Semi-supervised deep learning of entity clusters for ontology engineering

A dissertation submitted in partial fulfilment of the requirements for the MSc in Advanced Computing Technologies (Data Analytics)

by William Gilpin

Department of Computer Science and Information Systems,

Birkbeck College, University of London

September 2022

Academic Declaration

William Gilpin

19 September 2022

Contents

1. Abstract

Ontology engineering is the discipline of constructing ontologies, or the set of categories into which entities can be placed. A common task in natural language processing applications is the analysis, prior to deployment, of large amounts of end-user data to find those entity classes which are essential to the users but missing from the standard ontologies used by most systems. This is typically undertaken manually, involving interviews and tagging of entities in the documents.

This project aims to offer a semi-automated solution to this problem by developing tools allowing the unsupervised processing of large numbers of documents much more rapidly than a human could, and to suggest to the ontology engineer a set of clusters for consideration as new entity classes.

The approach is to produce embeddings from a BERT language model, then cluster these using a variety of techniques, with or without further processing, to generate a set of candidate clusters. The data is pre-processed to identify entities as identified by the spaCy Named Entity Recognition tool, and then these are used to train the clustererers, which will then also be able to generate previously unlabelled clusters of topics of relevance to the domain.

The application of the model to real-world data shows promise, in that the model proposes clusters which are relevant to the document domains, but not covered by the spaCy tools used for comparison. Analysis of a previously unseen set of documents related to the Kepler space telescope identified clusters relating to astronomical terms, algorithms used in analysis and formulae, among others. These are good candidates for to entity classes in this domain.

2. Introduction

2.1 Ontologies

The term ontology is used in various contexts. In the natural language processing (NLP) context, it is taken to be the study of the "classification and explanation of entities" (*Ontology*, 2017). The term is used here as "an engineering artefact, constituted by a specific vocabulary used to describe a certain reality" (Fonseca, 2007), or as usually interpreted in Natural Language Processing (NLP), the closed set of possible classifications of the entities and relationships referred to. A common NLP ontology is the Ontonotes 5.0 (ON5) ontology (Xue, 2012), as used by spaCy and NLTK, containing such classifications as PERSON, LOCATION and PRODUCT.

2.2 Embeddings

Clustering requires numeric representation, or at least the ability to measure the distance between two points. To this end, it is necessary to turn the input words into numerical representations. These representations exist in a multidimensional space which aims to preserve whatever structure is essential for the application. The embeddings used in this project are derived from distilBERT, itself derived from BERT, a transformer model from Google (Devlin *et al.*, 2019), described later.

BERT embeddings aim to preserve semantic relationships between entities so that, for example, in Figure 2.1, it can be seen that $\vec{CD}$ is identical to $\vec{KP}$ and could be said to represent "`immature-form-of`", in the sense that kitten is the immature form of cat. BERT embeddings are 768-dimensional, so the figure should be interpreted as the 2-dimensional projection of the embeddings space.

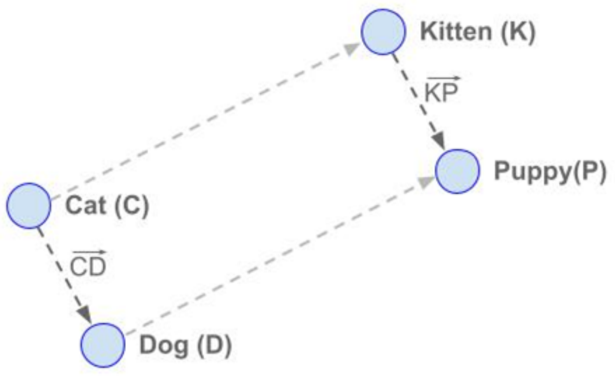


Figure 2.1: Embeddings preserve relationships

2.2.1 Transformers

BERT is a Transformer model. Transformers, introduced in the paper "Attention is all you need" (Vaswani *et al.*, 2017), are an evolution from recurrent neural networks designed to operate as sequence-to-sequence models and typically trained on large volumes of text to perform transduction. The transformer was trained on a massive corpus of around 4.5 million English-German sentence pairs and was able to exceed state-of-the-art scores on BLEU (Papineni *et al.*, 2002) evaluations.

The critical aspects of transformers are aimed at learning the contextual embeddings from the input material in order to accurately represent the meaning of words in the context in which they were used, thus allowing an accurate output (translation in this case) to be generated, reflecting the full semantics of the input. Two significant components enabling the learning of these contextual embeddings are positional encoding and multi-headed self-attention.

Positional encoding is an approach that uses a series of sinusoidal patterns with varying frequencies allowing a predictable numerical representation of the spatial relationships that exist between tokens in the input. This representation has the same dimensionality as the embeddings being learned, allowing simple summation.

Attention (Bahdanau, Cho and Bengio, 2014) is the mechanism whereby weights are assigned to tokens (words), allowing the decoder to determine the relevance of each token to the overall calculation. As per Vaswani et al., self-attention is attention calculated over the input tokens, effectively showing their relevance to each other within the same token stream. Relevance is computed as a similarity measure by a scaled dot product of the embeddings of the tokens in the stream, removing the necessity common in RNNs to assign equal weight to a long stream of tokens. They also found a benefit from deriving, in parallel, several sets of attention values, and then concatenating these to produce the final out vector. This technique is referred to as multi-headed attention.

Vaswani et al. state the transformer is the first such transduction model relying entirely on self-attention to generate the output without using RNNs or convolutions.

BERT (Devlin *et al.*, 2019) is an improved transformer optimised for language tasks. BERT comes from **B**i-directional **E**ncoder **R**epresentations from **T**ransformers, and refers to the learning of a representation of unlabelled text from both the prior and post tokens. The mechanism is masking; hence BERT is considered a masked language model (MLM). MLMs will apply masks at random to the input sequence and learn to predict the missing token(s), thereby learning a deep contextual representation of the language used for training.

2.3 Clustering

2.3.1 K-Means

One of the most common clustering mechanisms used is K-means (Jin and Han, 2010). This is the approach taken by both Xie and Dahal. The algorithm assigns a random location to *k* cluster centroids, and then works iteratively, assigning each point to its nearest cluster followed by updating the centroids to the mean of all the points in it. This way cluster centroids slowly migrate to the centre of groups of points that are not near another cluster.

