# Beyond Metadata for BBC iPlayer: an autoencoder-driven approach for embeddings generation in content similarity recommendation

Simone Spaccarotella

September 2024

## Contents

## Listings

## 1 Introduction and background

I am a Software Engineer at the BBC, Team Lead for the Sounds web team, and I have been training as a Data Scientist, working in attachment to the iPlayer Recommendation team.

I built a machine learning model pipeline that generates content-to-content (C2C) similarity recommendations of video-on-demand (VOD) for the "More Like This" section on BBC iPlayer [3]. This project is relevant to me because I have been crossing paths with the world of recommendations multiple times during my career at the BBC, which sparked my interest. I had a tangent encounter in 2015 while working for a team that built an initial recommender for BBC News and an API to provide recommendations using 3rd party engines. I also produced and presented a talk for a Hack Day. The talk was called "Recommendation Assumptions" [17], and it was about types of recommendations and contextual external factors affecting them. Until now, when I was able to finally put my knowledge into practice with an actual project on real data.

The BBC is a well-known British broadcaster that constantly evolves to remain relevant to its audience. Its mission is to inform, educate, and entertain, and it operates within the boundaries set by the Royal Charter [7]. The current media landscape requires the BBC to deliver digital-first content relevant to the audience. This transformation involves investments in data and personalised services, not to mention a certain revolution in generative machine learning modelling that is keeping everyone busy.

# 2 Outline of the issue, opportunity and the business problem to be solved

The BBC produces and stores vast amounts of data for its content, surfaced by countless services and APIs. One of the top priorities for the BBC is to increase the usage across the business of **Passport** [8], an internal BBC system that provides a rich set of metadata annotations for multimodal content (audio, video and text). The usage in production is low, and its BBC-wide adoption would make access to metadata consistent, remove duplications, and reduce effort and costs.

In addition, the current C2C solution is suboptimal and limits the quality of the generated item embeddings. The similarity score is directly proportional to the number of values in common between any pairs of items on a per-feature basis. However, the commonality is calculated with exact string equality, ignoring any relationship between different categorical values expressing a similar concept (e.g. "comedy" and "stand-up comedy"). Moreover, the limited number and type of tags cannot adequately describe the content. At the same time, the skewedness of the data distribution and a lack of pre-processing, shifts the recommendations towards the most popular categories. Lastly, each similarity score is multiplied by a hardcoded weight that modulates the importance of a feature, but it doesn't solve the polarising effect of a skewed distribution. Unfortunately, because these are hyperparameters and not learned weights, the model can't improve its performances by minimising them against a cost function.

I identified three main opportunities for improvement that can help the BBC meet its goals: enhancing the quality of recommendations, reducing costs, and accelerating innovation, thus increasing the licence fee value for money. These opportunities are:

- **Improve the quality of the content similarity recommendations**. This solution ingested a rich set of metadata that better described the content and applied a novel technique that improved the descriptive power of the transformed metadata, which improved the quality of the similarity calculation.
- **Build a foundational item-embeddings generator for upstream recommenders**. This project provided an immediate solution for non-personalised C2C recommendations that solely rely on content metadata. It also provided a foundational approach for embedding generation for upstream personalised recommenders, which could improve the

quality of the personalised recommendation and increase user engagement.

- **Reduce costs by building a general solution for the wider BBC**. The pipeline ingested a dataset of content metadata composed of tags used to annotate all BBC content, making this solution general and reducing duplications and costs.

# 3 Methods and justification

## 3.1 Data pre-processing

I used **one-hot encoding** to transform the categorical features (i.e. the metadata annotations) into a numerical vector. It is a simple yet effective encoding method and is perfect for transforming nominal categoricals because it doesn't introduce any ranking or arithmetic relationship among the encoded values. The downside of this approach is that it generates high-dimensional sparse arrays, introducing the so-called *curse of dimensionality* problem. Nonetheless, this curse became a blessing for the adopted modelling technique, which needed high-dimensional data to produce quality results.

## 3.2 Modelling and regularisation

I trained an **autoencoder** [1, 15] to learn the Passport tags' latent features and reduce the encoded vectors' size. The autoencoder is an encoder-decoder neural network, a self-supervised model capable of capturing non-linearity from the data. I used the "undercomplete" variant, which constrains the number of nodes in the hidden layers, creating a "bottleneck" of information flow through the network. This bottleneck is a form of *regularisation* that forces the model to learn latent attributes from the input while reconstructing it with minimal loss. Ultimately, it helps prevent the model from overfitting the training dataset by indexing it like a caching layer.

