

7

Customizing Embeddings and Model Architectures

Introduction

Two full chapters of prompt engineering equipped us with the knowledge of how to effectively interact with (prompt) LLMs, acknowledging their immense potential as well as their limitations and biases. We have also fine-tuned models, both open and closed source, to expand on an LLM's pre-training to better solve our own specific tasks. We have even seen a full case study of how semantic search and embedding spaces can help us retrieve relevant information from a dataset with speed and ease.

To further broaden our horizons, we will utilize lessons learned from earlier chapters and dive into the world of fine-tuning embedding models and customizing pre-trained LLM architectures to unlock even greater potential in our LLM implementations. By refining the very foundations of these models, we can cater to specific business use-cases and foster improved performance.

Foundation models, while impressive on their own, can be adapted and optimized to suit a variety of tasks through minor to major tweaks in their architectures. This customization enables us to address unique challenges and tailor LLMs to specific business requirements. The underlying embeddings form the basis for these customizations, as they are responsible for capturing the semantic relationships between data points and can significantly impact the success of various tasks.

Recalling our semantic search example, we identified that many off-the-shelf embedding models—both closed-source models from OpenAI and open-source models—were designed to preserve semantic similarity. In this chapter, we will expand upon this concept, exploring techniques to train autoencoding LLMs (the ones that excel at reading over writing) to effectively capture other business use-cases in embedding spaces. By doing so, we will uncover the potential of customizing embeddings and model architectures to create even more powerful and versatile LLM applications.

Case Study: Building a Recommendation System

Most of this chapter will explore the role of embeddings and LLM architectures in designing a recommendation engine while using a real-world dataset as our case study. We will be using embedding models from OpenAI and open-source ones as well. All of the embedding models we will look at are powered by LLMs. (Although not all embedding architectures in the world are powered by LLMs, they are in this chapter.) Our objective in this chapter is to highlight the importance of customizing embeddings and model architectures in achieving better performance and results tailored to specific use-cases.

Setting Up the Problem and the Data

To demonstrate the power of customized embeddings, we will be using the MyAnimeList 2020 dataset, which can be accessed on Kaggle. This dataset contains information about anime titles, ratings (from 1 to 10), and user preferences, offering a rich source of data to build a recommendation engine. [Figure 7.1](#) shows a snippet of the dataset on the Kaggle page.

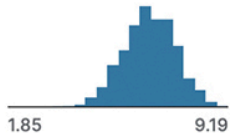
▲ Name	# Score	▲ Genres	▲ synopsis
full name of the anime.	average score of the anime given from all users in MyAniList database. (e.g. 8.78)	comma separated list of genres for this anime.	string with the synopsis of the anime.
16210 unique values	 1.85 9.19	Music 5% Comedy 4% Other (14756) 91%	No synopsis inform... No synopsis has be... Other (15470) !
Cowboy Bebop	8.78	Action, Adventure, Comedy, Drama, Sci-Fi, Space	In the year 2071, humanity has colonized several the planets and moons of the solar system leavin...
Cowboy Bebop: Tengoku no Tobira	8.39	Action, Drama, Mystery, Sci-Fi, Space	other day, another bounty-such is the life of the often unlucky crew of the Bebop. However, the rou...
Trigun	8.24	Action, Sci-Fi,	Vash the Stampede

Figure 7.1 The MyAnimeList database is one of the largest datasets we have worked with to date. Found on Kaggle, it has tens of millions of rows of ratings and thousands of anime titles, including dense text features describing each anime title. Source: Anime Recommendation Database 2020. Retrieved from <https://www.kaggle.com/datasets/hernan4444/anime-recommendation-database-2020>

To ensure a fair evaluation of our recommendation engine, we will divide the dataset into separate training, validation, and testing sets. As was the case in the previous chapter, all results we will see were run on the hold-out testing set. This process allows us to train our model on one portion of the data and evaluate its performance on a separate, unseen portion, thereby providing an unbiased assessment of its effectiveness. **Listing 7.1** shows a snippet of our code to load the anime titles and split them into an initial train and test split. The train split will be further split into training and validation sets.

Listing 7.1 Loading and splitting our anime data

[Click here to view code image](#)

```
# Load the anime titles with genres, synopsis, producers, etc.
# There are 16,206 titles
pre_merged_anime = pd.read_csv('../data/anime/pre_merged_anime.csv')

# Load the ratings given by users who have **completed** an anime
# There are 57,633,278 ratings!
rating_complete = pd.read_csv('../data/anime/rating_complete.csv')

import numpy as np

# Split the ratings into a 90/10 train/test split
rating_complete_train, rating_complete_test = \
    np.split(rating_complete.sample(frac=1, random_state=42),
            [int(.9*len(rating_complete))])
```

With our data loaded up and split, let's take some time to better define what we are trying to solve.

Defining the Problem of Recommendation

Developing an effective recommendation system is, to put it mildly, a complex task. Human behavior and preferences can be intricate and difficult to predict (the understatement of the millennium). The challenge lies in understanding and predicting what users will find appealing or interesting, which is influenced by a multitude of factors.

Recommendation systems need to take into account both user features and item features to generate personalized suggestions. User features can

include demographic information such as age, browsing history, and past item interactions (which will be the focus of our work in this chapter), whereas item features can encompass characteristics such as genre, price, and popularity. However, these factors alone may not paint the complete picture, as human mood and context also play a significant role in shaping preferences. For instance, a user's interest in a particular item might change depending on their current emotional state or the time of day.

Striking the right balance between exploration and pattern exploitation is also important in recommendation systems. **Pattern exploitation** refers to a system recommending items that it is confident the user will like based on their past preferences or are just simply similar to things they have interacted with before. In contrast, we can define **exploration** to mean suggesting items that the user might not have considered before, especially if the recommendation is not exactly similar to what they have liked in the past. Striking this balance ensures that users continue to discover new content while still receiving recommendations that align with their interests. We will consider both of these factors.

Defining the problem of recommendation is a multifaceted challenge that requires considering various factors, such as user and item features, human mood, the number of recommendations to optimize, and the balance between exploration and exploitation. Given all of this, let's dive in!

Content Versus Collaborative Recommendations

Recommendation engines can be broadly categorized into two main approaches: content-based and collaborative filtering. **Content-based recommendations** focus on the attributes of the items being recommended, utilizing item features to suggest similar content to users based on their past interactions. In contrast, **collaborative filtering** capitalizes on the preferences and behavior of users, generating recommendations by identifying patterns among users with similar interests or tastes.