Algorithm 1 k-means clustering

```
Input: Set of points $P$ and number of clusters to be found $k$
Output: Set of cluster assignments $p_i^\mu \in \{1..k\}$ for each $p_i \in P$

Initialise $k$ cluster centroids randomly in the space of $P$
Repeat until stopping criteria met
  For each point $p_i$
    Calculate the nearest cluster $c$ to $p_i$ and assign $p_i$ to that cluster as $p_i^c$
  For each cluster
    Update centroid position to be the mean of all points in it
Return cluster assignments $p_i^\mu$ for each $p_i \in P$, $\mu \in \{1..k\}$
```

Appropriate stopping criteria might be the number of iterations, or the minimum number of cluster reassignments per iteration.

2.3.2 Gaussian Mixture Models

Gaussian Mixture Models adopt a maximum likelihood approach to determine the most likely mixture of multivariate Gaussian density models describing the data. GMMs assume that the data consists of points drawn from a set of *k* Gaussian components, and then attempt to predict the parameters of those distributions statistically. Maximum likelihood estimation uses KL divergence between Bayesian predictions of the distributions to iteratively determine the maximally likely mixture parameters and hence cluster assignments for the points.

```
Algorithm 2 Gaussian Mixture Models
Input: Set of points $P$ and number of clusters to be found $k$
Output: Set of cluster assignments $c_j$ for each $P$
Initialise $k$ cluster mean and s.d. randomly $c_j = (\mu_j, \sigma_j^2)$ for each $j \in \{1..k\}$
Repeat until stopping criteria met
  For each point $p_i$
    Calculate probability $P(p)$ for all $p$ and $c$
  For each cluster
    Update $= (\mu_j, \sigma_j^2)$ based on member points
Return cluster assignments each $P$
```

For full details of the summarised strep “Calculate probabilities” see (Taboga, 2017). This project uses the scikit-learn implementation of GMMs (*sklearn.mixture.GaussianMixture*, no date)

2.3.3 OPTICS

OPTICS (Ankerst *et al.*, 1999) is a density-based clustering algorithm derived from DBSCAN (Ester *et al.*, no date). OPTICS has two important parameters, ε the maximum distance in which to consider points, and *MinPts*, the minimum number of points allowed in a cluster. All points are assessed initially to see if they might be “core points”, i.e. it has at least *MinPts* other points within a radius of ε. This is effectively a density measure. OPTICS then clusters the core points in a similar manner, deriving regions of density where they are connected via chains of core points, ensuring the clusters found consist of connected core points. OPTICS did not perform well in this project so no further explanation will be provided.

2.3.4 Agglomerative Clustering

Agglomerative clustering (Chidananda Gowda and Krishna, 1978) is a form of bottom-up hierarchical clustering. Initially all points are assumed to be individual clusters, then iteratively pairs are merged into a new cluster using the linkage calculation once the stopping conditions are met. Linkage methods used in this project were Ward and Single.

Single linkage uses the minimum distance between every possible pair of points in the two clusters. Ward linkage (Nielsen, 2016) minimises the variance of the clusters rather than measuring the distance directly.

2.3.5 Manhattan & Euclidean distance metrics

Manhattan distance is named after the grid layout of Manhattan (New York), wherein a taxi can only travel North-South or East-West, meaning the distance travelled is the sum of the easterly component and the northerly component (Craw, 2010).

Given two points in n dimensions, X= $\left(x_1, x_2 \dots x_n\right)$ and $Y = \left(y, y_2 \dots y_n\right)$

the Manhattan distance is given by

$$d(X, Y) = \sum_{i=1}^{n} |x_i - y_i|$$

And Euclidean distance by

$$d(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Aggarwal shows that Manhattan distance diverges to ∞ as dimensionality increases, whereas Euclidean distance converges to a constant – at sufficiently high dimensionality all points are equidistant by Euclidean measures (Aggarwal, Hinneburg and Keim, 2001). This effect is marked above 20 dimensions, and our embeddings are 768 dimensional.

2.4 Named Entity Recognition

Named Entity Recognition (NER) (Grishman and Sundheim, 1996) is the task of identifying and classifying entities mentioned in a text in order to allow future analysis or grouping. Many approaches have been taken over the last 30 years, from rule based systems, deep learning approaches and currently favouring approaches based on transformers. A widely used ontology used to classify entities is that developed by the Ontonotes project for their Ontonotes 5.0 release (Xue, 2012). ON5 entity types include:

PERSON	People, including fictional
NORP	Nationalities or religious or political groups
FACILITY	Buildings, airports, highways, bridges, etc.
ORGANIZATION	Companies, agencies, institutions, etc.

GPE	Countries, cities, states
etc.	

Table 1: Ontonotes 5.0 Entity Types

In this project spaCy is used which has evolved over the last 7 years into a widely respected industry toolkit for natural language processing, including NER. spaCy conforms to the ON5 entity labels. Our aim is of course to identify additional entity labels not covered by ON5 / spaCy.

2.5 Probability Distributions

The primary distribution assumed in both models is the Student's t-distribution, a variant of the Gaussian distribution designed for smaller sample sizes. The approach is to calculate the t-distribution for predictions on a sample batch for the encoder and the latent space, then find the KL divergence between those two distributions to use to generate a loss value for training. This should allow the model to converge on a latent space representation while also training an autoencoder for accurate reconstruction, as a form of self-supervised learning.

KL Divergence or relative entropy between two probability distributions is a measure of the relative likelihood that a given sample comes from the distribution *P(x)* vs all other *P*.

The Student's t-distribution, *P*, of the encoder output is calculated, as is the t-distribution, *Q*, of the latent space. The KL divergence $D_{KL}(P \| Q)$ is then calculated for the sample *x* as per

$$D_{KL}(P \| Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

(MacKay, Kay and Press, 2003)

The other loss function that can be applied to the latent representation is cross-entropy: a measure of the difference between two probability distributions. The cross-entropy of distributions *P* and *Q* is the number of bits needed to determine whether a sample is from *P* or from *Q,* and as such enables us to tell how closely the two distributions match.

For a discrete distribution such as ours, the cross entropy between *P* & *Q* is given by

$$H(P, Q) = -\sum_i p_i \log q_i$$

The difference between cross entropy and KL divergence is that KL divergence is the number of *additional* bits needed to represent the data drawn from *P* as opposed to if it were drawn from *Q*. The relationship between cross entropy and KL divergence is given by

$$D_{KL}(P \| Q) = H(Q, P) - H(P)$$

3 Design

3.1 Use Case

The application for the project is the identification of clusters of terms which are not recognised by the NER system but are semantically coherent and indicative of a potential new ontological category present in the data.