I extracted the trained *encoder* segment of the network to compress the one-hot encoded high-dimensional sparse array into a lower-dimensional and denser representation called *embedding* [10]. This technique solved the curse of dimensionality and the data sparsity problems and improved the calculation complexity and the quality of the recommendations at inference time.

To improve and assess the ability of the model to *generalise* on unseen data, I randomly shuffled the dataset and split it into three chunks: training, validation and test. I used **hyperparameter tuning** to find the best model parameters that minimised the cost function and used the validation set to regularise the model with **early stopping**. This technique monitored the reconstruction loss on an out-of-sample dataset, allowing the model to stop training within a set "patience" threshold after reaching a local minimum on the validation error. The test set was finally used to assess the model's performance using the best set of parameters.

I used **dropout** to further regularise the model and make it robust to small changes in the input, and **data augmentation**, by including in the training data the episodes that shared the same tags with their parent program. **Weight decay** and **batch normalisation** were tested during hyperparameter tuning and discarded for poor performance.

## 3.3 Content similarity

Content similarity was calculated with the **cosine** of the angle $\theta$ between each pair of embeddings [11]. This metric is insensitive to the magnitude of the vectors. Because high-frequency values tend to have a larger magnitude, it mitigates the impact of popularity in the similarity calculation. One-hot encoded vectors lack meaningful relations between them. They represent unit vectors bound in the "positive quadrant" of a Cartesian coordinate system for a multidimensional Euclidean space. Because each pair can only have a finite number of angles, the cosine similarity will also assume a finite number of discrete values between 0 and 1, causing information loss. Ideally, we would expect the similarity score to assume a continuous value bound between -1 and 1, and this is only possible if the angle $\theta$ of any vector pair can assume a value between 0 and 360 (i.e. $0\pi$ and $2\pi$), hence the use of embeddings.

## 3.4 Tools and frameworks

The entire project was written in **Python**. It is the *de facto* programming language for data science and machine learning tasks. Python has an established, diverse and well-documented ecosystem of external libraries and frameworks that facilitated the job, and it is also the language of choice at the BBC.

I used **Pandas** only for tabular data manipulation to generate and store the one-hot encoded vectors. Unfortunately, using it for exploratory data analysis (EDA) wasn't possible because the iPlayer catalogue had roughly one year's worth of data, which didn't fit in memory, causing Pandas to crash. Therefore, I used **Dask**, a library capable of running out-of-memory and parallel execution for faster processing on single-node machines and distributed computing on multi-node machines while using the familiar Pandas API.

I used **TensorFlow** and **Keras** for modelling to build and train the encoder-decoder neural network architecture. **Keras Tuner** for hyperparameters tuning. I also used **Scikit-learn** but not for modelling. It provided utility functions for the dataset splitting and the cosine similarity calculation, and I was already familiar with its API.

I used a combination of **Matplotlib** and **Seaborn** for visualisation and **rdflib** to fetch and parse the RDF documents from the BBC Ontology to extract the labels needed to visualise the recommendations for testing purposes. Worth also mentioning the use of **pytest** for unit testing and **black** for PEP 8 code compliance and formatting. Finally, I used **Jupyter Lab** to edit the project, **git** for code versioning, **GitHub** as a remote code repository and for collaboration, and **AWS Sagemaker** to run the pipeline on more capable virtual machines, especially during hyperparameter tuning.

# 4 Scope of the project and Key Performance Indicators

The scope of this project was to build an end-to-end machine learning solution that could produce non-personalised content-to-content similarity recommendations using Passport metadata tags as input.

The minimum viable outcome was to produce recommendations comparable to those currently in production, while the desired outcome was to increase user engagement. The integration with Passport would make this solution general and applicable to multimodal BBC content. If adopted by $N$ BBC products with a total cost of $C$, it could generate considerable savings, with an approximate cost reduction by a factor of $\frac{C}{N}$.

Comparability was a qualitative and subjective key performance indicator (KPI) that served as a compass, indicating whether the project was progressing towards the right direction. Technical and non-technical stakeholders with diverse domain knowledge and background were involved. I built a rudimentary tool that visualised the programme's title, image, description and Passport tags, comparing the seed programmes with the top-K recommendations. The people involved gave their subjective feedback on their perceived level of similarity of the output, testing edge cases and sensitive recommendations like content recommendations for children's accounts. They also discussed anomalies and unexpected results. Opening the More Like This tab on any program page on BBC iPlayer allowed them to compare the live recommendations with those generated by the pipeline.