On the one hand, in content-based recommendations, the system extracts relevant features from items, such as genre, keywords, or themes, to build a profile for each user. This profile helps the system understand the user's preferences and suggest items with similar characteristics. For instance, if a user has previously enjoyed action-packed anime titles, the content-based recommendation engine would suggest other anime series with similar action elements.

On the other hand, collaborative filtering can be further divided into user-based and item-based approaches. User-based collaborative filtering finds users with similar preferences and recommends items that those

users have liked or interacted with. Item-based collaborative filtering focuses on finding items that are similar to those the user has previously liked, based on the interactions of other users. In both cases, the underlying principle is to leverage the wisdom of the crowd to make personalized recommendations.

In our case study, we will fine-tune a bi-encoder (like the one we saw in [Chapter 2](#)) to generate embeddings for anime features. Our goal is to minimize the cosine similarity loss in such a way that the similarity between embeddings reflects how common it is for users to like both animes.

In fine-tuning a bi-encoder, our goal is to create a recommendation system that can effectively identify similar anime titles based on the preferences of promoters and *not* just because they are semantically similar.

[Figure 7.2](#) shows what this approach might look like. The resulting embeddings will enable our model to make recommendations that are more likely to align with the tastes of users who are enthusiastic about the content.

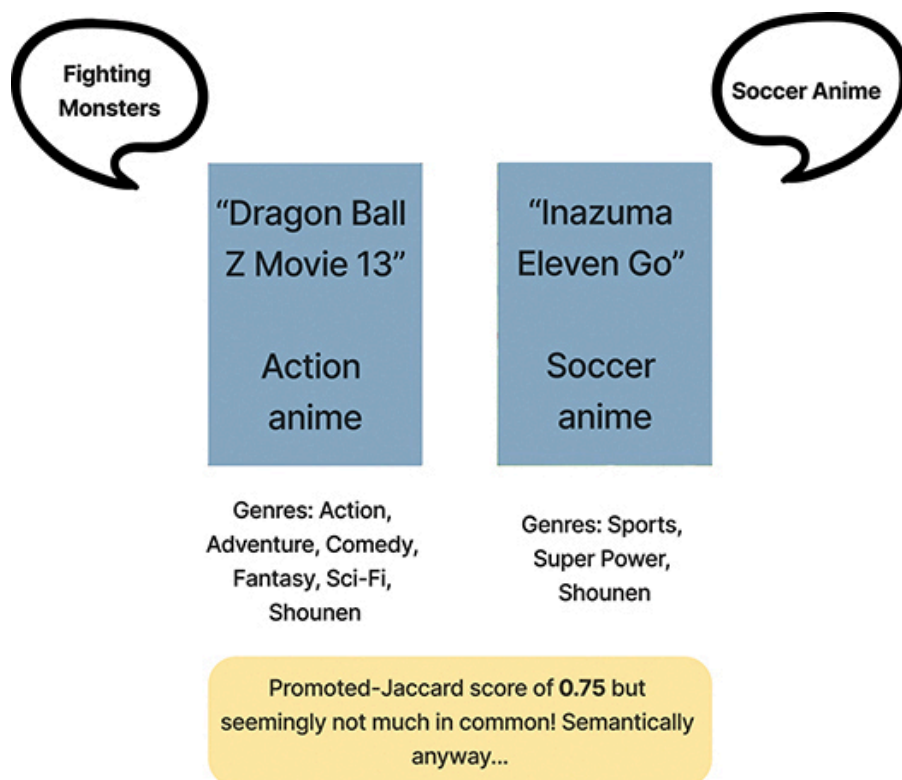


Figure 7.2 Embedders are generally pre-trained to place pieces of embedded data near each other if they are semantically similar. In our case, we want an embedder that places pieces of embedded data near each other if they are similar in terms of **user preferences**.

In terms of recommendation techniques, our approach combines elements of both content-based and collaborative recommendations. We leverage content-based aspects by using the features of each anime as input to the bi-encoder. At the same time, we incorporate collaborative filtering by considering the Jaccard score of promoters, which is based on the preferences and behavior of users. This hybrid approach allows us to take advantage of the strengths of both techniques to create a more effective recommendation system.

Explaining how we will construct this embedder, and how it will combine collaborative filtering and semantic similarity, might be helpful for envisioning the solution. In essence, we're basing this model on the collaborative filtering as a label.

To summarize, our plan involves four steps:

1. Define/construct a series of text embedding models, either using them as is or fine-tuning them on user-preference data.
2. Define a hybrid approach of collaborative filtering (using the Jaccard score to define user/anime similarities) and content filtering (semantic similarity of anime titles by way of descriptions or other characteristics) that will influence our user-preference data structure as well as how we score recommendations given to us by the pipeline.
3. Fine-tune open-source LLMs on a training set of user-preference data.
4. Run our system on a testing set of user-preference data to decide which embedder was responsible for the best anime title recommendations.

A 10,000-Foot View of Our Recommendation System

Our recommendation process will generate personalized anime recommendations for a given user based on their past ratings. Here's an explanation of the steps in our recommendation engine:

1. **Input:** The input for the recommendation engine is a user ID and an integer k (example 3). This k value will be used to query titles in relation to an anchor title.
2. **Identify highly rated animes:** We will use the Net Promoter Score (NPS) to measure likes. Given a score from 1 to 10, we will define 1–6 scores as “detractors,” scores of 7s and 8s as “passives,” and scores of 9s and 10 as “promoters” of the title. For each anime title that the user has rated as a 9 or 10 (a promoting score on the NPS scale—we will call them anchor titles), identify k other relevant animes by finding nearest matches in the anime's embedding space. From these, we consider both how often an anime was recommended and how high the resulting cosine score was in the embedding space, and take the top k re-

sults for the user. [Figure 7.3](#) outlines this process. The pseudocode would look like this:

[Click here to view code image](#)

```
given: user, k=3
promoted_animes = all anime titles that the user gave a score of 9 or a 10

relevant_animes = []
for each promoted_anime in promoted_animes:
    add k animes to relevant_animes with the highest cosine similarity to promoted
    anime along with the cosine score

# Relevant_animes should now have k * (however many animes were in promoted
# animes)

# Calculate a weighted score of each unique relevant anime given how many t
# it appears in the list and its similarity to promoted animes

final_relevant_animes = the top k animes with the highest weighted cosine/
occurrence score
```

