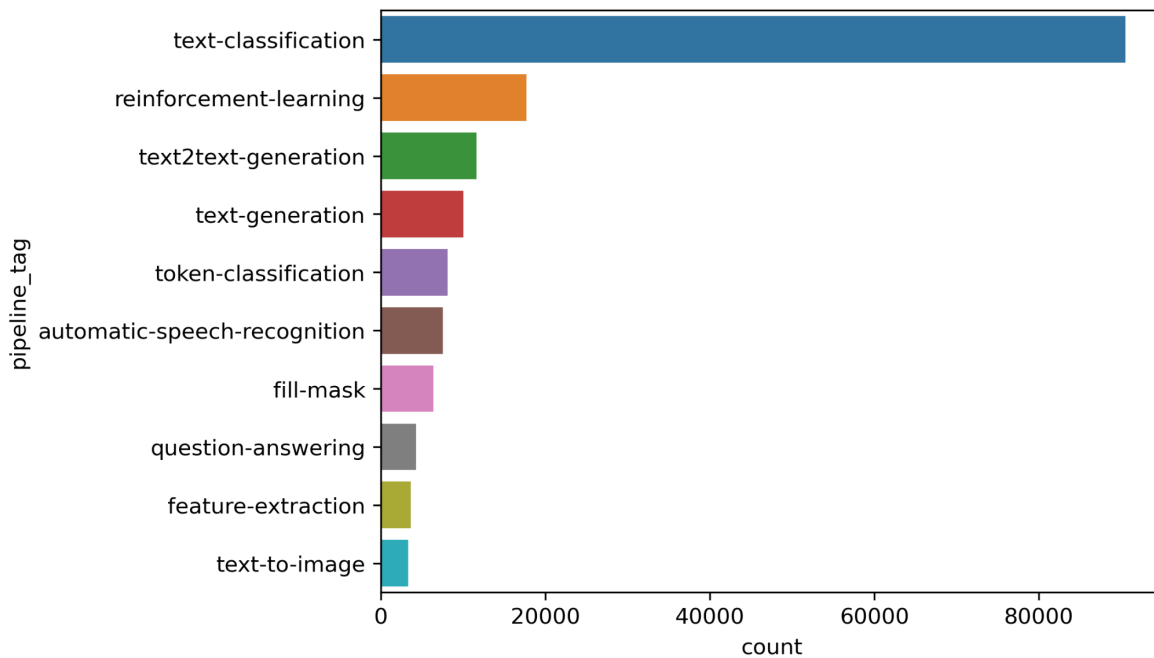
sha	Version control identifier for the model.	17aeb59edfd7c7732fb96248a95f5c271a9fa28f
lastModified	Date and time the model was last modified.	2021-01-13T15:29:06.000Z
tags	Additional information about the model, such as programming languages, tasks, language, datasets, and license.	['pytorch', 'tf', 'albert', 'fill-mask', 'en', 'dataset:bookcorpus', 'dataset:wikipedia', 'arxiv:1909.11942', 'transformers', 'license:apache-2.0', 'autotrain_compatible', 'has_space']
pipeline_tag	Type of task the model was specifically trained for.	fill-mask

siblings	Other files related to the model, such as configuration files and trained weights.	...
private	Whether the model is private or not.	FALSE
author	The author of the model.	None
config	Technical information about the model's architecture and type.	{'architectures': ['AlbertForMaskedLM'], 'model_type': 'albert'}
security Status	The security status of the model.	None
_id	Unique identifier for the model.	621ffdc036468d709f17432a
id	Same as modelId.	albert-large-v1

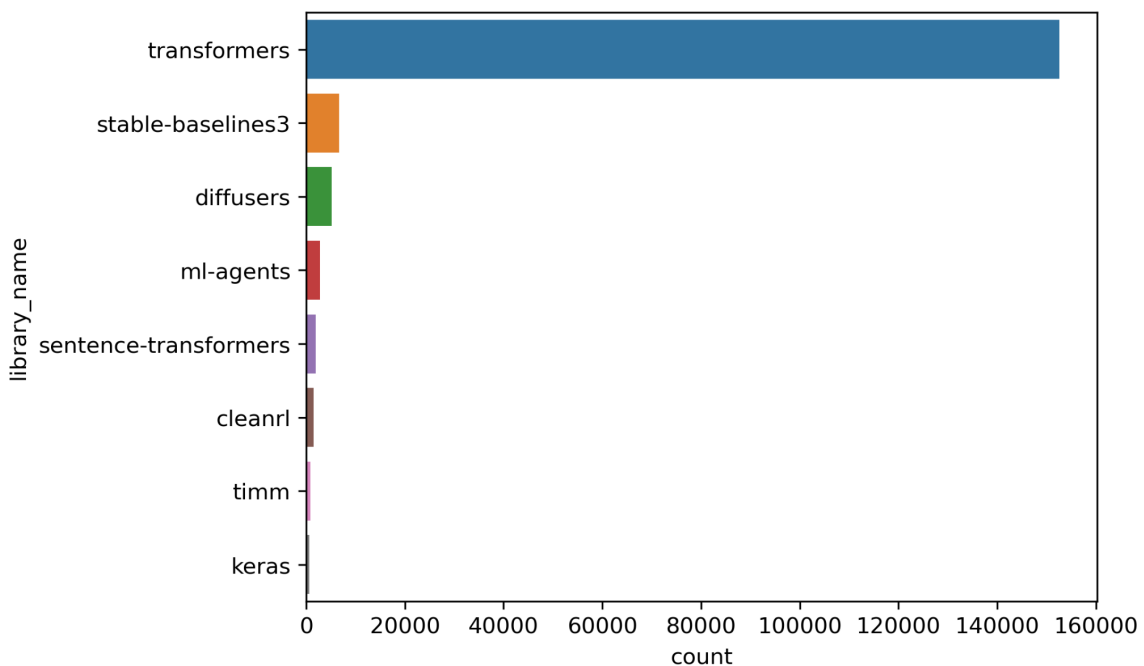
cardData	Additional information about the model, such as language, license, and datasets.	{'language': 'en', 'license': 'apache-2.0', 'datasets': ['bookcorpus', 'wikipedia']}
likes	The number of likes the model has received.	0
downloads	The number of downloads the model has received.	357
library_name	The name of the library that hosts the model.	transformers