The solution needed to be A/B tested in production, with live data, to measure user engagement. Unfortunately, too many moving parts outside my control were required to happen for me to build a production-worthy version to achieve that. Adopting this as a KPI would have delayed the project, increasing the odds of failure. To mitigate this risk, I had to decouple it from the project's success.

I defined a hypothesis that could be tested offline and within my control. The hypothesis stated that long-term user engagement is not just about accuracy. It can be affected by increasing diversity in the recommended content. A diverse set of recommendations generates new and unexpected results, increasing surprise and serendipity, pushing the user away from boredom. This theory was untested but grounded in active research on the topic, such as [13] and [9], which made it less far-fetched. For this reason, I decided to measure diversity offline based on the frequency and distribution of the tags that annotated the recommendations, pending future A/B testing to validate the hypothesis.

# 5 Data selection, collection and pre-processing

Recommended programs can be one-offs like movies or documentaries or have episodes like series or live shows. A program is mapped internally by a hierarchy of items defined by an ID called PID and a type. Each item of this tree-like structure represents a part of the program (a single episode, a season, or the program itself) and shares the same Passport tags by inheritance. These are the observations stored in the dataset, replicating the same set of tags for any program. This duplication was used as a data augmentation training technique to increase the model's performance.

BBC News and Sport articles, iPlayer videos, and Sounds audios are annotated with Passport tags. These tags describe any content produced by the BBC and can be used for retrieval (search) and filtering (recommendations). They can be applied either manually by an editorial team with domain knowledge or semi-automatically by machine learning algorithms with human supervision.

Passport tags are distributed across the BBC via the universal content exposure and delivery (UCED) system. This self-service delivery platform exposes data as a document stream for products to integrate. It provides different types of consumers, such as REST API, AWS S3 bucket, etc. Passport documents are JSON objects that contain a property called "`taggings`", an array of objects representing the metadata annotations. Two properties describe these objects: "`predicate`" and "`value`". They represent a tag's name and value and are expressed as URL-formatted strings, except for dates. The predicate is a class of the BBC Ontology [2], while the value can be a date or an entity defined as an RDF [20, 18] document, accessible in Turtle format [19] via the BBC Things API [4, 5, 6]. These entities are linked to each other and external resources and are described by attributes and relationships, giving the data a graph structure.

I decided not to integrate with UCED during development but to use batches of Passport files, manually collected and stored in a local folder. This trade-off allowed me to train the model with live data while keeping costs down. In addition, because I had to create resources on two AWS accounts, I didn't want to pass the burden of maintenance to the team that owned them without having tested the feasibility of the solution first.

Content metadata does not constitute personal data and, therefore, is not subject to the UK GDPR [12]. Nonetheless, this data is encrypted at rest and in transit by default. For this reason, no further actions were required during storage and processing.

I chose to use Passport because it provides a set of tags shared across all content produced by the BBC, making this a general solution that reduces duplications and costs. Passport offers a flexible and rich set of tags to describe any type of content. Annotations can describe canonical information such as *genre* and *format*, the *contributor* featured in the programme, the *narrative theme*, the *editorial tone*, what the content is *about* or what relevant entities are *mentioned*, and many more.

The pre-processing stage of the pipeline loaded a list of JSON files and extrapolated the tags into a dictionary data structure. The dictionary's key was the programme ID - known internally as *PID* - and the value was another dictionary describing the annotations. A program can be tagged with the same predicate multiple times if it has different values, while the same value (e.g. "Music") can be used by various predicates (e.g. `about` or `genre`).

The pipeline then transformed this dictionary data structure into a Pandas *Dataframe*, where the rows represented the programmes, and the columns represented the tags. I used a MultiIndex [16] for the columns because I needed to keep the duplicate values (2nd-level index) across the predicates (1st-level index). It then populated the cells of the *Dataframe* with the value "`1`" if the program was annotated with the corresponding tag or "`0`" otherwise, generating one-hot encoded arrays. I initially adopted a vectorisation approach known to be performant, using the "`get_dummies`" Pandas function. Surprisingly, this method was slower and less scalable than the solution that I adopted in the end.

A source of bias in the dataset was the "`mentions`" tag. This annotation type is automatically generated by an algorithm that extracts terms deemed important, appearing in the text of an article or the transcript of an audio/video content. If something is "mentioned", it doesn't necessarily describe what the content is about because of the intrinsic ambiguities of natural languages. Figure of speech devices, such as metaphors, analogies, allegories, and others, alter the meaning of a sentence for stylistic effect and can misrepresent the main topic. For example, if the phrase "being over the moon" is mentioned by someone delighted about something unrelated to the topic of "space" and "universe", extracting the term "moon" as a descriptor could mislead the representation. To mitigate this source of bias, I dropped the tag in favour of "`about`", another tag that describes what the content is really about. A team of editorials annotates content with this tag, using relevant topics.