3.2 Overall Design

The system operates in three main phases, pre-training, training and analysis.

In pre-training, the system takes a large corpus of text from an appropriate domain, performs NER on the text, identifies noun chunks in the corpus and generates embeddings from the material. The key determiners here are i) the NER system used, ii) the embeddings mechanism used, and iii) the radius or the number of tokens before and after the noun chunk that will be incorporated in the embeddings.

In training, the neural network is trained on the pre-training embeddings, with labels supplied by the prior NER. The aim is to generate a neural network that can be used, directly or indirectly, to produce a set of clusters wherein the NER categories are well identified. Other unlabelled clusters are also identified without obvious NER categories. These are then candidates for ontology assignment.

3.2.1 The fundamental concept

By design, the number of clusters found will be greater than the number of spaCy entity classes used. There will therefore be a set of clusters with a spaCy class label,l and a different set of clusters of unknown class. This latter set is predicted to be more interesting to the ontology engineer as they might represent novel entity types.

3.3 Autoencoders

Both approaches used in this project are based on autoencoders. An autoencoder is an unsupervised neural network that regenerates its input as its output. The key feature of autoencoders is the bottleneck *B,* a layer in the system with dimensionality lower than the input. This requires the network to learn a lower dimensional representation of the data which is capable of being regenerated into the input. The set of layers *E* from the input to the bottleneck is called the encoder, and the layers *D* from the encoder to the reconstructed output are the decoder. Once the whole autoencoder has been trained, the encoder layers alone can be used to generate a lower-dimensional representation of an input which, given the ability to decode into a faithful representation of the original input, retains much of the semantic richness of that input.

As depicted below:

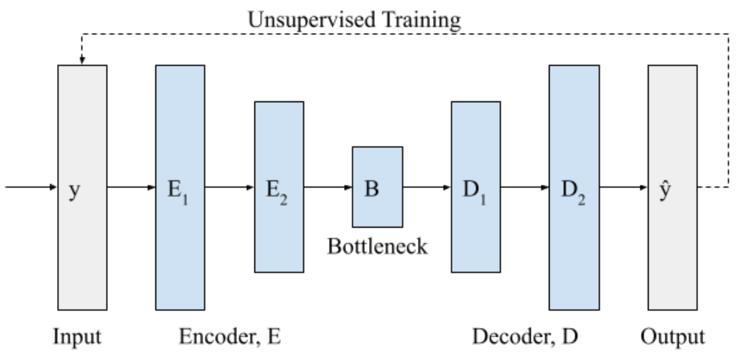


Figure 3.1. Autoencoder architecture

$$B = E(y)$$

$$\hat{y} = D(B)$$

$$\hat{y} = D(E(y))$$

3.4 Denoising autoencoders

To reduce the risk of overfitting in large autoencoders, Gaussian noise is added to the input, creating what is described as a denoising autoencoder. These networks are designed (Vincent *et al.*, 2008) to produce high-fidelity output despite the presence of noise in the input. Figure 3.2. Denoising autoencoder loss illustrates the training loss from the full model with and without noise. Note that the overall loss is higher when noise is added, which is how denoising reduces overfitting (Liang and Liu, 2015) by reducing the risk of memorisation of data.

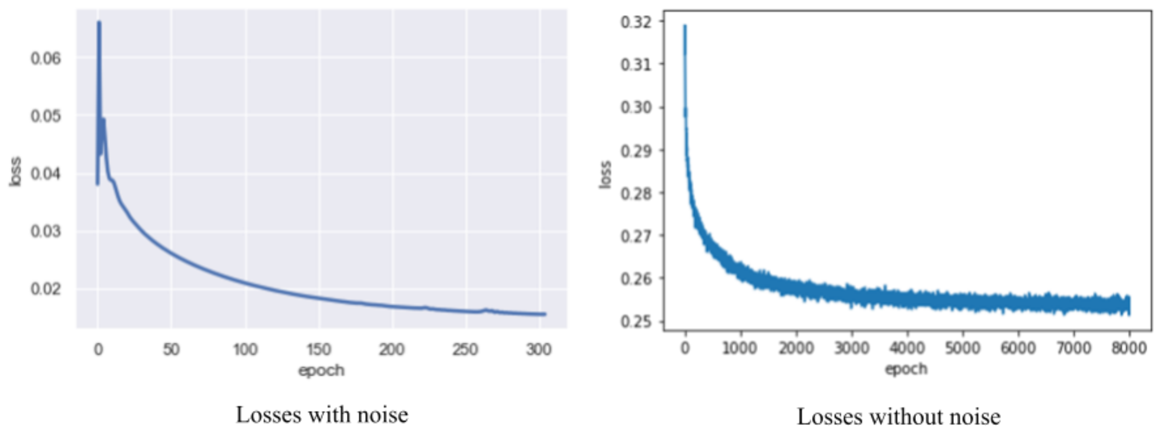
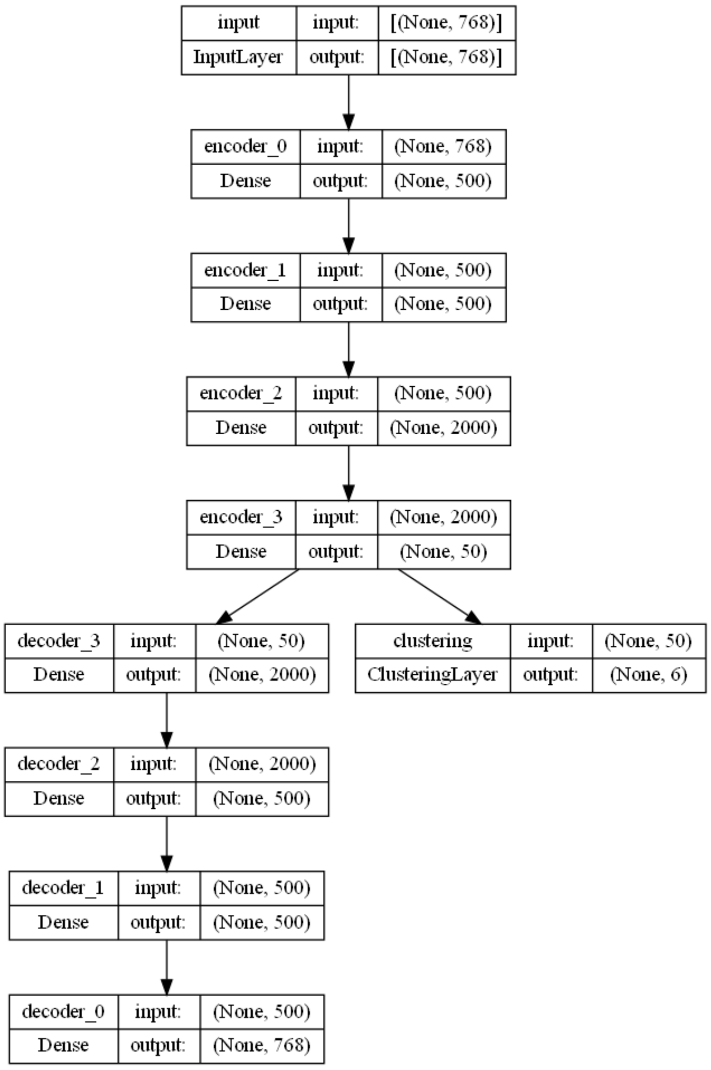