*Table 1: Data Dictionary of Model Metadata*

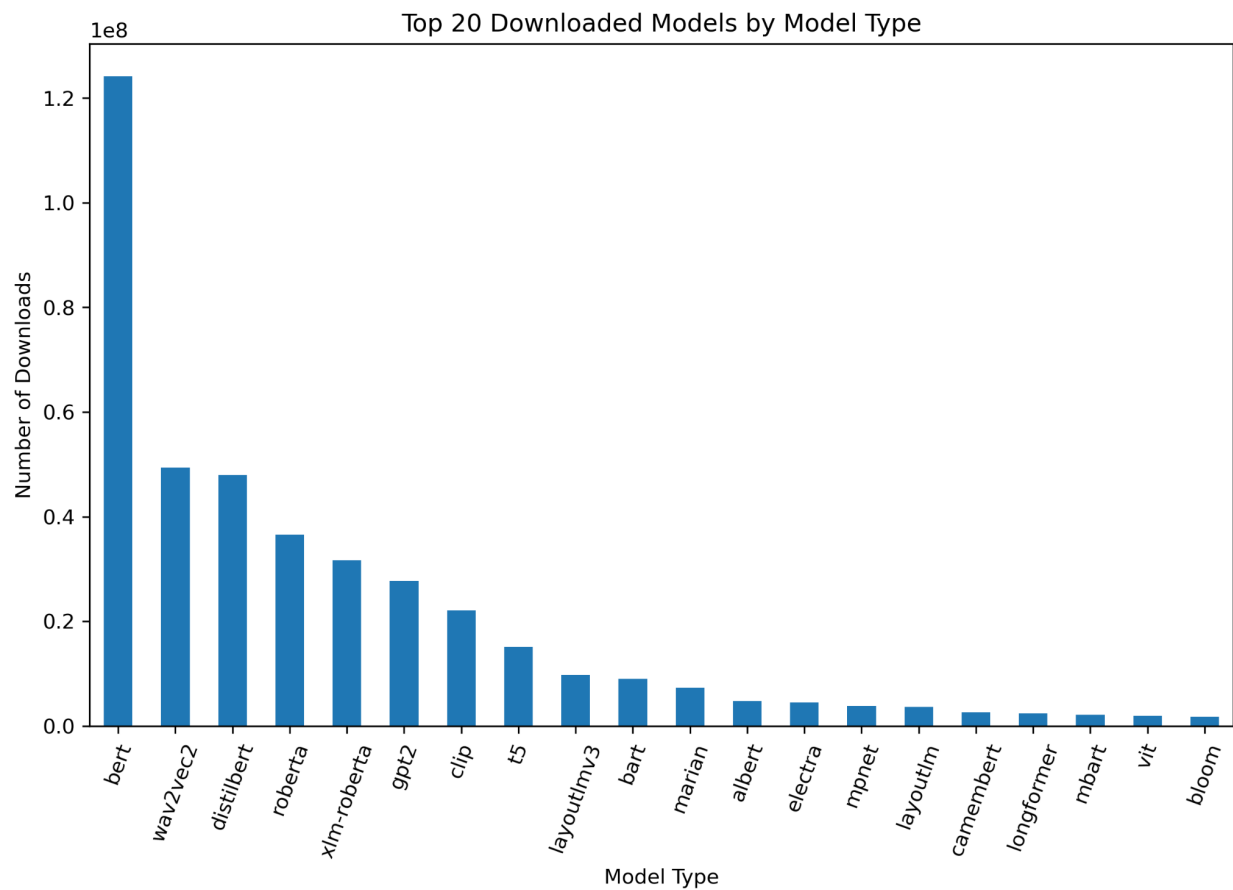
## Exploratory Data Analysis (EDA)



We count the number of models used for each type of task and sort them in descending order. We find the top 3 most popular tasks that those models were specifically trained for are text-classification, reinforcement-learning and text2text-generation

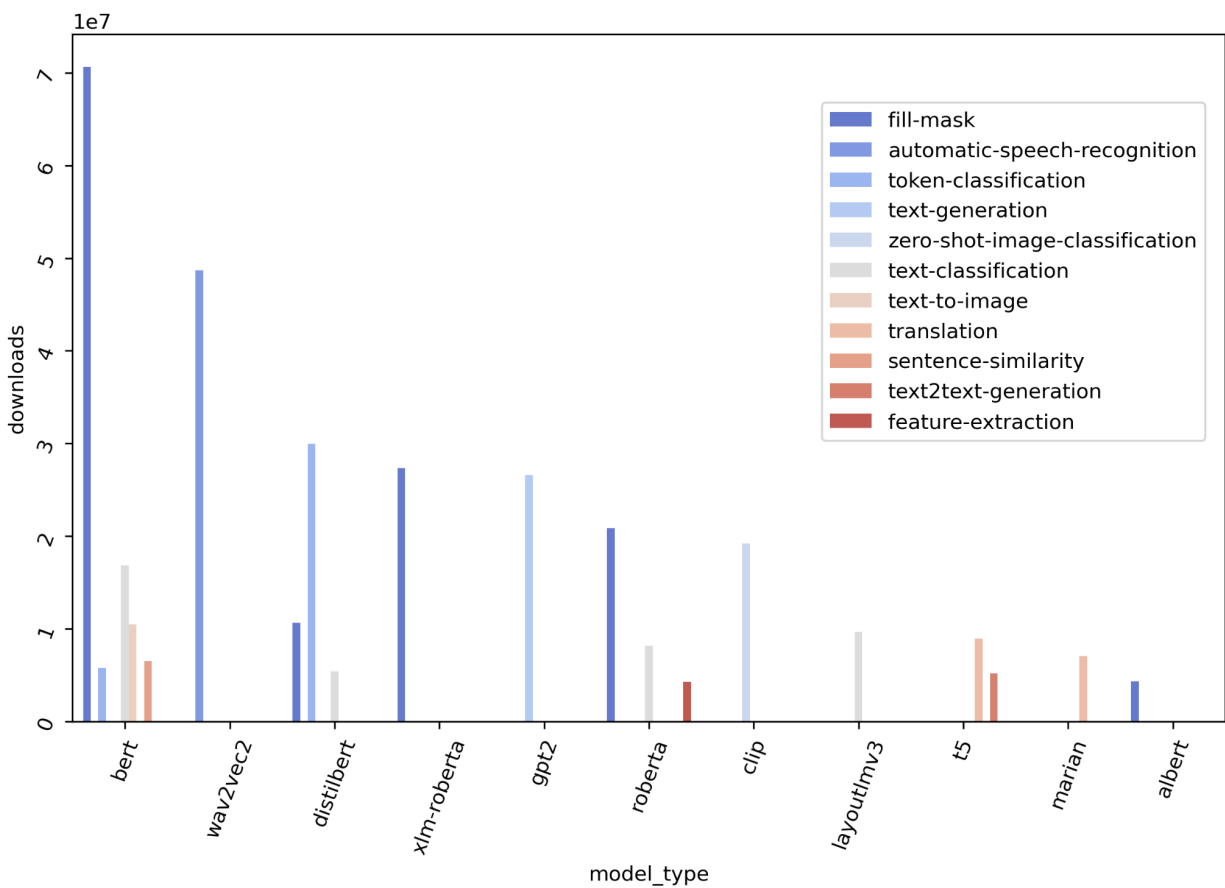


Similarly, we also count the number of models fall under different libraries and find that most models are hosted by the transformer library, followed by stable-baselines3 and diffusers.