Generating embeddings of one-hot encoded vectors with the same size used in training but with unseen tags leads to unpredictable errors. The encoding is positional, and the combination of `1` and `0` learned by the model belongs to the tags seen during training. So, I decided to drop the new tags and encode only the ones the model was trained on while padding the rest with zeros, pending retraining to capture the new information. Also, some programmes didn't have any annotations entirely. Adding them to the training data, created a group of entries with all zeros. If enough observations shared this uninformative characteristic, the model could have picked it up. I decided to drop these programmes and include only the ones with at least one annotation.

## 6 Survey of potential alternatives

**Clustering** the one-hot encoded features to group similar items wasn't viable. Recommenders return ranked lists, while clustering would have returned an unordered list of items belonging to the same cluster as the one considered for similarity. Furthermore, the number of clusters was unknown. Even using a density-based technique to auto-discover them would not have helped because similarity ranking requires every pair of items to have a score, and grouping does not map with this concept. Clustering was a coarse-grained discretisation of similarity, and I needed a more granular approach where every item could be compared with everything else.

The intuition was to map the concept of similarity with a geometric interpretation and calculate the **pairwise distance** between the vectors representing the items, given a metric and a vector space relevant to the similarity measure. So, I discarded clustering as a candidate option.

One-hot encoding doesn't use spatial proximity information to transform the categorical features into ones and zeros. It's a transformation process that pivots the unique values of each original feature to be the new variables of the transformed vector. If we project these raw vectors in a multidimensional space, we wouldn't be able to use their relative position to each other as a similarity measure. Moreover, the high dimensionality of the vectors would have increased the computational complexity.

To calculate the pairwise distance efficiently and produce a meaningful representation of similarity, the vectors needed to be transformed in a denser and lower dimensional space, a manifold embedded into the original high-dimensional ambient space.

I considered dimensionality reduction techniques such as **principal component analysis (PCA)**, **independent component analysis (ICA)** or **linear discriminant analysis (LDA)**, but there was a problem with them too. Their job is to find a linear projection of the data, but this is a strong assumption that misses important non-linear structures. I didn't use PCA (which I was more familiar with), but I didn't discard the idea of using dimensionality reduction. I just needed a non-linear approach and turned my attention to manifold learning.

I could have chosen an entirely different encoding technique to generate lower-dimensional vectors and avoid the curse of the dimensionality problem from the beginning, so I also investigated **feature hashing**. A hash function is a non-invertible function that can map data of arbitrary size to fixed-size values typically used for constant lookups. Hashing can also be used as a feature transformation technique for tabular data. It can generate smaller vectors than one-hot encoding but doesn't encode any concept of similarity either. I didn't adopt it because, in comparison, it has more hyper-parameters, which increases its complexity. In addition, although unlikely, it introduces the collision problem, where two distinct inputs can be mapped to the same index in the same target domain. This issue could be mitigated by choosing the latest and most robust algorithm to reduce the likelihood of that happening. However, the trade-off was too computationally expensive for pre-processing. The reality was that I needed a pre-processing technique to transform categorical features into a high-dimensional numerical vector to artificially inflate the encoding dimension for the modelling technique I had in mind since the beginning to produce quality embeddings.

## 7 Implementation and performance metrics

I used an *undercomplete autoencoder* to compress the high-dimensional one-hot encoded vectors to a lower-dimensional embedding representation. This technique captured non-linearity by learning the underlying latent structure of the data, which was vital to represent the original Passport tags in a geometrical space, exploiting local proximity as a measure of similarity.

Because the autoencoder has a symmetrical architecture between the *encoder* and the *decoder*, the input and output layers had equal dimensions: 8982 nodes. This number corresponded to the size of the one-hot encoded array.

Some of the hyperparameters were decided based on the nature of the problem. The input data was a tensor of zeros and ones and the main objective of the network was to reconstruct the output with minimal loss. I used the *Sigmoid* activation function for the output layer and *binary cross-entropy* as a loss function to allow the network to push the values of the reconstructed output as close as possible to 0 and 1. I didn't use the *SoftMax* activation function for the output layer because each single node needed to be able to assume those values. I didn't use any residual-based loss function either because they don't have a steep slope when the prediction is far from the actual value, allowing the gradient to move fast enough. The negative log loss was the best choice because it heavily penalises significant differences between $y$ and $\hat{y}$, with a natural logarithmic progression.