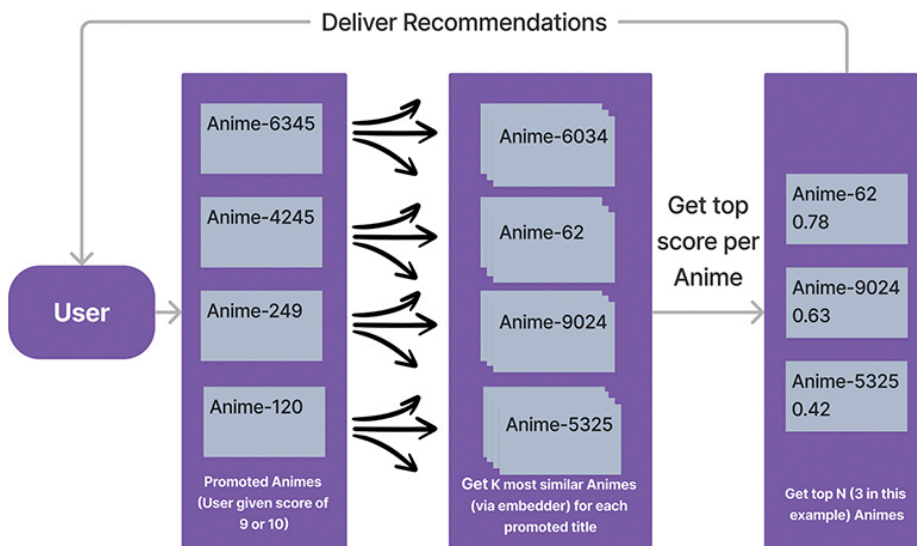


Figure 7.3 Step 2 takes in the user and finds k animes **for each** user-promoted (gave a score of 9 or 10) anime. For example, if the user promoted 4 animes (6345, 4245, 249, and 120) and we set k = 3, the system will first retrieve 12 semantically similar animes (3 per promoted animes with duplicates allowed) and then de-duplicate any animes that came up multiple times by weighing that anime slightly more than the original cosine scores. We then take the top k unique recommended anime titles considering both cosine scores for promoted animes and how often they occurred in the original list of 12.

GitHub has the full code to run this step—with examples, too. For example, given k = 3 and user ID 205282, step 2 would result in the fol-

lowing dictionary, where each key represents a different embedding model used and the values are anime title IDs and corresponding cosine similarity scores to promoted titles the user liked:

[Click here to view code image](#)

```
final_relevant_animes = {  
    'text-embedding-ada-002': { '6351': 0.921, '1723': 0.908, '2167': 0.905 },  
    'paraphrase-distilroberta-base-v1': { '17835': 0.594, '33970': 0.589, '172  
0.586 }  
}
```

3. Score relevant animes: For each of the relevant animes identified in step 2, if the anime is not present in the testing set for that user, ignore it. If we have a user rating for the anime in the testing set, we assign a score to the recommended anime given the NPS-inspired rules:

1. If the rating in the testing set for the user and the recommended anime was 9 or 10, the anime is considered a “promoter” and the system receives +1 points.
2. If the rating is 7 or 8, the anime is considered “passive” and receives 0 points.
3. If the rating is between 1 and 6, the anime is considered a “detractor” and receives −1 point.

The final output of this recommendation engine is a ranked list of the top N (depending on how many we wish to show the user) animes that are most likely to be enjoyed by the user and a score of how well the system did given a testing ground truth set. [Figure 7.4](#) shows this entire process at a high level.

Finding Top K relevant recommendations given a User ID

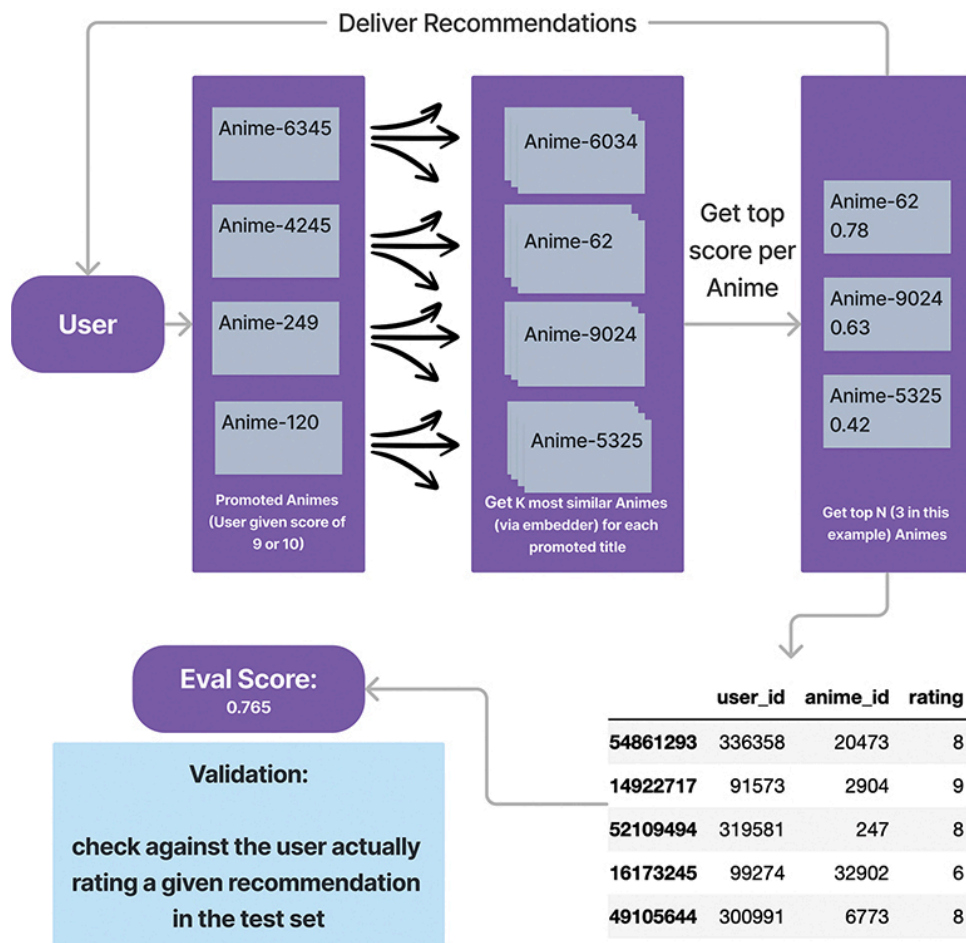


Figure 7.4 The overall recommendation process involves using an embedder to retrieve similar animes from a user's already promoted titles. It then assigns a score to the recommendations given if they were present in the testing set of ratings.