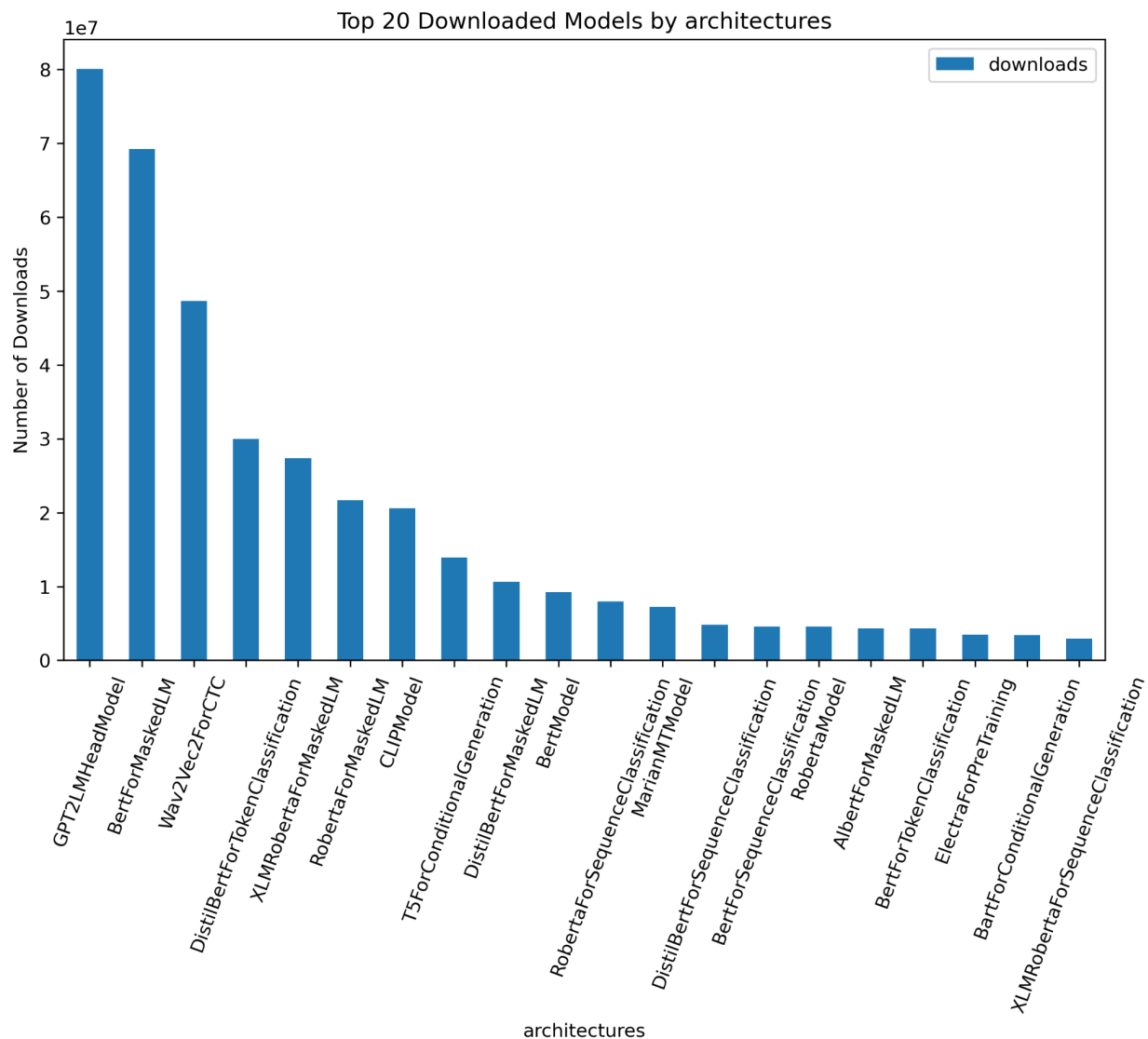


We also grouped models by model types and sum up the total number of downloads for each type of model. We found that the most downloaded model type is bert, followed by wav2vec2 and distilbert.