I used the *Rectified Linear Unit (ReLU)* as an activation function for the hidden layers while testing other variants like *Leaky ReLU (LReLU)* and *Parametric ReLU (PReLU)*, but without any relevant improvements in performance. Some other hyperparameters were:

- the optimizer: Adam
- number of epochs: 100
- batch size: 300
- dropout rate: 0.2
- early stopping patience: 10
- early stopping monitoring: binary cross-entropy on the validation set
- data split: 80% training, 10% validation, 10% test

A final round of hyperparameter optimisation selected the remaining ones. I decided to use a "Bandit-based" approach called *Hyperband* [14], which improves upon *Random Search* by running fewer epochs on the randomly sampled set of parameters and moving on to the next stage by only testing the best-performing ones, returning a ranked list of the best hyperparameter sets. The hyperparameters considered by Hyperband were:

- the number of hidden layers
- the embedding size
- the learning rate
- whether to use dropout
- whether to use batch normalisation

The number of hidden layers was a positive odd integer $N > 0$. The encoder and decoder had $N - 1$ layers, and one was used for the bottleneck in the middle. If the value were 1, the network would only have the bottleneck.

The number of nodes per layer was a positive integer $N > 0$. Unfortunately, the number of combinations was too high to be meaningfully optimised. To reduce the complexity, I decided to couple the number of nodes per layer as a progression of integer divisions by 2. Each hidden layer in the encoder had half the number of nodes compared to the previous one, and double the amount of the successive one (and *vice versa* in the decoder), except for the layers close to the input and output. In that case, they could assume any of the values, depending on the number of hidden layers and embedding size. For example, because I had 8982 nodes in input, the progression of layer dimensions was `[4491, 2245, 1122, 561, 280, 140, ...]`. The embedding size was also bound to assume one of these values and it would determine the maximum number of hidden layers allowed. Therefore, a network with `5` hidden layers and an embedding size of `280` would have had the following configuration: `8982 -> 1122 -> 561 -> 280 -> 561 -> 1122 -> 8982`. While a network with `3` hidden layers but an embedding size of `2245` would have had the following configuration: `8982 -> 4491 -> 2245 -> 4491 -> 8982`, representing the maximum extension of the progression.

The strength of this approach was the efficient use of space and the fast inference time. The space scaled linearly with respect to the input size to store the embeddings and quadratically to store the similarity scores. Hyperparameter tuning took quite some time, but this drawback was compensated by the fact that training was performed offline while the recommendations were cached and served instantly. In addition, retraining could be scheduled to cover the new programmes added to the iPlayer catalogue and the unavailable ones being removed. The main weakness was the model's interpretability. The autoencoder is a black box by definition, and using embeddings to calculate the similarity just worsened the problem. It was practically impossible to interpret which of the tags influenced the ranking in the top-K recommendation by untangling the weights and biases of the neural network. It was also challenging to provide explanations using model-agnostic techniques like SHAP because the recommendations were calculated by some distance metric, applied on humanly meaningless embeddings that needed to be linked to the original input, which needed to be decoded back to the original tags.

# 8 Discussion and conclusions

## 8.1 Results

This general solution works with any content that uses Passport tags, and could provide recommendations for multiple BBC products. Adopting it would reduce effort, duplication of code and data, and, consequently, costs. I shared the findings with the stakeholders, explaining the main benefits and showing the results using the visualisation tool I built. I presented it once to the data scientists and engineers of the iPlayer recommendations team I worked with and another time to the team in charge of the non-personalised recommendations for the entire BBC. The feedback was positive in both cases, and we discussed how to move forward with this project, including the possibility of an A/B test.

## 8.2 Summary of findings and recommendations

The results were perfectly aligned with the initial objectives and measure of success set at the beginning of the project. I recommended building an initial minimum viable product (MVP) consisting of a Sagemaker pipeline built on the AWS development account that ingested batch Passport tags. This recommendation would allow us to break down the engineering effort and spot any blockers/challenges that must be addressed as early as possible to correct them or reconsider some assumptions ahead of the production build.

We would need to build two pipelines, one for training and one for inference to generate the embeddings and the similarity scores. The embeddings and the similarities score need to be cached to improve performance. The second stage of this approach would require integrating UCED to fetch real-time data automatically.

If this solution is viable and passes the A/B test, it could also be employed to generate embeddings for other personalised recommenders that use item metadata in conjunction with user interactions and contextual data such as day and time of interaction, location and device used.