Generating a Custom Description Field to Compare Items

To compare different anime titles and generate recommendations more effectively, we will create our own custom generated description field that incorporates several relevant features from the dataset (shown in [Figure 7.5](#)). This approach offers several advantages and enables us to capture a more comprehensive context of each anime title, resulting in a richer and more nuanced representation of the content.

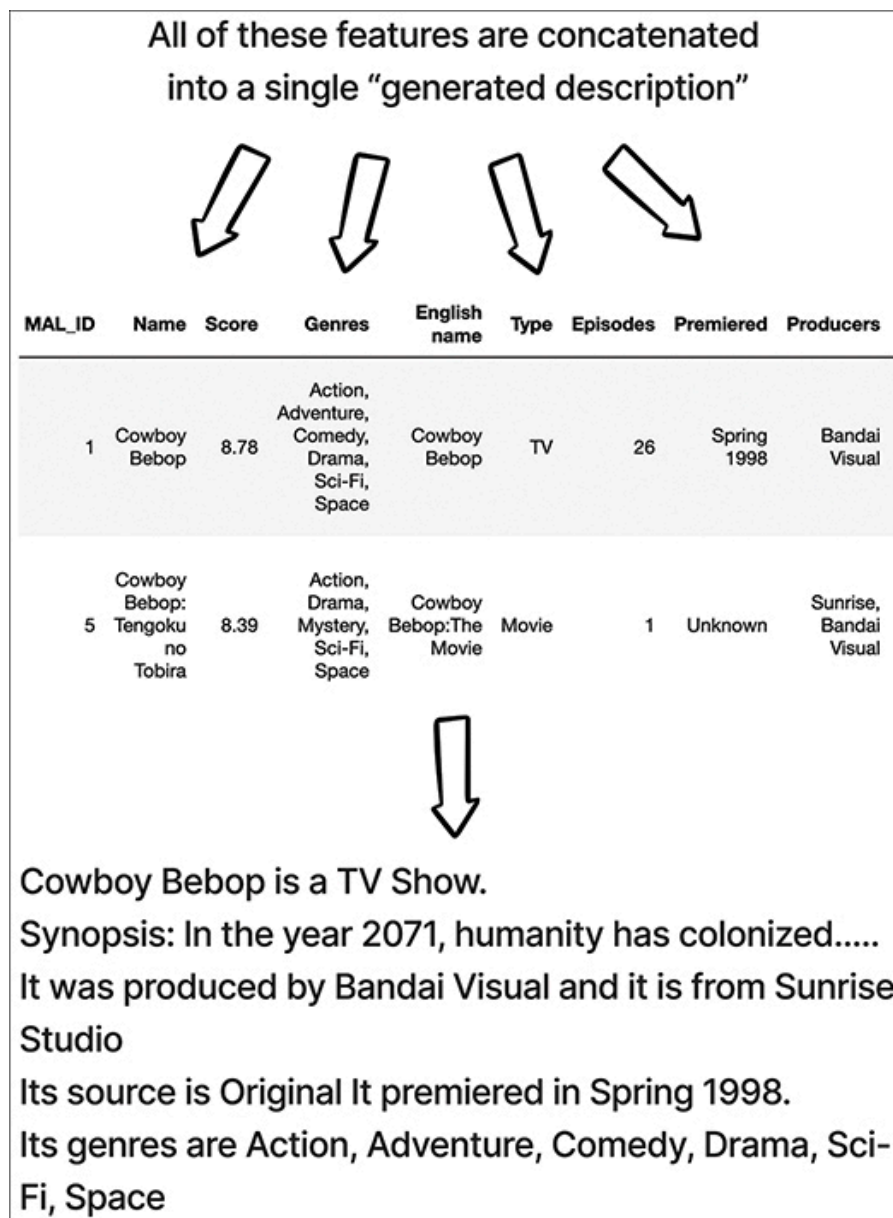


Figure 7.5 Our custom-generated (see [Listing 7.2](#) for how to generate it) description of each anime combines many raw features, including the title, genre list, synopsis, producers, and more. This approach can be contrary to how many developers think because instead of generating a structured, tabular dataset, we are deliberately creating natural text representation of our anime titles, which we will let our LLM-based embedders capture in a vector (tabular) form.

By combining multiple features, such as plot summaries, character descriptions, and genres, we can create a multidimensional representation of each anime title that allows our model to consider a broader range of information when comparing titles and identifying similarities, leading to more accurate and meaningful recommendations. Incorporating various features from the dataset into a single description field can also aid in

overcoming potential limitations in the dataset, such as missing or incomplete data. By leveraging the collective strength of multiple features, we ensure that our model has access to a more robust and diverse set of information and mitigates the effect of individual titles missing pieces of information.

In addition, using a custom-generated description field enables our model to adapt to different user preferences more effectively. Some users may prioritize plot elements, whereas others may be more interested in certain genres or media (TV series versus movies). By capturing a wide range of features in our description field, we can cater to a diverse set of user preferences and deliver personalized recommendations that align with users' individual tastes.

Overall, this approach of creating our own custom description field from several individual fields ultimately should result in a recommendation engine that delivers more accurate and relevant content suggestions.

Listing 7.2 provides a snippet of the code used to generate these descriptions.

Listing 7.2 **Generating custom descriptions from multiple anime fields**

[Click here to view code image](#)

```
def clean_text(text):
    # Remove nonprintable characters
    text = ''.join(filter(lambda x: x in string.printable, text))
    # Replace multiple whitespace characters with a single space
    text = re.sub(r'\s{2,}', ' ', text).strip()
    return text.strip()

def get_anime_description(anime_row):
    """
    Generates a custom description for an anime title based on various features f
    input data.

    :param anime_row: A row from the MyAnimeList dataset containing relevant anim
    information.
    :return: A formatted string containing a custom description of the anime.
    """

    ...
    description = (
        f"{anime_row['Name']} is a {anime_type}.\n"
    ... # Note that I omitted over a dozen other rows here for brevity
        f"Its genres are {anime_row['Genres']}\n"
    )
    return clean_text(description)
```

```
# Create a new column in our merged anime dataframe for our new descriptions
pre_merged_anime['generated_description'] = pre_merged_anime.apply(get_anime_
description, axis=1)
```

Setting a Baseline with Foundation Embedders

Before customizing our embeddings, we will establish a baseline performance using two foundation embedders: OpenAI's powerful Ada-002 and the embedder-3 family of embedders, as well as a small open-source bi-encoder based on a distilled RoBERTa model. These pre-trained models offer a starting point for comparison, helping us to quantify the improvements achieved through customization. We will start with these two models and eventually work our way up to comparing four different embedders: one closed-source embedder and three open-source embedders.