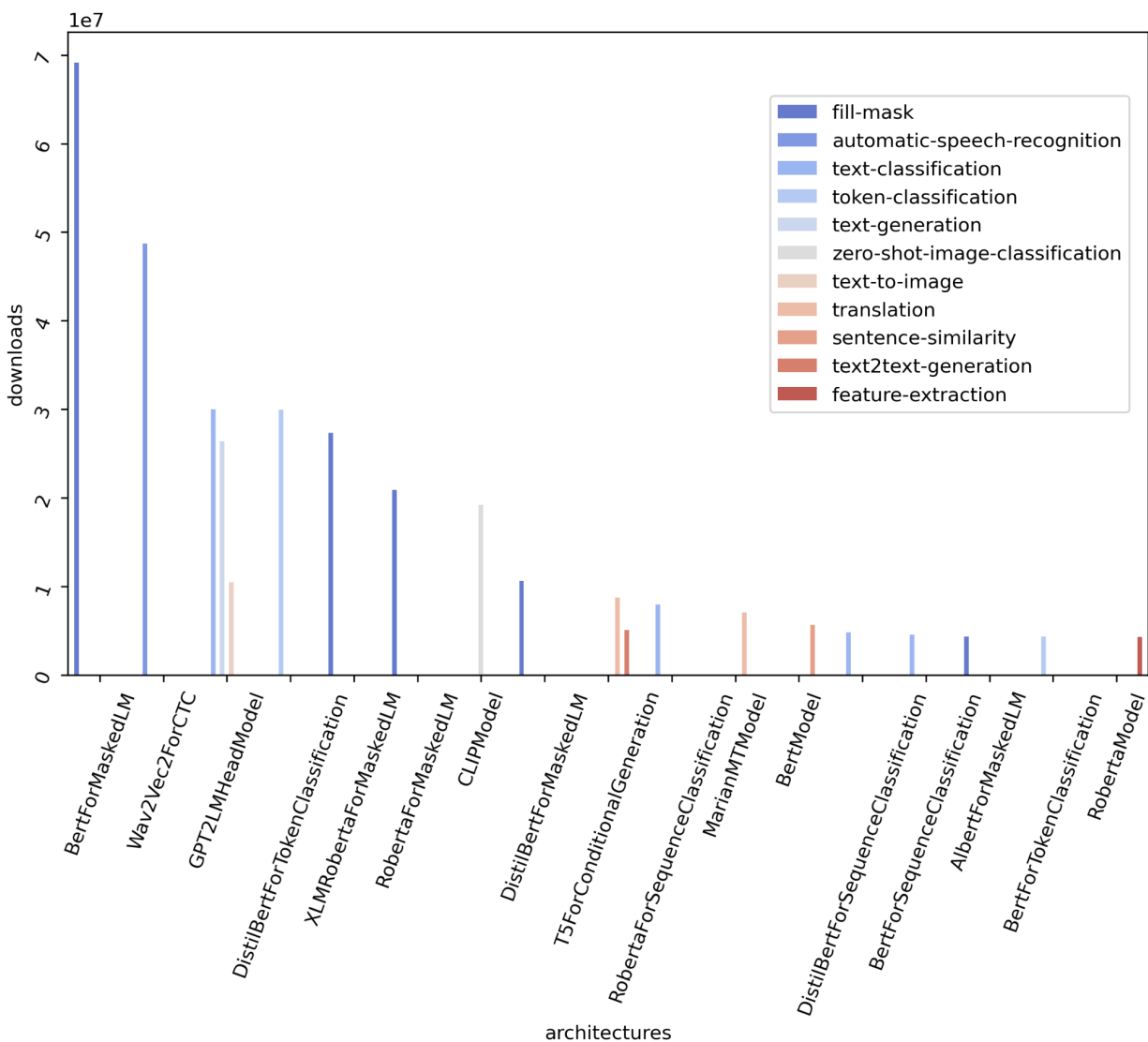




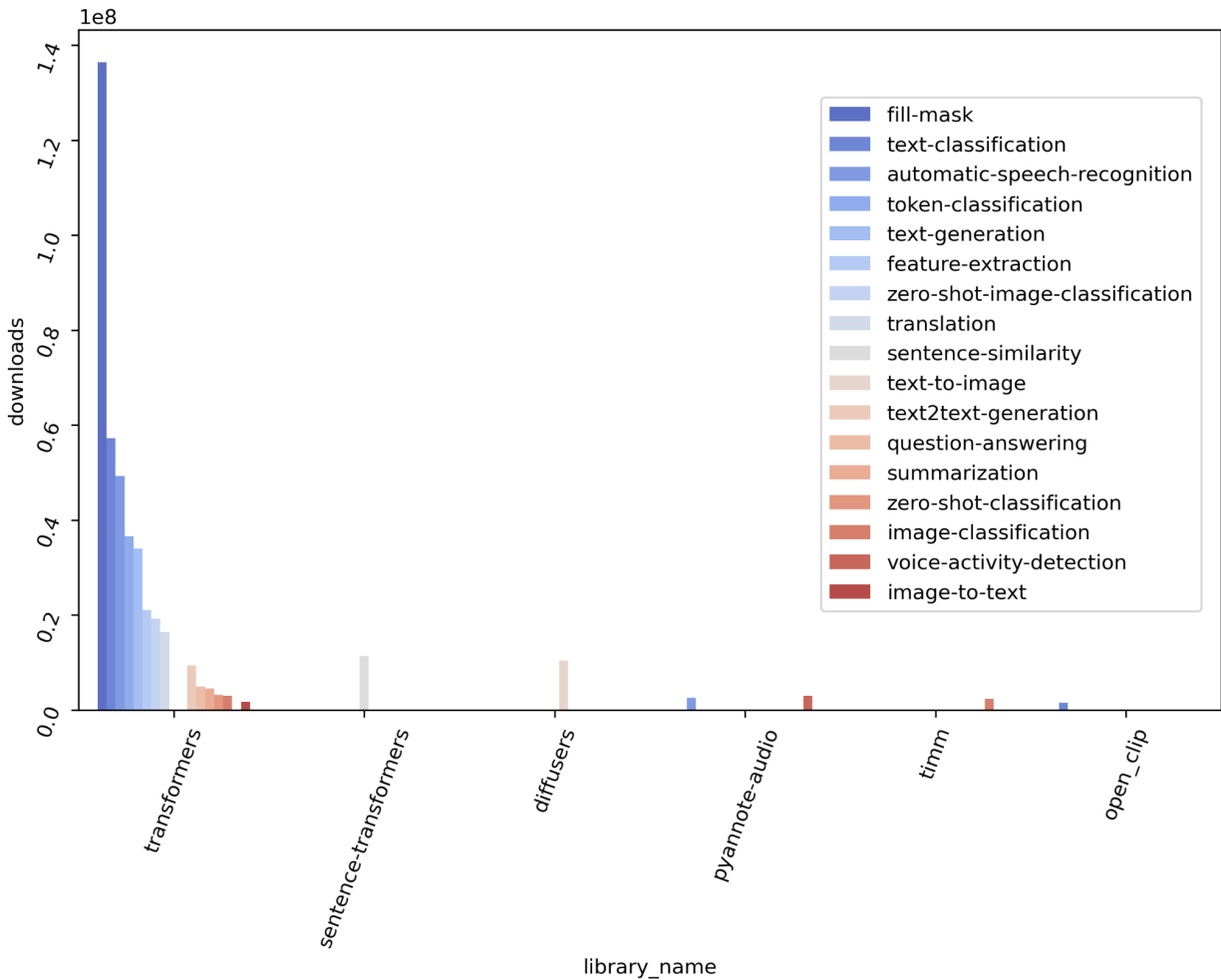
We further segment each library into different tasks. The most common task for bert, wav2vec2 and distilbert is fill-mask, automatic-speech-recognition and token-classification, respectively.



We grouped different models by their architectures and sorted by number of downloads. The top three most downloaded architectures are GPT2LMHeadModel, BertForMaskedLM and Wav2Vec2ForCTC.



We also investigate the number of downloads for different model architectures segmented by different tasks. The most downloaded model architecture is BertForMaskedLM for fill-mask, followed by Wav2Vec2ForCTC for automatic-speech-recognition.



Lastly we investigate the number of downloads for different libraries and find that the transformer is a generic library that can be used for most tasks while the rest libraries are specifically used for one or two tasks.

### C. Algorithm selection and implementation

-Discuss simple similarity algorithm eg. Cosine, Jaccard

#### Cosine Similarity

Cosine similarity is a metric commonly used to measure the similarity between two vectors in a high-dimensional space. The score ranges from -1 to 1, where a score of 1 indicates that the two vectors are identical, and a score of 0 indicates that they are completely dissimilar.

#### Jaccard Similarity

Jaccard similarity measures how much overlap there is between the sets of words used in two documents. The higher the Jaccard similarity, the more similar the two documents are.

In the context of our recommender system project, the similarity algorithms are applied to the features that have been transformed into vectors using pre-trained language model embeddings. These embeddings represent the underlying semantic meaning of the tags associated with each model. The similarity scores can then be used to recommend similar items to a user based on the similarity of the feature attributes, which in this case are the words used in the tags. The higher the cosine similarity score between two models, the more similar their semantic meaning and the more likely they are to be recommended to a user who has shown interest in one of the models.

## GNN

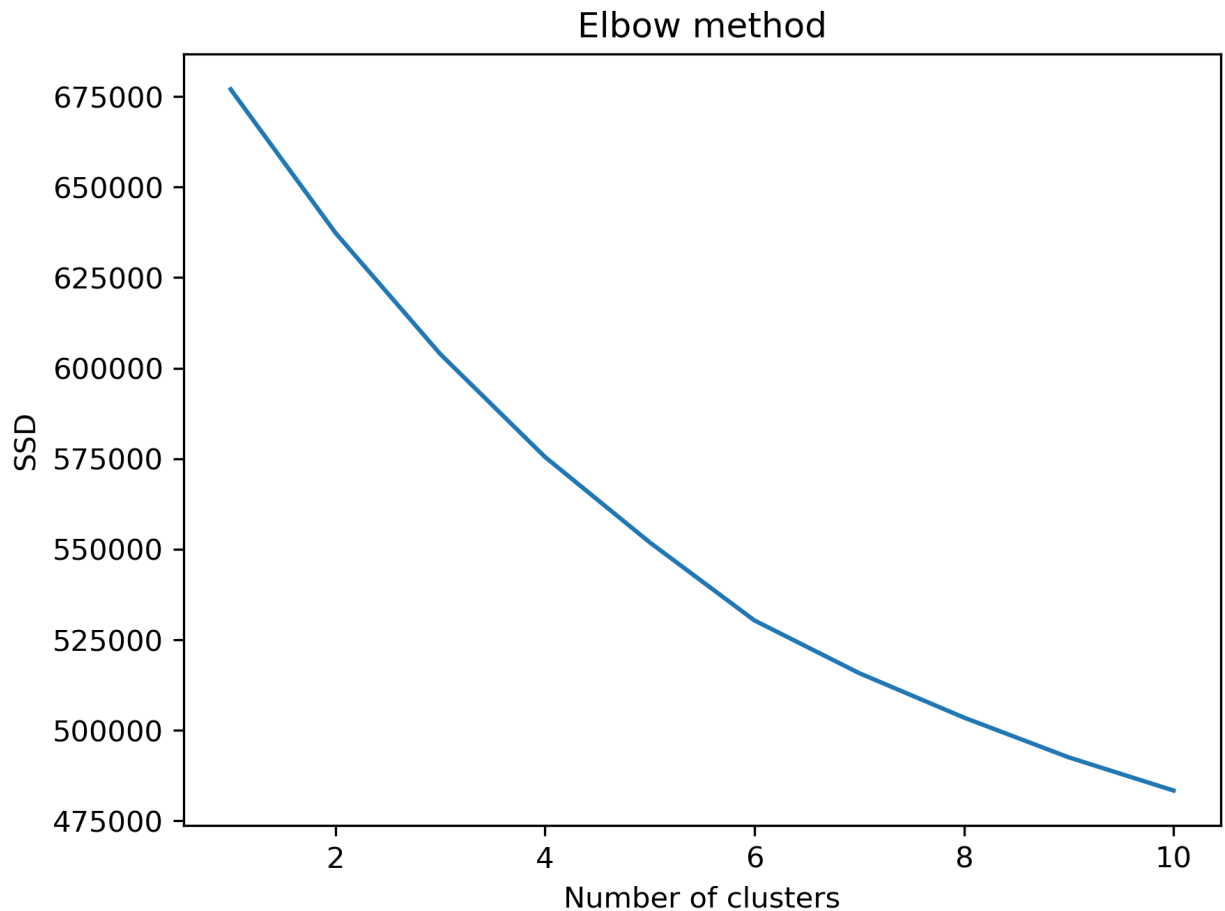
### K-means clustering with cosine similarity

K-means clustering is a popular unsupervised machine learning algorithm used to partition a dataset into K clusters, where K is a user-defined number of clusters. The goal of K-means clustering is to group similar data points together and separate dissimilar data points into different clusters. We use the K-means' predict function to assign the test point to its corresponding cluster based on its features. Once identified the cluster to which the test sample point belongs, we retrieve the sample points within the same cluster and calculate the cosine similarity between the test sample point and each of the other sample points in the cluster. After then we rank sample points in the cluster based on their similarity to the test sample point in descending order. Sample points with top n highest similar score corresponds to the n most relevant models to be recommended.