The project could be further expanded by exploiting the graph nature of the data using a graph neural network (GNN) and, in particular, a graph autoencoder (GAE) to learn a meaningful representation of the graph data, capturing the topological structure and the node content. This extension could improve upon the current autoencoder, which flattens the graph structure in a list of tags and relies on the positional encoding of these tags to generate the embeddings. This effort will require further research and prototyping.

## 8.3 Implications

The project presented a unique opportunity for me to work on an end-to-end machine learning pipeline from data preprocessing to inference, practising my technical skills, building a real neural network, and learning about embedding techniques and content-based recommendation systems. The positive feedback from stakeholders has reinforced my professional confidence and provided invaluable experience in presenting data-driven solutions to a business audience.

For my colleagues and the team, this project has established a replicable framework for C2C similarity recommendations that can be adapted to other BBC products. The solution's modularity enables flexibility in extending it to multimodal content, enhancing the potential for collaborative developments across departments. This adaptability can promote knowledge sharing and foster a data-centric approach to problem-solving in the broader team, as members can leverage this solution to address similar business problems and build upon it

The project presents a scalable solution for stakeholders and the business to reduce data redundancies, decrease maintenance overhead, and potentially reduce costs associated with content recommendation systems. The approach provided a standardised method for generating "More Like This" suggestions, potentially improving user engagement on non-personalised content, with a consistent experience across the BBC portfolio. Moreover, the project's adaptability encourages strategic, data-driven content management across the organisation, supporting future initiatives with robust foundations for content similarity and recommendation. Overall, this project not only aligns with the business goals of optimising resource allocation but also empowers the organisation with a sustainable, scalable recommendation solution for future developments.

## 8.4 Caveats and limitations

Data drift can cause a significant decrease in performance, requiring a model re-training. When new programs are added to the iPlayer catalogue, their feature vectors must be encoded, the encoded vectors must be transformed into embeddings, and the new cosine similarities must be re-calculated. Also, both the encodings and the embeddings need to be cached. When the new programs don't share some or all of the tags, the encoding phase produces vectors with some or all zeros, indicative of a loss of information.

# 9 Appendices

The Passport tags used for this project covered the programs that were available in the iPlayer catalogue from 13th June 2023 to 15th April 2024 inclusive. The catalogue had 83098 available items, 81311 of which had coverage of Passport tags. The 8 tags considered were: `about`, `genre`, `format`, `contributor`, `motivation`, `editorialTone`, `narrativeTheme`, `relevantTo`. The encoding process generated 8982 new features.

## 9.1 Code and documentation

### 9.1.1 Data encoding

```python
import os
import json
import pandas as pd

def get(something, fromTag):
  '''
    Returns the value of the predicate or the tag, extrapolated from
    the URI-formatted string.

    Predicate example: http://www.bbc.co.uk/ontologies/creativework/genre
    Value example: http://www.bbc.co.uk/things/1c3b60a9-14eb-484b-a750-9f5b1aeaac31#id
  '''
    return fromTag.get(something, '').split('/')[-1].split('#')[0]

def should_skip(predicate):
  '''
    Returns true if the Passport tag is not in the allow list
  '''

  # allow list of predicates to include in the encoding
  return True if predicate not in [
    'about',
    'format',
    'contributor',
    'genre',
    'motivation',
    'editorialTone',
    'narrativeTheme',
    'relevantTo'
  ] else False

tags_dict = dict({})
pids_dict = dict({})
pids_list = []

# lists the Passport files collected from UCED
uced_files = os.listdir('uced')

###
## Extrapolates the Passport tags per PID
###

for file_name in uced_files:

  # gets the PID by splitting "urn:bbc:pips:pid:b05sxyhw.
 json"
  pid = file_name.split(':')[4].split('.')[0]
  pids_list.append(pid)

  # loads the JSON file
  with open(f'./uced/{file_name}') as json_file:
      passport = json.loads(json_file.read())

  tags = dict({})

  # for each Passport tag
  for tag in passport.get('taggings', []):
    # extrapolates the tag's name from the URI
    predicate = get('predicate', tag)

    # checks the allow list
    if should_skip(predicate):
      continue

    # extrapolates the tag's value from the URI
    value = get('value', tag)

    # builds a dictionary of predicates containing a list
 of values related to the current PID
    if predicate in tags:
      tags[predicate].append(value)
    else:
      tags[predicate] = [value] # list

    # Builds a global dictionary of predicates containing
 all the tag values used for all PIDs.
    # These tag values may be repeated across predicates.
    if predicate in tags_dict:
```

```python
      tags_dict[predicate].add(value)
    else:
      tags_dict[predicate] = {value} # set

  # Builds the dictionary of PIDs, with the associated predicate tags
  pids_dict[pid] = tags

###
## We need to build a dataframe where the rows are indexed by the PIDs, and the columns indexed by the Passport tags.
## To this end, we need to create a multi-index for the columns, where the first level is the tag predicate,
## and the second level the tag value. This will allow duplicate of values across predicates, and access to any
## given cell by speficying the PID, the predicate and the value, in order for the algorithm to set the value "1"
## to signal the PID is tagged.
###

columns = []

for key in tags_dict.keys():
  columns.extend(pd.MultiIndex.from_product([[key], tags_dict.get(key)]))

###
## Creates a zero-padded dataframe of type integer with PIDs as row indexes and the Passport predicates and tags
## as multi-level column index, as specified earlier.
###

df = pd.DataFrame(0, index=pids_list, columns=pd.MultiIndex.from_tuples(columns), dtype='uint8')

###
## Sets 1 for each PID and predicate/tag pair (i.e. One-Hot Encoding)
###

for pid in pids_dict.keys():
  for predicate in pids_dict[pid].keys():
    for tag in pids_dict.get(pid).get(predicate):
      df.at[pid, (predicate, tag)] = 1

###
## Drop the first level of the multi-level column index to flatten the dataframe, now that it is no longer needed.
###
```

```python

df = df.droplevel(level=0, axis=1)
```
Listing 1: One-hot encoding