Figure 3.2. Denoising autoencoder loss

3.5 Deep Clustering

The deep clustering model, after Deep Embedded Clustering DEC(Xie, Girshick and Farhadi, 2015), pre-trains an autoencoder on the training dataset to derive an encoder, then uses a single-layer densely connected network with softmax activation to predict the cluster label for a sample. The cluster layer is initialised using the cluster centroids from a k-means clustering of the data. Our implementation uses the network shown in Figure 3.3, where the encoder feeds two different networks: the decoder while pre-training; and then the clustering layer for learning cluster assignments. This example is generating 6 clusters, as can be seen from the output shape of the clustering layer, but the number is configurable and was subject to optimisation.


input | input: | [(None, 768)]
InputLayer | output: | [(None, 768)]
encoder_0 | input: | (None, 768)
Dense | output: | (None, 500)
encoder_1 | input: | (None, 500)
Dense | output: | (None, 500)
encoder_2 | input: | (None, 500)
Dense | output: | (None, 2000)
encoder_3 | input: | (None, 2000)
Dense | output: | (None, 50)
decoder_3 | input: | (None, 50)
Dense | output: | (None, 2000)
clustering | input: | (None, 50)
ClusteringLayer | output: | (None, 6)
decoder_2 | input: | (None, 2000)
Dense | output: | (None, 500)
decoder_1 | input: | (None, 500)
Dense | output: | (None, 500)
decoder_0 | input: | (None, 500)
Dense | output: | (None, 768)
Figure 3.3: Deep Embedded Clustering

3.5.1 Training

The Deep Cluster network is trained in two stages. The autoencoder is trained unsupervised on the training corpus allowing it to learn to generate the reduced-dimensional representation E, by learning $X \rightarrow E \rightarrow X'$ followed by training the clustering layer on top of the encoder, allowing it to learn to cluster $X \rightarrow E \rightarrow \mu$ where μ is the cluster allocation. The clustering layer is pre-seeded with the cluster centroids as found by k-means across the training dataset.

The model then simultaneously learns the parameters θ of the non-linear mapping $f_\theta : X \rightarrow E$ and also the cluster centroids $\{\mu_i \in E\}_{i=1}^{k}$ for k clusters. This is trained as a single-headed network with the softmax of the clustering layer as the output – the decoder is not used in training after the initial seeding with the cluster centroids from k-means.

The loss function used is not the direct cluster mappings $X \rightarrow \mu$ rather, the following process is followed:

Maaten and Hinton, in their T-SNE paper on visualising multidimensional data (Van Maaten and Hinton, no date), use the Student's t-distribution with one degree of freedom to map distances into probabilities. The distribution is computationally efficient compared to using Gaussian mixtures, as well as ensuring good separation between cluster centroids. This aspect was important for clear visualisation of high dimensional clusters, but also suits our purpose well in increasing the cluster separation and thus reducing the likelihood of cluster overlap and false classification.

Algorithm 3 Training Deep Embedded Clustering

Input: Set of $(s, l)_i$ for sentence *s* and $l \in L^k$, the label for sentence *i*, from *k* unique labels

Output: Set of weights W for the trained model and *k*, the number of clusters.

```
Calculate the number of k unique entries in L.
For each sentence $s_i$
    Compute $e_i \in E \in R^{768}$, the distilBERT embeddings for $s_i$

Calculate the clusters $C^k$ from E using k-means given k clusters
Initialise weights $W$ as $C^k$
Calculate n batches $b_j$ of size $|b|$ from E where $n = \frac{|E|}{|b|}$
    $b_j = e : e \in E_{nj}^{n(j+1)}$
For each batch b
    Calculate $D^{\sigma}$, softmax distribution of output from batch
        $D_i^{\sigma} = \sigma(z)_i = \frac{e^{z_i}}{\Sigma e^{z}}$
    Calculate $\delta_i^{\mu}$, the cluster centroid distance for each point in b
    Calculate $D^t$, student t-distribution of $\delta^{\mu}$ with $\alpha = 1$ DOF
        $D_i^t = \frac{\sum_j \left(1+\delta_j^{\mu 2}\right)}{1+\delta_i^{\mu 2}}$
    Calculate $D^{KL}$, the KL Divergence between $D^t$ and $D^{\sigma}$
        $D\hat{}KL\ (D\hat{}t\ ||D^{\sigma}) = \sum_i \sum_j D_{ij}^t \log \frac{D_{ij}^t}{D_{ij}^{\sigma}}$
    Update W by stochastic gradient descent with loss = $D^{KL}$
Return W, k
```

Once the network has been trained it is then used for inference, predicting cluster assignments from the embeddings of each sentence.

Inference with the trained model is now a forward pass of the embedded chunks:

```
Algorithm 4 Inference with Deep Embedded Clustering
Input: Set of sentences $s_i \in S$
Output: Set of tuples $\left(s_c, l_c\right)$ for sentence $s_c$, where
        $l \in \mu^k$ for μ the set of k cluster centroids
Initialise results $r = \{ \}$
For each sentence $s_i$ in S
    Compute , $s_i^{emb} \in \mathbb{R}^{768}$ the distilBERT embeddings for $s_i$
    Compute x, a forward pass of network f with softmax activation
        $f: e \in R^{768} \rightarrow \mu$
        $x = f\left(s_i^{emb}\right)$
    Append $(s_i^{emb}, x_i)$ to r
Return r
```

3.6 Joint Learning of the Latent Space

Dahal (Dahal, 2018) describes a model with a pre-trained autoencoder as per Xie et al. (Xie, Girshick and Farhadi, 2015), then adds a network to learn the latent space representation ***Z*** of the data. The loss function is then the KL divergence between ***X'*** and ***Z: X'*** the probability distribution of the decoder**,** and ***Z*** the probability distribution of the latent representation. Once the network is trained, the encoder output ***E*** can be used for clustering via a separate inference pass.