Find optimal K:

1. By elbow method

The elbow method is a way to choose the best number of clusters for k-means clustering. It works by plotting the sum of squared distances between the data points and their assigned cluster centroids as a function of the number of clusters.



Sum of squared distances (SSD) measures the distance between each data point and its centroid, squaring this distance, and summing these squares across one cluster. The "elbow" in the plot is a point where the decrease in SSD starts slows down. This indicates that adding more clusters does not result in a significant reduction in WSS and may instead lead to overfitting. The elbow method is a heuristic and may not always give a clear elbow point. In such cases, it can be helpful to consider other metrics or methods, such as the silhouette score or hierarchical clustering.

## 2. By clustering evaluation metrics

**Intrinsic Measures:** These measures do not require ground truth labels (applicable to all unsupervised learning results):

1. **Silhouette Score:** Silhouette Coefficient measures the between-cluster distance against within-cluster distance. A higher score signifies better-defined clusters. The Silhouette Coefficient of a sample measures the average distance of a sample with all other points in the next nearest cluster against all other points in its cluster. A higher ratio

signifies the cluster is far away from its nearest cluster and that the cluster is more well-defined. The Silhouette Coefficient for a set of samples takes the average Silhouette Coefficient for each sample.

2. Davies Bouldin Score: The score is defined as the average similarity measure of each cluster with its most similar cluster, where similarity is the ratio of within-cluster distances to between-cluster distances. It measures the size of clusters against the average distance between clusters. The minimum score is zero, with lower values indicating better clustering.

3. The Calinski-Harabasz Index, or Variance Ratio Criterion, measures the sum of between-cluster dispersion against the sum of within-cluster dispersion, where dispersion is the sum of distance squared. A higher score signifies better-defined clusters.

**Silhouette  
score**

**Scikit-Learn**

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \text{ if } |C_I| > 1$$

**Davies-  
Bouldin score**

**Calinski-  
Harabasz score**

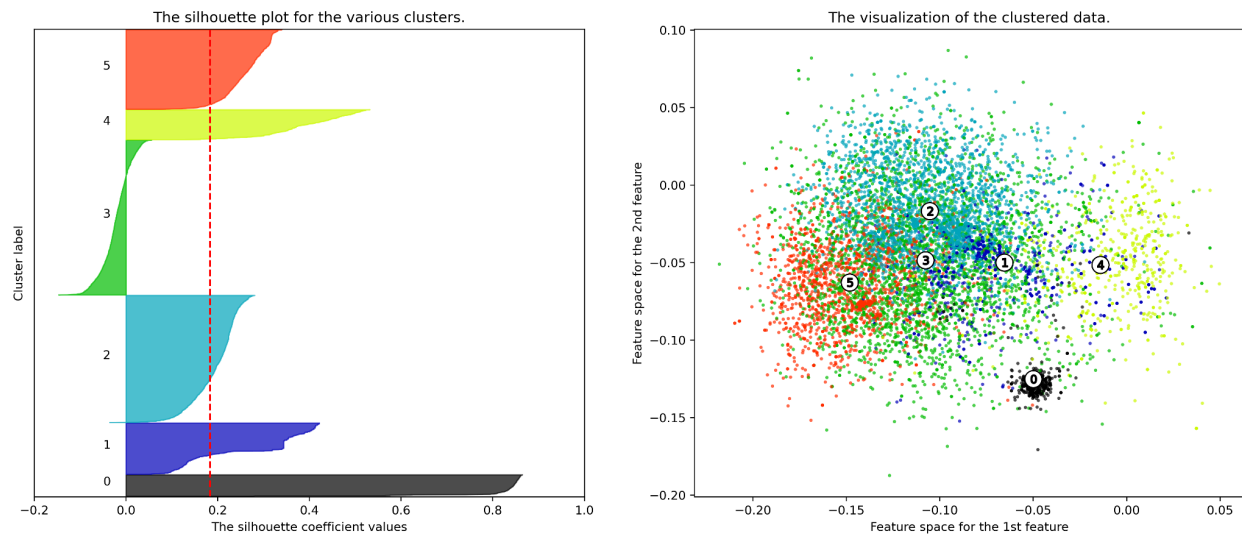
**Python**

$$DB \equiv \frac{1}{N} \sum_{i=1}^N D_i$$

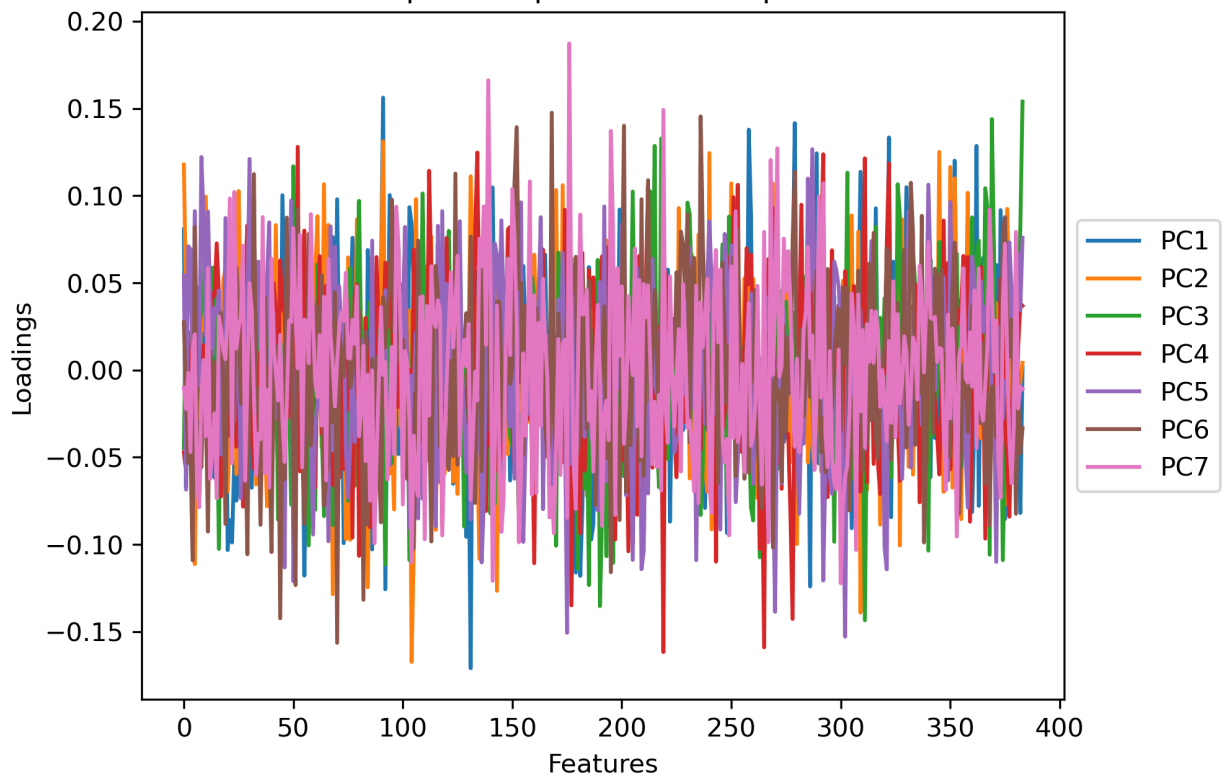
$$CH = \frac{\frac{BGSS}{K-1}}{\frac{WGSS}{N-K}} = \frac{BGSS}{WGSS} \times \frac{N-K}{K-1}$$

Silhouette Plots: measures the separation distance between clusters. Y-axis represents the cluster label and x-axis represents the Silhouette Score for each sample point. The higher the Silhouette Coefficients (the closer to +1), the further away the cluster's samples are from the neighboring clusters samples. A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters. Negative values, instead, indicate that those samples might have been assigned to the wrong cluster. Averaging the Silhouette Coefficients, we can get to a global Silhouette Score which can be used to describe the entire population's performance with a single value, as shown by the red dash line.