Preparing Our Fine-Tuning Data

As part of our quest to create a robust recommendation engine, we will fine-tune open-source embedders using the Sentence Transformers library. We will begin by calculating the Jaccard similarity between promoted animes from the training set.

Jaccard similarity is a simple method to measure the similarity between two sets of data based on the number of elements they share. It is calculated by dividing the number of elements that both groups have in common by the total number of distinct elements in both groups combined.

Let's say we have two anime shows, Anime A and Anime B. Suppose we have the following people who like these shows:

- People who like Anime A: Alice, Bob, Carol, David
- People who like Anime B: Bob, Carol, Ethan, Frank

To calculate the Jaccard similarity, we first find the people who like both Anime A and Anime B. In this case, it's Bob and Carol.

Next, we find the total number of distinct people who like either Anime A or Anime B. Here, we have Alice, Bob, Carol, David, Ethan, and Frank.

Now, we can calculate the Jaccard similarity by dividing the number of common elements (2, as Bob and Carol like both shows) by the total number of distinct elements (6, as there are 6 unique people in total):

Jaccard similarity (Anime A, Anime B) = $2/6 = 1/3 \approx 0.33$

So, the Jaccard similarity between Anime A and Anime B, based on the people who like them, is about 0.33 or 33%. In other words, 33% of the distinct people who like either show have similar tastes in anime, as they enjoy both Anime A and Anime B. **Figure 7.6** shows another example.

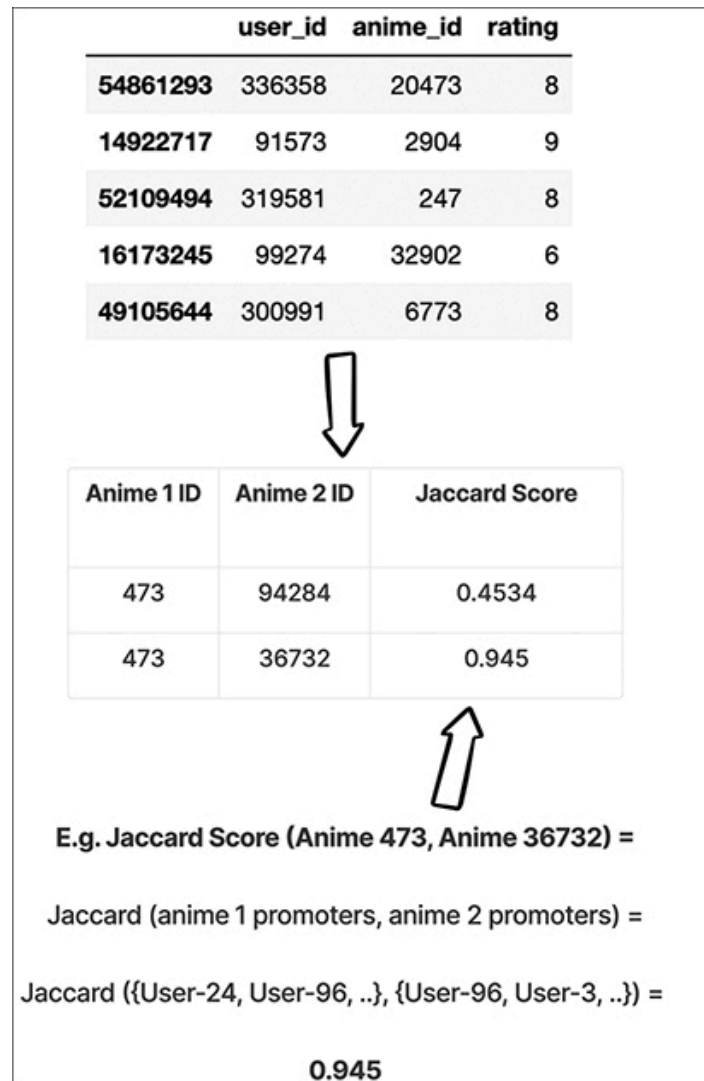
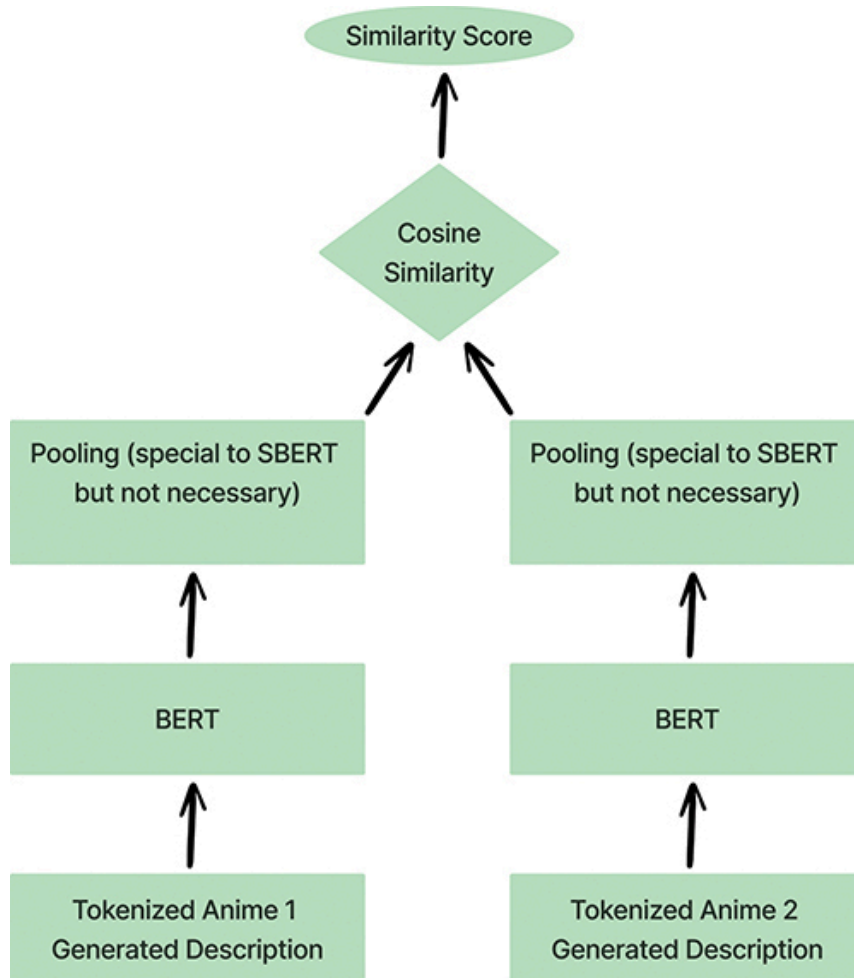