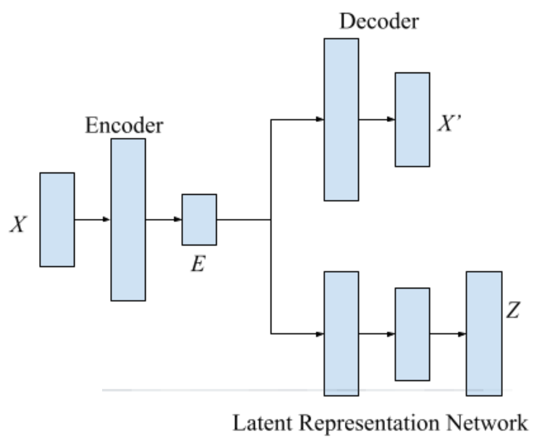


Figure 3.4: Learning Latent Representation

As shown in Figure 3.5, our implementation of a latent representation network replaces the single clustering layer of Figure 3.3 with a deep network to learn the latent space representation. The output of layer `encoder_3` is ***E***, the encoder embeddings; the output of `decoder_0` is ***X'***, the reconstructed embeddings; and the output of `latent_out` is ***Z***, the latent space representation.

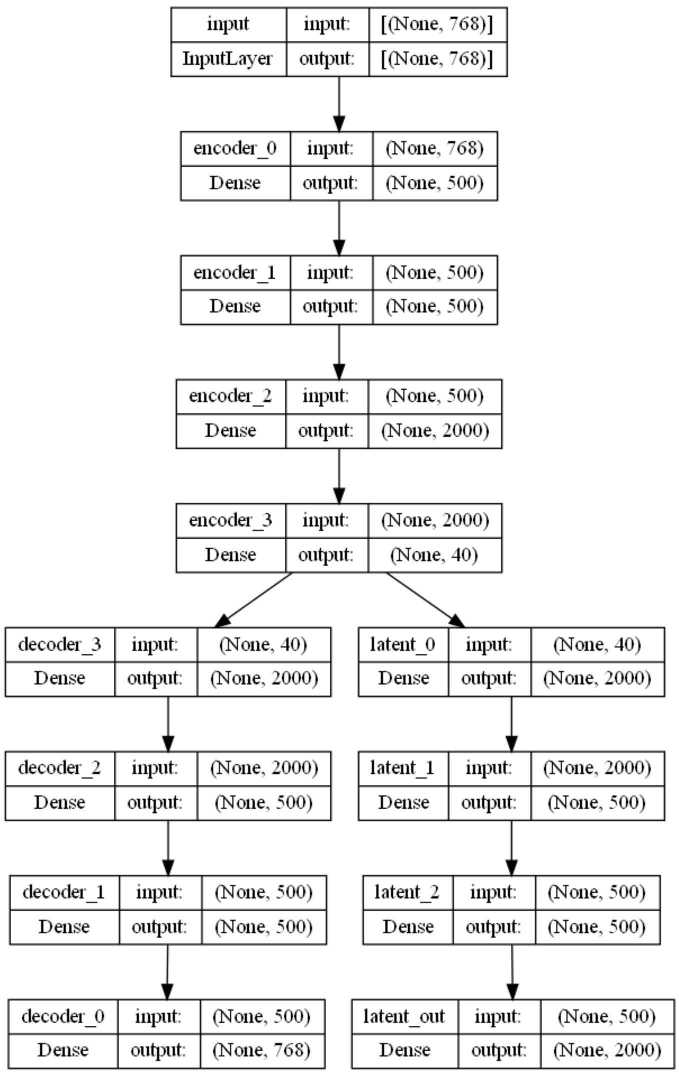

input | InputLayer | input: | output: | [(None, 768)] | [(None, 768)]
encoder_0 | Dense | input: | output: | (None, 768) | (None, 500)
encoder_1 | Dense | input: | output: | (None, 500) | (None, 500)
encoder_2 | Dense | input: | output: | (None, 500) | (None, 2000)
encoder_3 | Dense | input: | output: | (None, 2000) | (None, 40)
decoder_3 | Dense | input: | output: | (None, 40) | (None, 2000)
decoder_2 | Dense | input: | output: | (None, 2000) | (None, 500)
decoder_1 | Dense | input: | output: | (None, 500) | (None, 500)
decoder_0 | Dense | input: | output: | (None, 500) | (None, 768)
latent_0 | Dense | input: | output: | (None, 40) | (None, 2000)
latent_1 | Dense | input: | output: | (None, 2000) | (None, 500)
latent_2 | Dense | input: | output: | (None, 500) | (None, 500)
latent_out | Dense | input: | output: | (None, 500) | (None, 2000)

Figure 3.5: Latent representation model

3.6.1 Training

This model pre-trains the autoencoder and then discards the decoder for the final training.

The network is trained with a loss function that calculates the KL-divergence between two Student's t-distributions: that of the Encoder output *E*; and that of the latent representation *Z*.

```
Algorithm 5 Training the Latent Representation Network
Input: Set of $(s, l)_i$ for sentence s and $l \in L^k$, the label for sentence i, from k unique labels
Output: Set of weights W for the trained model and k, the number of clusters.
Calculate the number of k unique entries in L.
For each sentence $s_i$
    Compute transformer embeddings $T : \{t_i\} \in R^{768}$, the distilBERT embeddings for $s_i$
    Calculate n batches $b_j$ of size $|b|$ from T where $n = \frac{|T|}{|b|}$
        $b_j = t : t \in T_{nj}^{n(j+1)}$
For each batch b
    Calculate by forward pass the encoded representation E and the latent representation Z
    Calculate $\delta_{ij}^{Z}$, $\delta_{ij}^{E}$, pairwise distances in Z and E respectively
    Calculate $q_{ij}$, the t-distribution of $\delta^{Z}$ with $\alpha = 1$ DOF
        $q_{ij} = \frac{\sum_{k \neq l} \left(1 + \delta_{kl}^{E^2}\right)}{1 + \delta_{ij}^{Z^2}}$
    Calculate $p_{ij}$, the t-distribution of $\delta^{E}$
        $p_{ij} = \frac{(1 + \delta_{ij}^{E^2} / \alpha)^{-\frac{\alpha+1}{2}}}{\sum_{k \neq l} (1 + \delta_{kl}^{E^2} / \alpha)^{-\frac{\alpha+1}{2}}}$
    Calculate $D^{KL}$, the KL Divergence between $p_{ij}$ and $q_{ij}$
        $D^{KL}(p_{ij} || q_{ij}) = \sum_i \sum_j p_{ij} \log q_{ij}$
    Update W by stochastic gradient descent using $D^{KL}$ for the loss

Once network training is complete:
Calculate cluster assignments $s_i^{\mu}$ for noun phrase $s_i$ in cluster $\mu \in \{1..k\}$ for k clusters
        using the chosen clustering algorithm

Return W, k
```