### Silhouette analysis for KMeans clustering on sample data with n\_clusters = 6



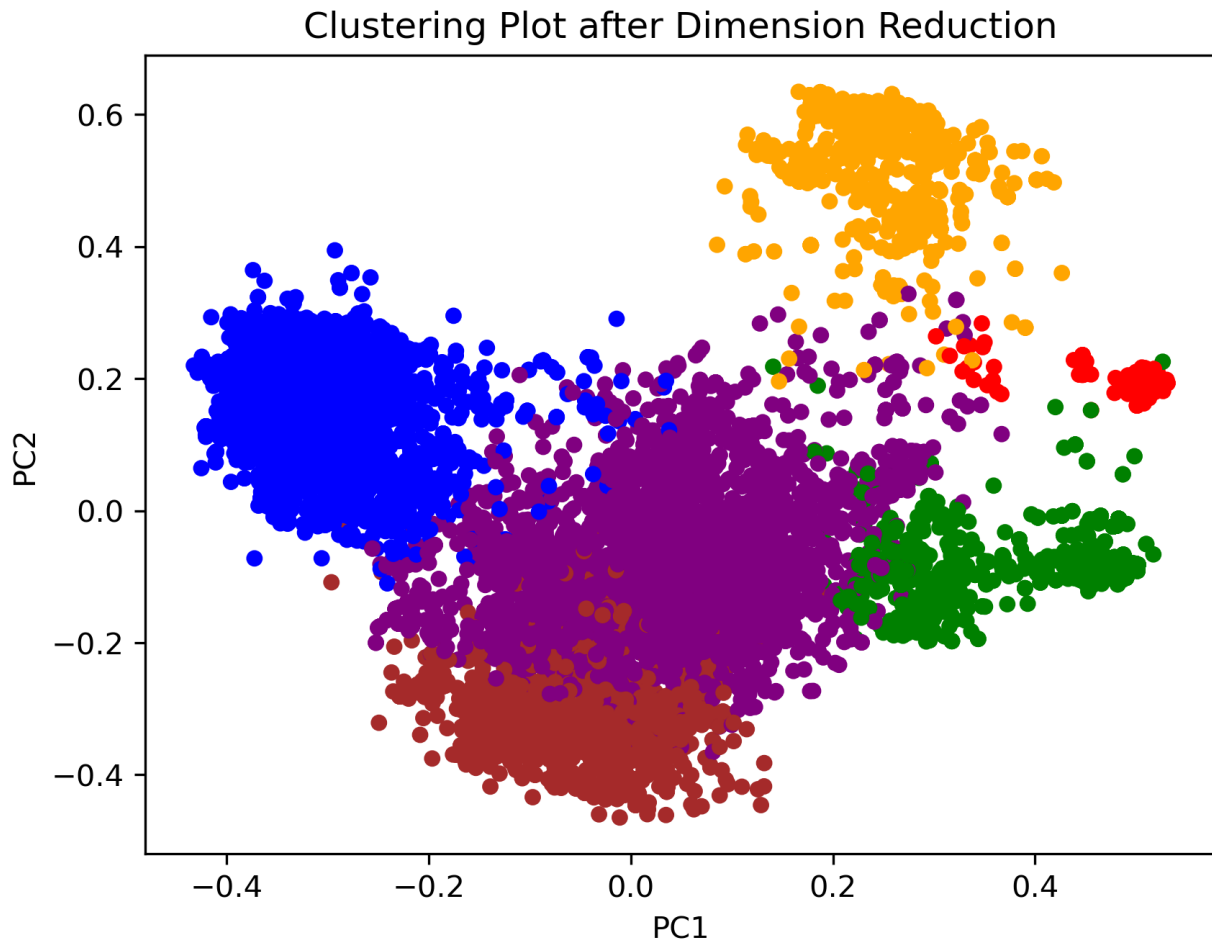
### Principle Component Decomposition



Principal components are linear combinations of the original variables that capture the most variation in the data. The first principal component (PC1) explains the largest amount of variation in the data, followed by the second principal component (PC2), and so on. The



loadings represent the weights of each feature in the corresponding principal component and can be interpreted as the contribution of each feature to the overall variation captured by the component. Features with large positive or negative loadings have a strong influence on the corresponding principal component, while features with loadings close to zero have little influence. The sign of the loading indicates the direction of the feature's influence on the component. A positive loading means that higher values of the feature are associated with higher scores on the component, while a negative loading means that higher values of the feature are associated with lower scores on the component.



## Implementation

### A. Data preprocessing and cleaning

From the EDA, we can see that the columns "tags", "datasets" contain lists. There is a need to preprocess the lists to unpack the string values within. Empty lists are also represented with square brackets [ ], which would be a problem for later steps.

The dataset also contains over 170k rows, in which more than 140k samples have less than 10 downloads. This suggests that the dataset includes model samples submitted by non-professional users who are likely to be experimenting with their code.

Thus, we have come up with the following preprocessing steps:

```
def _process_list_features(self, features: list = \
    ['tags', 'architectures', 'datasets']):
    """Process features with list values to string values and impute
    exceptions values with NaN values.

    :param features: Features with list values.
    Defaults to \['tags', 'architectures', 'datasets'].
    :type features: list, optional
    """
    for feat in features:
        self.df[feat] = self.df[feat].apply(str)
        self.df[feat] = self.df[feat].apply(
            lambda x: re.sub('[%s]' %
re.escape(string.punctuation), "", x))
        self.df[feat] = self.df[feat].apply(lambda x: x.lower())
        self.df[feat].replace('[]', np.nan, inplace=True)

    logging.info("Preprocessed features with list values")
```

```
['pytorch', 'tf', 'albert', 'fill-mask', 'en', 'dataset:bookcorpus',
'dataset:wikipedia', 'arxiv:1909.11942', 'transformers',
'license:apache-2.0', 'autotrain_compatible', 'has_space']
```

Firstly, we have to unpack features which contain lists and also handle exceptions. After dropping unimportant columns such as `"lastModified"`, the columns which contain text data will be concatenated to create a feature soup. The dataset will also be limited to only 10000 rows to decrease compute time/memory requirements.

This corpus will then be used as an input to the SentenceTransformer library for processing. The SentenceTransformer library can encode this corpus into high-dimensional vectors that capture the semantic meaning. After which, the vectors can then be used to perform similarity analysis using common text similarity algorithms such as Cosine similarity and Jaccard similarity.