Figure 7.6 To convert our raw ratings into pairs of animes with associated scores, we will consider every pair of anime titles and compute the Jaccard similarity score between promoting users.

We will apply this logic to calculate the Jaccard similarity for every pair of animes using a training set of the ratings DataFrame. We will keep only scores above a certain threshold as “positive examples” (label of 1); the rest will be considered “negative” (label of 0).

Important note: We are free to assign any anime pairs a label between -1 and 1 —but I’m using only 0 and 1 here because I’m just using *promoting* scores to create my data. In this case, it’s not fair to say that if the Jaccard score between animes is low, then the users totally disagree on the anime. That’s not necessarily true! If I expanded this case study, I would want to explicitly label animes as -1 if and only if users were genuinely rating them in an opposite manner (i.e., if most users who promote one anime are detractors of the other).

Once we have Jaccard scores for the anime IDs, we need to convert them into tuples of three elements—two anime descriptions plus a label (in our case, the Jaccard score will be converted to either 0 or 1). Then we can update our open-source embedders and experiment with different token windows (shown in [Figure 7.7](#)).



Anime 1 Desc	Anime 2 Desc	Label
"Cowboy Bebop..."	"One Piece.."	1
"Haiyku!! ..."	"Naruto ..."	0

Anime 1 ID	Anime 2 ID	Jaccard Score
473	94284	0.4534
473	36732	0.1

Figure 7.7 Jaccard scores are converted into cosine labels and then fed into our bi-encoder, enabling the bi-encoder to attempt to learn patterns between the generated anime descriptions and how users co-like the titles.

Once we have Jaccard similarities between anime pairs, we can convert these scores to labels for our bi-encoder by applying a simple rule. In our case, if the score is greater than 0.3, then we label the pair as “positive” (label 1), and if the label is less than 0.1, we label it as “negative” (label 0).

Adjusting Model Architectures

When working with open-source embedders, we have much more flexibility to change things around if necessary. For example, the open-source model we’ll use in this case study was pre-trained with the ability to take in only 128 tokens at a time and truncate anything longer than that.

Figure 7.8 shows the histogram of the token lengths for our generated anime descriptions. Clearly, we have many descriptions that are more than 128 tokens—some in the 600-token range!

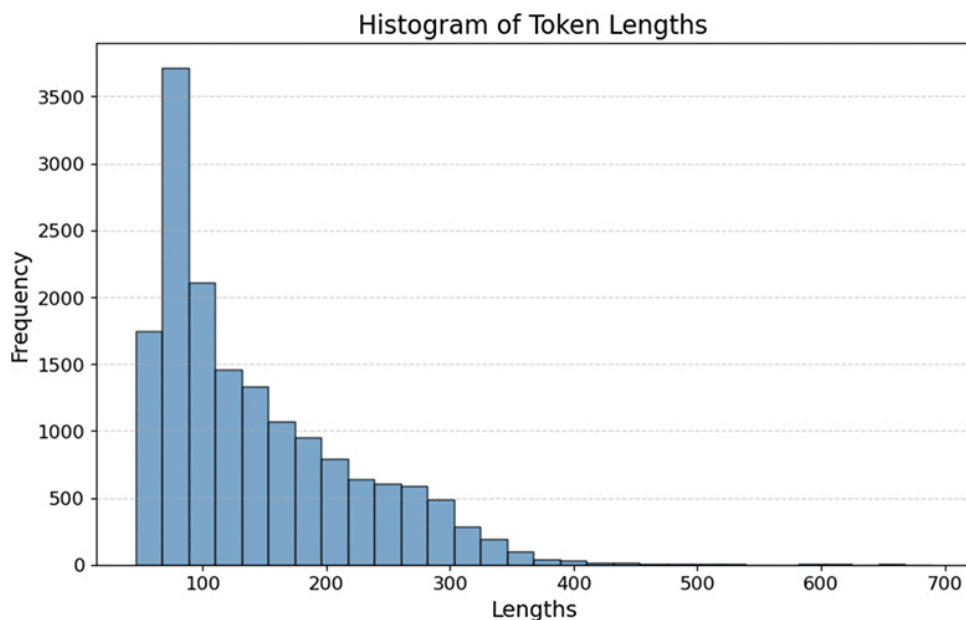


Figure 7.8 We have several animes that, after tokenizing, are hundreds of tokens long. Some have more than 600 tokens.

In **Listing 7.3**, we change the input sequence length to be 384 instead of 128.

Listing 7.3 Modifying an open-source bi-encoder’s max sequence length


```
from sentence_transformers import SentenceTransformer

# Load a pre-trained SBERT model
model = SentenceTransformer('paraphrase-distilroberta-base-v1')
model.max_seq_length = 384 # Truncate long documents to 384 tokens
model
```

Why 384?

- The histogram of token lengths ([Figure 7.8](#)) shows that 384 would capture most of our animes in their entirety and would truncate the rest.
- $384 = 256 + 128$, the sum of two binary numbers, and we like binary numbers. Modern hardware components, especially graphics processing units (GPUs), are designed to perform optimally with binary numbers so they can split up workloads evenly.
- Why not 512, then, to capture more training data? We still want to be conservative here. The more we increase the maximum token window size, the more data we will need to train the system, because we are adding parameters to our model and therefore there is more to learn. It will also take more time and compute resources to load, run, and update the larger model.
- For what it's worth, I did initially try this process with an embedding size of 512. I got worse results, and the process took approximately 20% longer on my machine.