Inference for the latent representation model is two-stage, first computing the latent representation of each sample, then predicting cluster assignments for each sample using the previously fitted clustering algorithm.

```
**Algorithm 6** Inference with Latent Representation
**Input**: Set of noun phrases $s_i \in S$
**Output**: Set of tuples $(s_c, \mu_c)$ for sentence $s_c$, where $\mu \in \{1..k\}$ for k clusters
Initialise results $r = \{\}$
**For** each sentence $s_i$ in $S$
    Compute $s_i^{emb} \in R^{"768"}$, the distilBERT embeddings for $s_i$
    Compute $x$, a forward pass of network $f$, for d-dimensional latent representation:
        $f: e \in R^{768} \rightarrow z\ R^{d}$
        $z = f\left(s_i^{emb}\right)$
    Compute cluster predictions $s_i^{\mu}$ for noun phrase $s_i$ in cluster $\mu \in \{1..k\}$ for k clusters
        using the previously trained clustering algorithm.
**Return** $S^{\mu}$, cluster assignments for each noun phrase
```

The four clustering algorithms considered for this project were:

- K-means, as used by(Xie, Girshick and Farhadi, 2015) and (Dahal, 2018)
- Gaussian Mixture Models (GMM)
- Agglomerative clustering
- OPTICS

Although the choice was made empirically, a key part of each model is the distance metric. Manhattan was used for k-means, GMM and optics. For Agglomerative clustering, Manhattan metrics are incompatible with ward linkage so both Euclidean distance with Ward linkage and single linkage were tried. Single linkage performed better, likely due to the high dimensionality of the data.

3.7 Data generation

The models require labelled data for the training phase, then unlabelled data for inference. The data is generated as follows:

Text samples are obtained from the conll2003 (Tjong Kim Sang and De Meulder, 2003) dataset in the Hugging Face community (The HF Datasets community, no date). The supplied NER labels are not required and are discarded as they will be reconstructed using spaCy, allowing for additional training sets to be used later, containing text not part of the conll dataset.

The datasets, in the form of sentences, are processed using spaCy (spaCy, no date) to identify for each sentence

i. the tokens;
ii. Noun chunks in the text
iii. Any entities recognised by spaCy, along with their ontology class labels.

Identified entities were then matched with the noun chunks based upon token span overlap, in order to assign class labels to those noun chunks containing an entity, or <*UNKNOWN*> to all other noun chunks

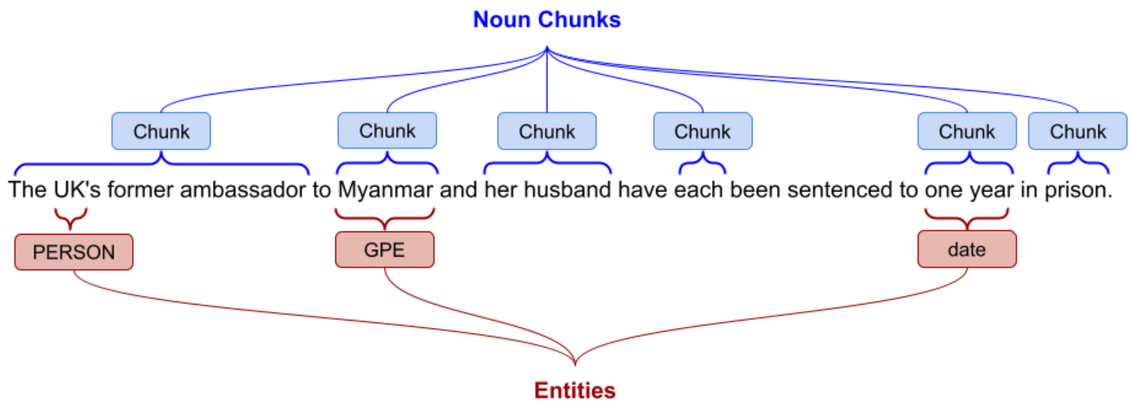
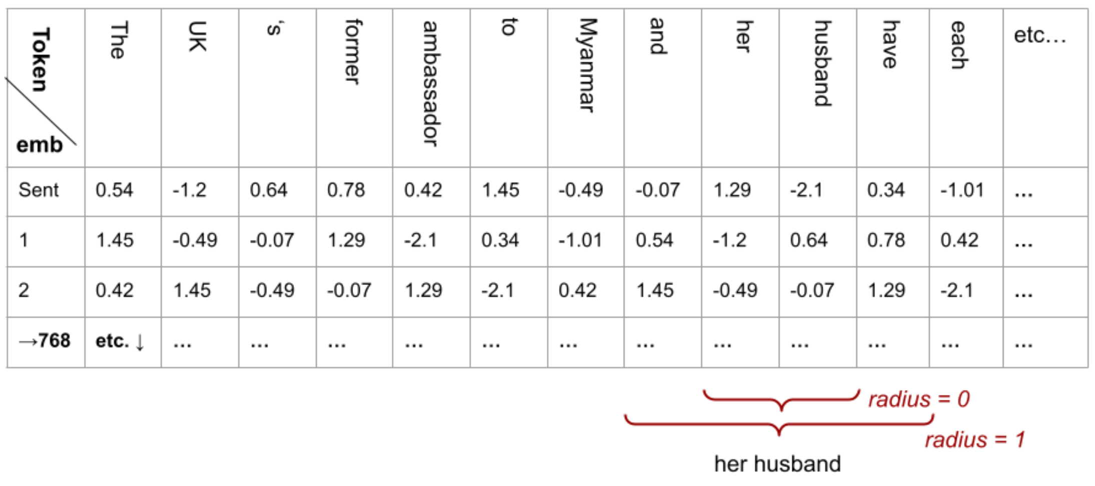

Noun Chunks
Chunk
Chunk
Chunk
Chunk
Chunk
Chunk
The UK's former ambassador to Myanmar and her husband have each been sentenced to one year in prison.
PERSON
GPE
date
Entities

Figure 3.6: Overlapping of entities and noun chunks

This generates a set of noun chunks as shown in Figure 3.6, some of which have an associated spaCy entity label. If a chunk contains more than one entity label, each entity will lead to a new sample differing only in the class label.