## B. Implementation of the recommendation algorithm

# Inference

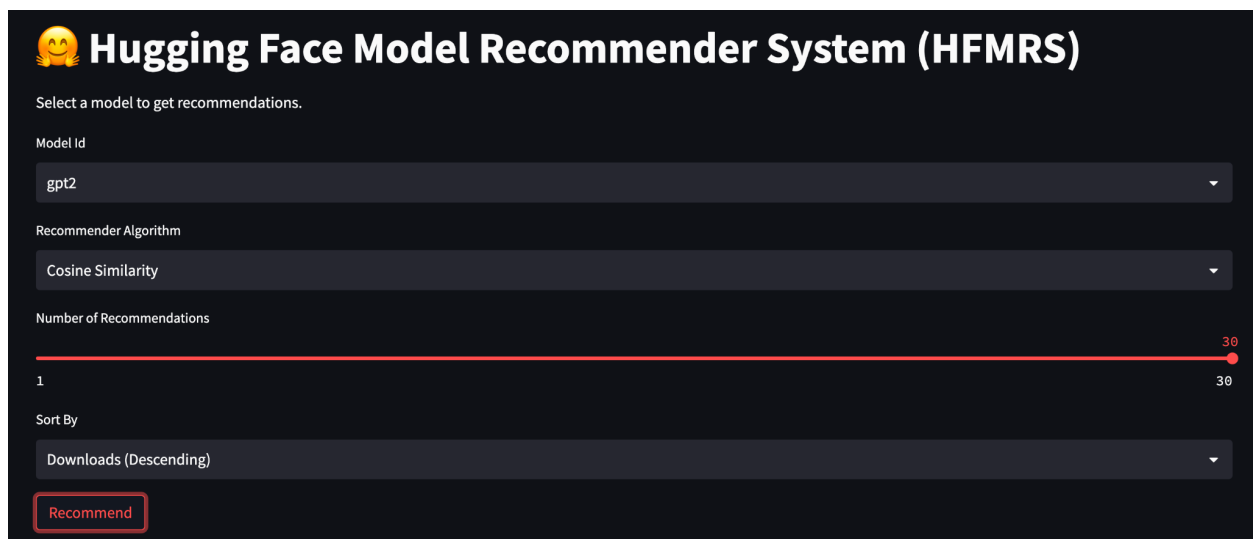
## Streamlit Application

HFMRS provides two user-friendly frontends to assist AI practitioners in quickly discovering and comparing similar pre-trained models. The first front-end is a standalone Streamlit application dockerized for easy deployment. This front-end offers a visually appealing interface that allows users to explore and compare different models for benchmarking purposes easily.

To run the Streamlit application, run the following command.

```
docker compose -f docker-compose.streamlit.yml up
```

The web application will be served at <http://localhost:8051>.



The screenshot shows the Hugging Face Model Recommender System (HFMRs) interface. It features a dark theme with a title bar at the top containing a smiley face emoji and the text "Hugging Face Model Recommender System (HFMRs)". Below the title, there is a instruction "Select a model to get recommendations." followed by several interactive elements: a "Model Id" dropdown menu with "gpt2" selected, a "Recommender Algorithm" dropdown menu with "Cosine Similarity" selected, a "Number of Recommendations" slider ranging from 1 to 30 (currently set at 30), and a "Sort By" dropdown menu with "Downloads (Descending)" selected. At the bottom left, there is a red-outlined button labeled "Recommend".

Figure X: HFMRs Streamlit App

To get recommended models, the user has to select the model Id of interest. He can customize his recommendation by selecting the recommender algorithm of choice, the number of recommendations, and sort by order.

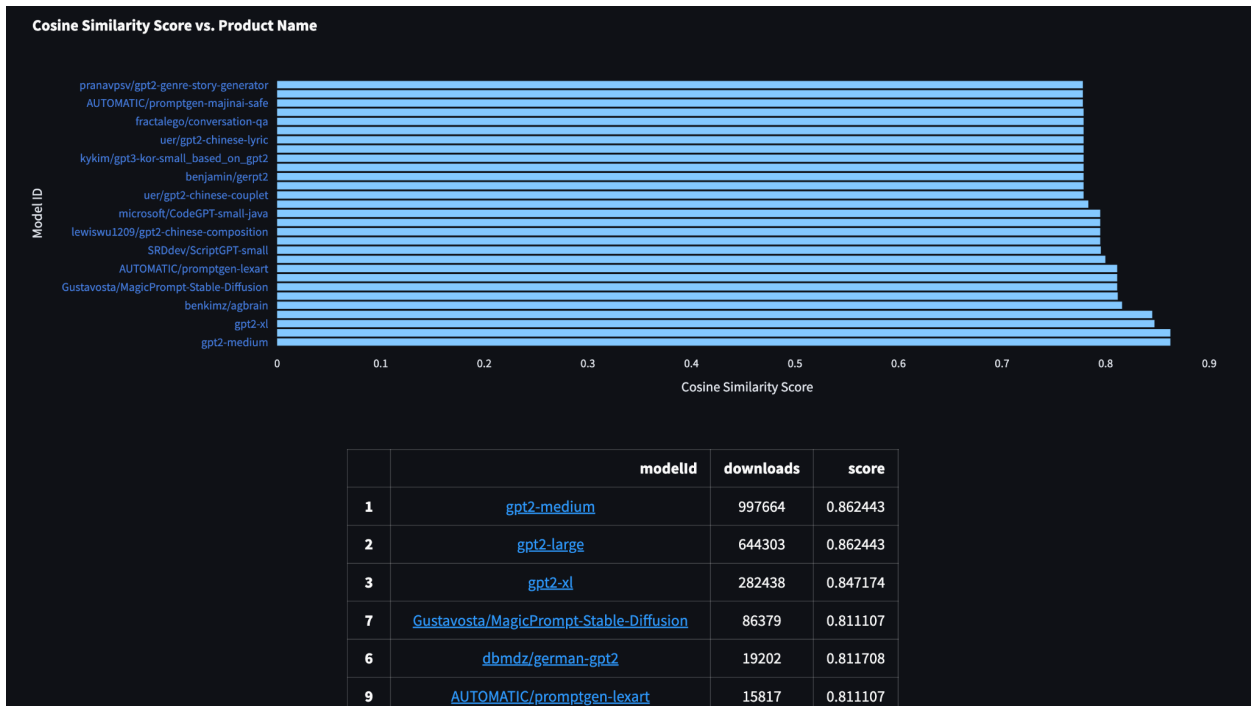


Figure X: Streamlit Inference Results

## FastAPI + Chromium Extension

The second front-end is a Chromium extension that integrates seamlessly with the Hugging Face model sidebar. This frontend is powered by a FastAPI dockerized container, which makes it lightning-fast and responsive. The chrome extension provides users with an intuitive interface that displays personalized model recommendations for the task at hand. This frontend is particularly useful for users who frequently use the Hugging Face website, as it offers them model recommendations without interrupting their workflow.

To run the FastAPI back-end, run the following command.

```
docker compose -f docker-compose.fastapi.yml up
```

The back-end API will be served at <http://localhost:8000>.

Once the user has obtained the HFMRS Chrome extension from the repository, they can easily install it into their Chrome browser by following a few simple steps. First, the user needs to access the Chrome Extensions settings and enable "Developer Mode". Next, they can select "Load unpacked" and load the "chrome-ext" folder from the repository into Chrome.