To be explicit, whenever we alter an original pre-trained foundation model in any capacity, the model must learn something from scratch. In this case, the model will learn, from scratch, how text longer than 128 tokens can be formatted and how to assign attention scores across a longer text span. It can be difficult to make these model architecture adjustments, but it is often well worth the effort in terms of performance. In our case, changing the maximum input length to 384 is only the starting line because this model now has to learn about text longer than 128 tokens.

With modified bi-encoder architectures, data prepped and ready to go, we are ready to fine-tune!

Fine-Tuning Open-Source Embedders Using Sentence Transformers

It's time to fine-tune our open-source embedders using Sentence Transformers. A reminder: Sentence Transformers is a library built on top of the Hugging Face Transformers library.

First, we create a custom training loop using the Sentence Transformers library shown in [Listing 7.4](#). We use the provided training and evaluation functionalities of the library, such as the `fit()` method for training and the `evaluate()` method for validation.

Listing 7.4 Fine-tuning a bi-encoder using custom data

[Click here to view code image](#)

```
# Create a DataLoader for the examples
train_dataloader = DataLoader(
    train_examples,
    batch_size=16,
    shuffle=True
)

...

# Create a DataLoader for the validation examples
val_dataloader = DataLoader(
    all_examples_val,
    batch_size=16,
    shuffle=True
)

# Define the loss function
loss = losses.CosineSimilarityLoss(model=anime_encoder)

# Train the model
num_epochs = 1
warmup_steps = int(len(train_dataloader) * num_epochs * 0.1) # 10% of training
for warm-up

# Get initial metrics
anime_encoder.evaluate(evaluator) # Initial embedding similarity score: 0.1475
# Configure the training process
anime_encoder.fit(
    # Set the training objective with the train dataloader and loss function
    train_objectives=[(train_dataloader, loss)],
    epochs=num_epochs, # Set the number of epochs
    warmup_steps=warmup_steps, # Set the warmup steps
    evaluator=evaluator, # Set the evaluator for validation during training
    output_path="anime_encoder" # Set the output path for saving the fine-tuned m
)

# Get final metrics (better!!)
anime_encoder.evaluate(evaluator) # Final embedding similarity score: 0.3668
```

Our job now is to update the underlying LLM (BERT, in this case) to embed anime titles in such a way that if animes are co-liked among the audience (which we labeled via the Jaccard similarity), their embeddings will be similar. We now need to decide on several hyperparameters, such as learning rate, batch size, and number of training epochs. I have experimented with various hyperparameter settings to find a good combination that leads to optimal model performance. I will dedicate most of [Chapter 10](#) to discussing dozens of open-source fine-tuning hyperparameters—so if you are looking for a deeper discussion of how I came to these numbers, please refer to [Chapter 8](#).

We gauge how well the model learned by checking the change in the cosine similarity on our test set. The score jumped up from an average of 0.15 to 0.37! That's great.

With our fine-tuned bi-encoder, we can generate embeddings for new anime descriptions and compare them with the embeddings of our existing anime database. By calculating the cosine similarity between the embeddings, we can recommend animes that are most like the user's preferences.

Once we go through the process of fine-tuning a single custom embedder using our user preference data, we can then relatively easily swap out different models with similar architectures and run the same code, rapidly expanding our universe of embedder options. For this case study, I also fine-tuned another LLM called `all-mpnet-base-v2`, which (at the time of writing) is regarded as a very good open-source embedder for semantic search and clustering purposes. It is a bi-encoder as well, so we can simply swap out references to our RoBERTa model with mpnet and change virtually no code (see GitHub for the complete case study).

Summary of Results

To recap the steps we took during this case study, we performed the following tasks:

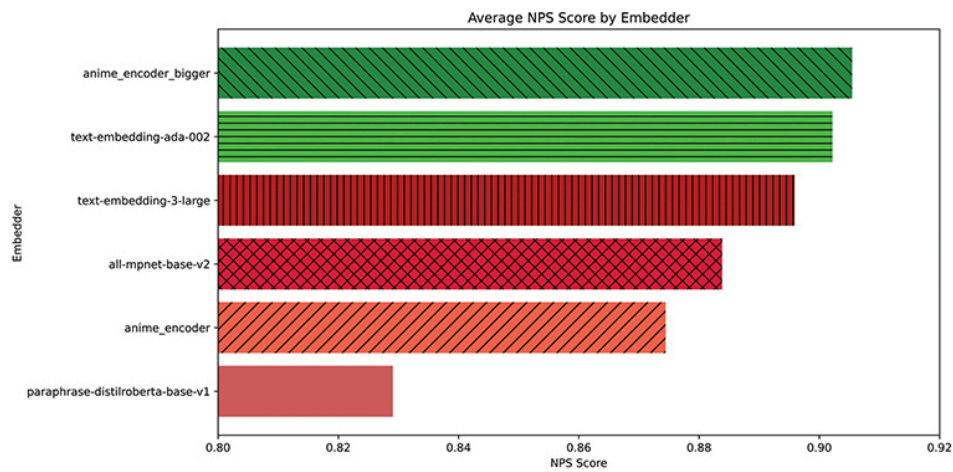
- Generated a custom anime description field using several raw fields from the original dataset
- Created training data for a bi-encoder from users' anime ratings using a combination of NPS/Jaccard scoring and our generated descriptions
- Modified an open-source architecture model to accept a larger token window to account for our longer description field (see [Listing 7.3](#))
- Fine-tuned two bi-encoders with our training data to create a model that mapped our descriptions to an embedding space more aligned to our users' preferences

- Defined an evaluation system using NPS scoring to reward a promoted recommendation (i.e., users giving an anime a score of 9 or 10 in the testing set) and punishing detracted titles (i.e., users giving it a 1–6 score in the testing set)

We had six candidates for our embedders:

- **text-embedding-002** : OpenAI’s older embedder for all use-cases, mostly optimized for semantic similarity
- **text-embedding-3-small** : OpenAI’s newer smaller embedder
- **text-embedding-3-large** : OpenAI’s newer preferred embedder for all use-cases, mostly optimized for semantic similarity in multiple languages
- **paraphrase-distilroberta-base-v1** : An open-source model pre-trained to summarize short pieces of text with no fine-tuning
- **anime_encoder** : The same **paraphrase-distilroberta-base-v1** model with a modified 384-token window and fine-tuned on our user preference data
- **anime_encoder_bigger** : A larger open-source model (**all-mpnet-base-v2**) pre-trained with a token window size of 512, which I further fine-tuned on our user preference data, in the same way and using the same data as for **anime_encoder**

Figure 7.9 shows the final results for our six embedder candidates. In these charts, we are testing our models on a holdout testing set by letting each embedder offer a sample of animes to users based on their likes and measuring the NPS (–1 if they ranked the title 1–6, 0 if they gave it a 7 or an 8, and 1 if they gave it a 9 or a 10).