The sentences are also processed through a distilBERT transformer, then feature extraction is applied to derive the 768-dimensional BERT embeddings for each token. The Hugging Face BERT implementation offers a feature extraction pipeline for BERT-based models. This pipeline affords us the 768-dimensional output of the final layer, upon which fine-tuning would normally be executed. In this project, those embeddings are taken as the input to our models. It should be noted that the recommended approach to generating a full sentence embedding is to take embedding zero - the embedding of the initial <CLS> token in the input stream (Reimers and Gurevych, 2019), however, our approach is to generate embeddings for the chosen noun chunk *in-context*, by averaging the embeddings of the tokens in the chunk, along with *radius* additional tokens before and after the chunk. This was empirically optimised to a radius of 6. This approach was compared with treating each noun chunk as a sentence in its own right, then generating distilBERT embeddings for that sentence. Those sentence embeddings were found to be less accurate, likely due to loss of contextual information the sentence provides, as well as computationally expensive - each conll sentence needed several passes through distilBERT to generate a novel embedding, as opposed to a single pass per sentence for the chosen approach. On the machine used (see 4.4) distilBERT embeddings are computed at an average rate of 8 per second.

Training samples are then generated using the embeddings derived, averaging the embeddings for the tokens representing the chunk, with an optional number, the radius, of additional tokens before and after the chunk for context as per Algorithm 1.



Token / **emb**	The	UK	's	former	ambassador	to	Myanmar	and	her	husband	have	each	etc…
Sent	0.54	-1.2	0.64	0.78	0.42	1.45	-0.49	-0.07	1.29	-2.1	0.34	-1.01	…
1	1.45	-0.49	-0.07	1.29	-2.1	0.34	-1.01	0.54	-1.2	0.64	0.78	0.42	…
2	0.42	1.45	-0.49	-0.07	1.29	-2.1	0.42	1.45	-0.49	-0.07	1.29	-2.1	…
→768	**etc. ↓**	…	…	…	…	…	…	…	…	…	…	…	…

radius = 0
radius = 1
her husband

Table 2. Entity & radius mapped to embeddings for a noun chunk

```
**Algorithm 7** Data Preparation
**Input**: Set of sentence S = $(w_1, w_2, \dots w_n) \in W$ for vocabulary $W$
**Output**: Set of tuples with the words in a chunk, the embeddings, and any labels:
$T_c = (w_c, emb_c, l_c)$ for chunk $c$, where $l \in N\{0 \dots N_{labels}\}$, $emb \in R^{768}$, $w \in W$
**For** each sentence $s_i$ in $S$
    Compute the BERT embeddings $S_{emb} \in \mathbb{R}^{n \times 768}$ for $s_i$ for $n$ tokens in $s_i$
    Compute the labels $L_i = \{l_{i0}, l_{i1}, \dots l_{ik}\}$ for all entities $e$ in sentence $i$.
    Compute $C_{s_i}$ the set of noun chunks $c$ in $s_i$
    **For** each noun chunk $c$ in $C_{s_i}$
        Compute $w_{start}$, $w_{end}$ for the start and end token numbers in $c$
        Compute $emb_c$ the embeddings for $c$ in $S_i$ given radius $r$ as the average of the chunk span extended by $r$ tokens
            $emb_c = \frac{1}{n} \sum_{i=w_{start}-r}^{i=w_{end}+r} S_i$
        Add $T_c = (w_c, emb_c, l_c)$ to *result*
**Return** *result*
```

The system now has a list of samples, each with a word chunk, the embeddings of that chunk and a label (which may be <UNKNOWN>), the final training data:

Chunk	Lab	1	2	3	4	5	6	7	8	9	10	11	12	...768
UK's former...	PER	0.54	-1.2	0.64	0.78	0.42	1.45	-0.49	-0.07	1.29	-2.1	0.34	-1.01	...
Myanmar	GPE	-0.49	-0.7	1.29	-2.1	0.34	-1.01	0.54	-1.2	0.64	0.78	0.42	1.45	...
Her husband	UNK	-2.1	0.34	-1.01	0.54	-1.2	0.64	-0.49	-0.7	1.29	-2.1	0.34	-1.01	...

Table 3. Training Data Structure

3.8 Data Balancing

The distribution of class labels is not uniform in the training data prepared as above.

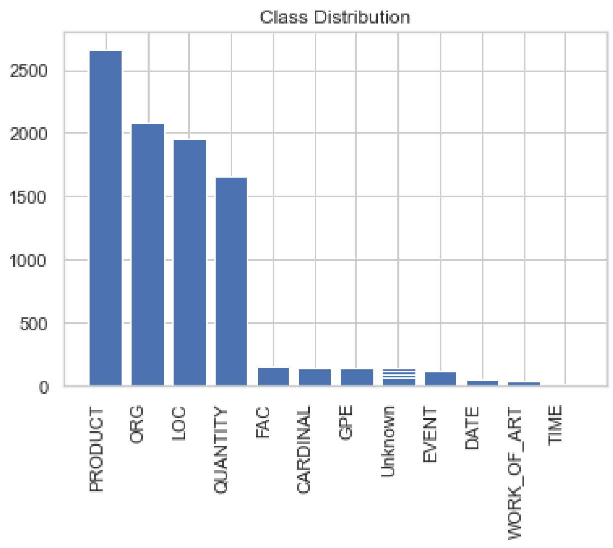


Figure 3.7: Imbalanced entity classes

Imbalanced data presents a problem for deep networks (Koziarski, Woźniak and Krawczyk, 2020), particularly in the case of multi-class classification (Wang and Yao, 2012), leading to a tendency to favour the larger classes. As a trivial example, if 99% of samples say "Yes", and 1% say "No", then a model can achieve 99% accuracy by always predicting "Yes" while failing to ever predict a "No".