### 9.1.2 Model training and embedding generation

```python
import pandas as pd
from sklearn.model_selection import train_test_split

data = pd.read_parquet('passport_encoding.parquet')

X_train, X_test = train_test_split(data, test_size=0.2)
X_val, X_test = train_test_split(X_test, test_size=0.5)

assert X_train.shape[0] + X_val.shape[0] + X_test.shape[0]
 == data.shape[0]
```
Listing 2: Data splitting

```python
import tensorflow as tf
from tensorflow import keras
import keras_tuner as kt

# https://www.tensorflow.org/tutorials/keras/keras_tuner

input_size = X_train.shape[1]

# https://keras.io/api/keras_tuner/hyperparameters/
def build_model(hp: kt.HyperParameters):
  # Parameters Set
  hidden_layers = hp.Choice('hidden_layers', [1, 3])
  embeddings_size = hp.Choice('embeddings_size', [70, 140,
 280, 560])
  batch_norm = hp.Boolean('batch_norm')
  dropout = hp.Boolean('dropout')
  learning_rate = hp.Choice('learning_rate', [0.1, 0.01,
 0.001])

  activation = 'relu'
  dropout_rate = 0.2

  model = keras.Sequential()
  model.add(keras.layers.InputLayer(input_shape=(input_size
 ,)))

  if hidden_layers == 1:
    model.add(keras.layers.Dense(
      units=embeddings_size,
      activation=activation
    ))
    if batch_norm:
      model.add(keras.layers.BatchNormalization())
    if dropout:
      model.add(keras.layers.Dropout(dropout_rate))

  if hidden_layers == 3:
    model.add(keras.layers.Dense(
      units=embeddings_size * 2,
      activation=activation
    ))
    if batch_norm:
      model.add(keras.layers.BatchNormalization())
    if dropout:
      model.add(keras.layers.Dropout(dropout_rate))

    model.add(keras.layers.Dense(
      units=embeddings_size,
      activation=activation
    ))
    if batch_norm:
      model.add(keras.layers.BatchNormalization())
    if dropout:
      model.add(keras.layers.Dropout(dropout_rate))

    model.add(keras.layers.Dense(
      units=embeddings_size * 2,
      activation=activation
    ))
    if batch_norm:
      model.add(keras.layers.BatchNormalization())
    if dropout:
      model.add(keras.layers.Dropout(dropout_rate))

  if hidden_layers == 3:
    model.add(keras.layers.Dense(
      units=embeddings_size * 4,
      activation=activation
    ))
    if batch_norm:
      model.add(keras.layers.BatchNormalization())
    if dropout:
      model.add(keras.layers.Dropout(dropout_rate))

    model.add(keras.layers.Dense(
      units=embeddings_size * 2,
      activation=activation
    ))
  if batch_norm:
    model.add(keras.layers.BatchNormalization())
  if dropout:
    model.add(keras.layers.Dropout(dropout_rate))

  model.add(keras.layers.Dense(
    units=embeddings_size,
    activation=activation
  ))
  if batch_norm:
    model.add(keras.layers.BatchNormalization())
  if dropout:
    model.add(keras.layers.Dropout(dropout_rate))

  model.add(keras.layers.Dense(
    units=embeddings_size * 2,
    activation=activation
  ))
  if batch_norm:
    model.add(keras.layers.BatchNormalization())
  if dropout:
    model.add(keras.layers.Dropout(dropout_rate))

  model.add(keras.layers.Dense(
    units=embeddings_size * 4,
    activation=activation
  ))
  if batch_norm:
    model.add(keras.layers.BatchNormalization())
  if dropout:
    model.add(keras.layers.Dropout(dropout_rate))

  model.add(keras.layers.Dense(input_size, activation='sigmoid'))

  model.compile(
      optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
      loss=keras.losses.BinaryCrossentropy()
  )

  return model
```