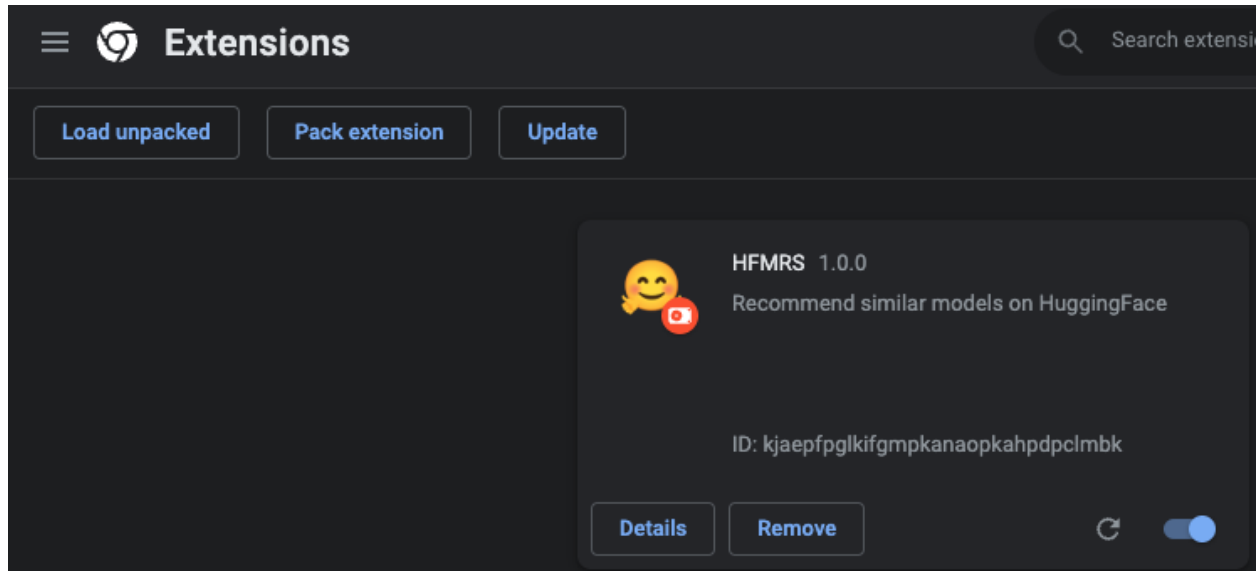


Figure X: Installing Chrome Extension Locally

The user can also customize the recommendations similar to the streamlit application by clicking "Details" and select "Extension Options"



Figure X: HFMRS extension options

Finally, the user can find similar models while browsing through the models on Hugging Face.

sentences.

More precisely, inputs are sequences of continuous text of a certain length and the targets are the same sequence, shifted one token (word or piece of word) to the right. The model uses internally a mask-mechanism to make sure the predictions for the token  $i$  only uses the inputs from 1 to  $i$  but not the future tokens.

This way, the model learns an inner representation of the English language that can then be used to extract features useful for downstream tasks. The model is best at what it was pretrained for however, which is generating texts from a prompt.

This is the **smallest** version of GPT-2, with 124M parameters.

**Related Models:** [GPT-Large](#), [GPT-Medium](#) and [GPT-XL](#)

#### Intended uses & limitations

You can use the raw model for text generation or fine-tune it to a downstream task. See the [model hub](#) to look for fine-tuned versions on a task that interests you.

#### How to use

You can use this model directly with a pipeline for text generation. Since the generation relies on some randomness, we set a seed for reproducibility:

```
>>> from transformers import pipeline, set_seed
>>> generator = pipeline('text-generation', model='gpt2')
>>> set_seed(42)
>>> generator("Hello, I'm a language model,", max_length=30, num_return_sequences=1)
```

#### Spaces using gpt2 455

Gustavosta/MagicPrompt-Stable-Diffusion    microsoft/HuggingGPT  
shi-labs/Versatile-Diffusion    microsoft/Promptist    OFA-Sys/OFA-Image\_Caption  
wangrongsheng/ChatPaper    flax-community/image-captioning  
nateraw/lavila    deepklarity/poster2plot    EleutherAI/magma  
akhaliq/CLIP\_prefix\_captioning    OFA-Sys/OFA-Visual\_Grounding  
doevent/Stable-Diffusion-prompt-generator    johko/capdec-image-captioning  
OFA-Sys/OFA-vqa    bipin/image2story    OFA-Sys/OFA-Generative\_Interface  
yizhangliu/Text-to-Image    phenomenon1981/MagicPrompt-Stable-Diffusion  
hkunlp/Binder    + 435 Spaces

#### Similar Models

Model Id	Similarity Score
<a href="#">gpt2-medium</a>	0.86244
<a href="#">gpt2-large</a>	0.86244
<a href="#">gpt2-xl</a>	0.84717
<a href="#">Gustavosta/MagicPrompt-Stable-Diffusion</a>	0.81111
<a href="#">dbmdz/german-gpt2</a>	0.81171
<a href="#">AUTOMATIC/promptgen-lexart</a>	0.81111
<a href="#">anonymous-german-nlp/german-gpt2</a>	0.78322
<a href="#">pranavpsv/gpt2-genre-story-generator</a>	0.77799
<a href="#">Gustavosta/MagicPrompt-Dalle</a>	0.81111

Figure X: Recommendations for the current model on Hugging Face

## Challenges

### Developing Recommenders without User-Model Interactions

Developing a recommender system solely based on model features can be challenging, primarily because it restricts the system's ability to provide personalized recommendations. Gaining insight into a user's preferences and behavior is difficult without user-product interactions. A recommender that solely relies on product features might overlook important aspects such as user context, and fail to capture the nuances of user preferences, which can lead to suboptimal recommendations.

### Lack of Universal Metrics and Gold Label Datasets

During our exploration of recommender techniques, we considered several evaluation metrics specific to each method. For clustering techniques such as K-Means, Within-Cluster Sum of Squares (WCSS), and silhouette scores are popular metrics for evaluation. Graph neural network-based approaches are typically evaluated using accuracy, precision, recall, F1 score, and area under the ROC curve (AUC-ROC).

Despite these different evaluation metrics, no universal metric can be used to compare all recommender techniques. Another challenge is the lack of a gold-label dataset, particularly in the absence of user-product interactions. One possible solution is to use synthetic data generation techniques to approximate the ground truth data. This approach can help overcome the challenges of creating a gold-label dataset, which can be time-consuming and resource-intensive.

## Future Work

### Explore Different Word Embeddings

In the future, one possible improvement for the recommender system could be to explore different pre-trained text embeddings to improve the accuracy of recommendations. For example, SentenceTransformers text embeddings offer a large collection of pre-trained models optimized for various tasks. By selecting and fine-tuning the best pre-trained embeddings for the specific use case of the recommender system, the recommendations could be made more accurate.

Furthermore, the quality of word embeddings can be assessed quantitatively using performance metrics like GLUE or SuperGLUE benchmarks.

### Consider Representation of Tasks

Additionally, it would be beneficial to consider the proportion of tasks in the data segmentation process to ensure a more balanced representation of tasks. Our current approach of selecting only the top 10000 models by downloads for training could lead to the underrepresentation or exclusion of models associated with more obscure tasks (i.e. robotics). By taking the proportion of tasks (pipeline\_tag) into account during training, the model could cover a wider range of tasks and better meet the needs of a broader user base.

### Model Updates

Most model metadata is static and rarely changes after publishing the model. Only the download numbers will be updated monthly. In this case, we anticipate to fetch the model metadata monthly for retraining. We can also engineer more features such as the download velocity over the past month, and increase the weights for those models with a higher monthly download rate for recommendations.

## Cloud Deployment

Our system architecture is cloud-ready, and the solution can be easily deployed to the cloud using containerization. For example, we can consider the following high-level solutions on Google Cloud Platform (GCP).

1. Use Google Cloud Platform (GCP)'s Compute Engine to host the docker images for Streamlit and FastAPI.
2. Use Cloud Scheduler and Cloud Function to run a monthly cron job to update the data and retrain the model
3. Use Cloud Storage to store the data and model files.
4. Consider MLFlow for experiment tracking and hyperparameter tuning
5. Use Cloud Monitoring to integrate monitoring data into the workflow

## Chromium Extension

The Chromium extension is natively tested in Chrome and can be installed into other Chromium-based browsers such as Microsoft Edge. However, some changes to the [Manifest file](#) may be required depending on the browser requirements.

## Conclusion

The team has dedicated significant effort towards developing a recommender system to assist AI learners in making a more informed decision when picking similar models for experimentation. Our comprehensive research has encompassed the exploration of various recommender algorithms and proof of concepts, with the primary objective of presenting model recommendations to the user. The ultimate aim of our system is to assist AI learners in identifying the appropriate models for their projects, thereby reducing inefficiencies and increasing the likelihood of project success. We firmly believe that HFMRS will provide invaluable guidance to AI learners and foster their learning journey.

## References

- [HuggingFace Documentation](#)