Top: Overall NPS ratings on the test show our fine-tuned mpnet model outperforms
Bottom: Breaking the recs down by buckets reveal our fine-tuned embedder is stronger in the short and long tail of recommendations but struggles a bit in the middle

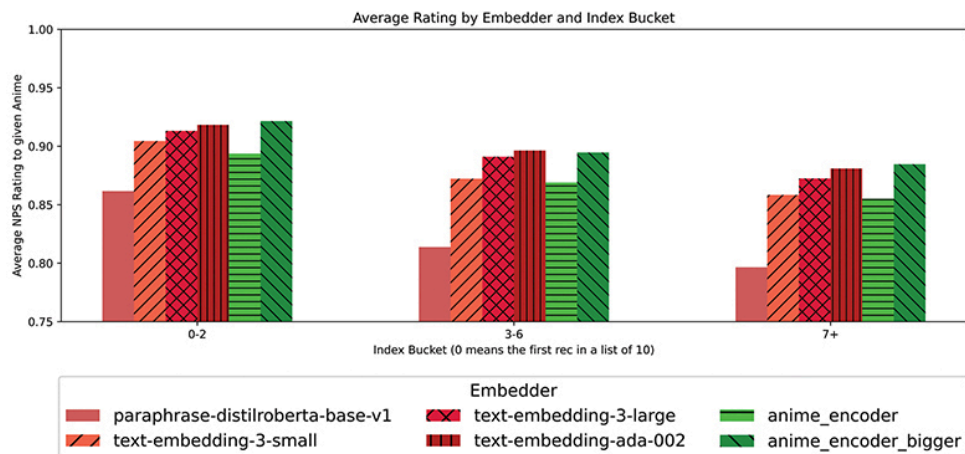


Figure 7.9 Our larger open-source model (`anime_encoder_bigger`) outperforms OpenAI's embedders in recommending anime titles to our users based on historical preferences.

Some interesting takeaways:

- The best-performing model is our larger fine-tuned model. It consistently outperforms OpenAI's embedder in delivering recommendations to users that they would have loved!
- The fine-tuned `distilroberta` model (`anime_encoder`) outperforms its pre-trained cousin (base `distilroberta` with no fine-tuning).

- All models start to degrade in performance when expected to recommend more and more titles, which is fair. The more titles any model recommends, the less confident it will be as it goes down the list.

Exploring Exploration

Earlier I mentioned that a recommendation system’s level of “exploration” can be defined as how often it recommends something that the user may not have watched yet. We didn’t take any explicit measures to encourage exploration in our embedders, but it is still worth seeing how they stack up. **Figure 7.10** shows a graph of the number of unique anime titles recommended to all the users in our test dataset.

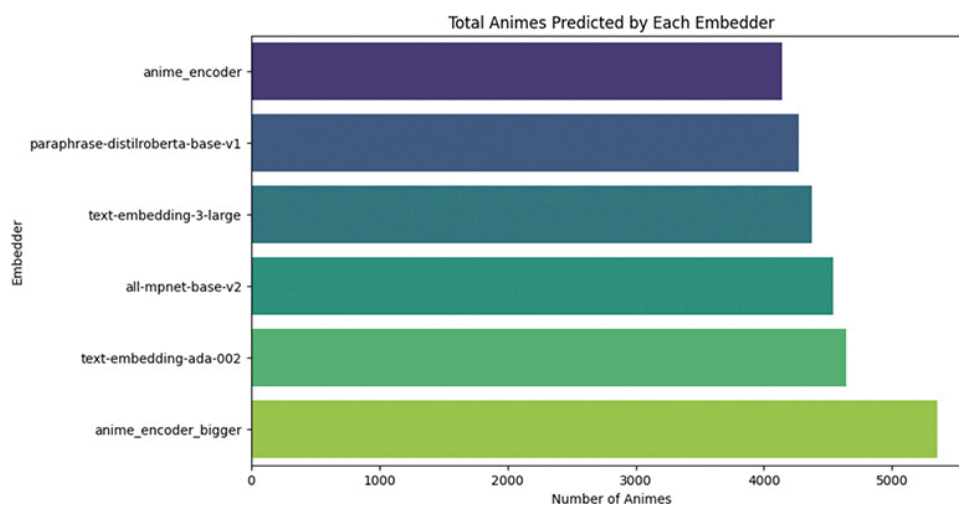


Figure 7.10 Comparing how many unique animes were recommended during the testing process. Our fine-tuned model not only seems to be performant, but also is giving out the most unique recommendations.

Our fine-tuned model is not only performant, but also seems to be in the lead in terms of the diversity of unique animes recommended. This is great because it truly is a win-win. Our model is giving out more recommendations, providing more opportunities for users to rate diverse titles, and the recommendations are more likely to be rated higher.

To answer these questions and more, we would want to continue our research. For example, we could:

- Try different open-source models as the underlying model for our bi-encoder and closed-source models such as Cohere’s embedding

service.

- Calculate new training datasets that use other metrics like correlation coefficients instead of Jaccard similarity scores.
- Toggle the recommendation system hyperparameters, such as k . We only considered grabbing the first $k = 3$ animes for each promoted anime—what if we let that number vary as well?

There is more than one way to recommend an item, product, or anime title, and this chapter merely scratches the surface of the universe of options available to us when building performant recommendation engines. For now, we leave it to you to expand on our work—go have big ideas!

Summary

This chapter walked through the process of fine-tuning open-source embedding models for a specific use-case—generating high-quality anime recommendations based on users’ historical preferences. Comparing the performance of our customized models with that of OpenAI’s embedder, we observed that a fine-tuned LLM powering an embedding model could consistently outperform OpenAI’s embedder.

Customizing embedding models and their architectures for specialized tasks can lead to improved performance and provide a viable alternative to closed-source models, especially when access to labeled data and resources for experimentation is available. I hope that the success of our fine-tuned model in recommending anime titles serves as a testament to the power and flexibility that open-source models offer, paving the way for further exploration, experimentation, and application in whatever tasks you might have.