Listing 3: Model training and optimisation using Keras and Keras Tuner

```
# https://keras.io/api/keras_tuner/tuners/hyperband/
tuner = kt.Hyperband(
  hypermodel=build_model,
  objective='val_loss',
  max_epochs=100,
  factor=2
)

early_stop = tf.keras.callbacks.EarlyStopping(
  monitor='val_loss',
  mode='min',
  restore_best_weights=True,
  patience=10
)

tuner.search(
  X_train,
  X_train,
  epochs=100,
  batch_size=300,
  verbose=1,
  callbacks=[early_stop],
  validation_data=(X_val, X_val)
)
```

Listing 4: Hyperband search and early stopping setup

### 9.2 Statistics

### 9.3 Figures and tables

### 9.4 Mapping of the project report to the pass criteria

## References

[1] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *CoRR*, abs/2003.05991, 2020.

[2] BBC. BBC Ontologies. https://www.bbc.co.uk/ontologies/.

[3] BBC. Bluey - "more like this" tab. https://www.bbc.co.uk/iplayer/episodes/m000vbrk/bluey?seriesId=more-like-this.

[4] BBC. Things. https://www.bbc.co.uk/things.

[5] BBC. Things - About. https://www.bbc.co.uk/things/about.

[6] BBC. Things - API. https://www.bbc.co.uk/things/api.

[7] BBC. The Royal Charter. https://www.bbc.com/aboutthebbc/governance/charter, 2017. Valid until 31 December 2027.

[8] BBC. How metadata will drive content discovery for the bbc online. https://www.bbc.co.uk/webarchive/https%3A%2F%2Fwww.bbc.co.uk%2Fblogs%2Finternet%2Fentries%2Feacbb071-d471-4d85-ba9d-938c0c800d0b, 2020. This page was archived on 1st August 2023 and is no longer updated.

[9] Tomislav Duricic, Dominik Kowald, Emanuel Lacic, and Elisabeth Lex. Beyond-accuracy: A review on diversity, serendipity and fairness in recommender systems based on graph neural networks, 2023.

[10] Google. Embeddings. https://developers.google.com/machine-learning/crash-course/embeddings.

[11] Google. Measuring similarity from embeddings. https://developers.google.com/machine-learning/clustering/dnn-clustering/supervised-similarity.

[12] UK Government. UK GDPR. https://www.legislation.gov.uk/eur/2016/679/contents, 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council.

[13] Marius Kaminskas and Derek G. Bridge. Diversity, serendipity, novelty, and coverage. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 7:1 – 42, 2016.

[14] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *CoRR*, abs/1603.06560, 2016.

[15] Umberto Michelucci. An introduction to autoencoders. *CoRR*, abs/2201.03898, 2022.

[16] pandas via NumFOCUS, Inc. MultiIndex / advanced indexing. `https://pandas.pydata.org/docs/user_guide/advanced.html`, 2024.

[17] Simone Spaccarotella. Recommendation assumptions. `https://www.slideshare.net/slideshow/recommendations-assumptions/236291920`, 2015. Presented at Prototyping Day @ Mozilla London on 3 September 2015, at Engineering Summit @ BBC on 7 March 2018, uploaded on SlideShare on 27 June 2020.

[18] W3C. RDF 1.1 Concepts and Abstract Syntax. `https://www.w3.org/TR/rdf11-concepts`, 2014. W3C Recommendation 25 February 2014.

[19] W3C. RDF 1.1 Turtle - Terse RDF Triple Language. `https://www.w3.org/TR/turtle/`, 2014. W3C Recommendation 25 February 2014.

[20] W3C. Resource Description Framework (RDF). `https://www.w3.org/RDF`, 2014. Publication date: 2014-02-25 (with a previous version published at: 2004-02-10).