Our approach in this project was to improve the representation of the less frequent classes by naïve random oversampling of the data. Although more sophisticated techniques are available, such as the Synthetic Minority Oversampling Technique - SMOTE (Chawla *et al.*, 2002) which generates new samples by interpolation, the effect of this data manipulation on a high-dimensional embedding is unclear. Naive random oversampling involves the generation of the distribution table for the sample classes, then generating a set of samples by randomly choosing entries from each class inversely to the proportion in the distribution, resulting in a similar number of samples for each class in the final sample set, although as this is performed *with replacement* there may be multiple copies of samples from smaller classes.

```
Algorithm 8 Naïve Random Oversampling
Input: Set of labelled samples, $S$: $s_i = (emb_i, c_i)$ with label $c \in C$
Output: Set of labelled samples, $\hat{S}$: $\hat{s}_i = (emb_i, c_i)$ with label $c \in C$
Compute the label distribution D in S give a set of class labels C
    $D = \lceil \frac{|s_i^l = c_j|}{|S|} \rceil \; for\ c_j\ in\ C$
For each class $c_j$ in $C$
    Compute class sample size $n_j = \frac{1}{(D)}$
    Compute sample $s^c$ of $c_j$ by random selection of $n_j$ samples $s \in \{[s^c = c_j]\}$
Return result
```

Our project implements this using the imbalanced-learn implementation (Lemaître, Nogueira and Aridas, 2017).

4 Implementation

4.1 Software Architecture

Initial development was undertaken in Jupyter Notebooks, to provide a history of decisions taken and to allow easy experimentation. Eventually, the models stabilised and the architecture was organised hierarchically as follows:

At the top level, there are Test and Source (src) folders.

In src most processing has evolved from the Jupyter Notebooks to their own files and imported them into the notebooks when needed. This allows pytest to test each module and reduces the risk of multiple versions of a function existing in a notebook.

The final major component is the PDF processing module. This allows users to process a collection of PDFs through the tool in order to generate proposed new clusters from the data, specifically for the knowledge engineering task set in the use case. This is covered in

Key modules and the main routines are:

4.1.1 data.py

This module coordinates the preparation and availability of training and test data.

Loads the data from prepared datasets stored locally, then generates embeddings, noun chunks and entity labels if not prepared, and caches these locally for future use.

Resample the data if required - see 3.8 above

Generates test and train splits. Training splits include all entity chunks as found by spaCy. For test splits, also included were chunks labelled “unknown”, i.e. they did not contain an entity recognised by spaCy. The intention is to train the network to cluster on known entities, but then to evaluate the model against unlabelled test data containing entities but also not containing known entities, thereby allowing clusters to form relating to semantic classes not contained in the spaCy ontology.

Test/train split is generated as follows

- Drop all rows with missing values using Pandas `DataFrame.dropna()`
- Randomly sample the required rows using pandas `DataFrame.sample()`
- Split the dataset into two, the test and train datasets.
- Create a *y* dataset for each, with only the label for each row
- Drop all columns from the *X* datasets apart from the embeddings
- Calculate the data balance per class using `numpy.unique()`
- Oversample the training dataset data using imbalanced-learn’s `RandomOverSampler()`

The method returns data containing the correct entities for the training or test run being undertaken.

4.1.2 deep_latent.py

This is the main module for the latent representation network, inspired by and partly built upon Dahal's original implementation (Dahal, 2019) of his paper (Dahal, 2018).

The main class is `DeepLatentCluster`, which has many roles. Firstly, to define the model using a configuration dictionary. This contains a set of default values, but train or test runs may choose to vary these. For example, for config

```
self.config = {
   ...
   "output_fn": 'sigmoid',
   "noise_factor": 0.0,
   "loss": k.losses.binary_crossentropy,
   "train_size": 10000,
   "num_clusters": 25,
   "cluster": "GMM",
   … etc.
```

one might then define for a specific training run

```
run_config = {
   "noise_factor": 0.25,
   "train_size": 10000,
   "num_clusters": 25,
   "cluster": "Kmeans",
}
```

The training routines in the class will then execute

```
current_config = {**self.config, **run_config}
```

to merge the two, overwriting default values.

This module also builds the various forms of the model - encoder, decoder, autoencoder, latent representation network, and the full model. The model architecture is shown in figure 3.5 above.

The class also contains methods to train the model. While standard Keras models can be trained by compilation with appropriate loss functions and then executing the `fit` method, this was not appropriate for this design given the complex loss calculation. A custom model training loop was therefore required. Standard Keras loss functions need to satisfy the signature `loss_fn(y, y_pred)` which is appropriate when the model has a single head, or in dictionary form for multiple heads with a different loss function of the same form required for each head, e.g.

```
losses = { "a_output": "mse", "b_output": my_loss_fn }
```

The model needed to compute two loss functions:

- For the autoencoder in both pre-training and in full model training, the mean squared error is used to compare the original input embeddings x with the reconstructed embedding $\hat{x}$:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left(x_i - \hat{x}_i \right)^2$$

```
tf.reduce_mean(tf.square(tf.subtract(x, x_pred)))
```

- For the latent representation, the loss is calculated as per algorithm 5 above, using the KL divergence of the two Student's distributions for the encoder output, and the latent network output. The probability distributions are calculated thus:

```
def make_q(z, batch_size, alpha):
    sqd_dist_mat = np.float32(
             pairwise.pairwise_distances(z, metric='sqeuclidean'))
    q = tf.pow((1 + sqd_dist_mat/alpha), -(alpha+1)/2)
    q = tf.linalg.set_diag(q, tf.zeros(shape=[batch_size]))
    q = q / tf.reduce_sum(q, axis=0, keepdims=True)
    q = tf.clip_by_value(q, 1e-10, 1.0)
```

Then the latent loss is found for $\hat{y}$ and z_{enc} the encoder output, here with KL divergence

```
p = make_q(y_pred, self.batch_size, alpha=self.config['alpha1'])
q = make_q(z_enc, self.batch_size, alpha=self.config['alpha2'])
return tf.reduce_sum(-(tf.multiply(p, tf.math.log(tf.divide(p,q)))))
```

Batches were used for the training, with batch size being subject to hyperparameter optimisation see Appendix A.

The encoder header is not an output of the model, so it is necessary to create a custom training loop, evaluate per-batch the loss functions above, and store the training variables in the Gradient Tape for application of the gradients per variable using the Adam optimizer. The `train` function of the class runs the epochs and iterations within epochs, calling the custom `train_step` function:

```
def train_step(self, x, y):
    # A single training step
    with tf.GradientTape() as tape:
        dec, lat = self.model(x, training=True)
        loss_l = self.latent_loss(self.encoder(x))(y, lat)
        loss_r = self.reconstr_loss(x, dec)
        loss_value = self.config['latent_weight'] * loss_l +\
                     self.config['reconstr_weight'] * loss_r

    grads = tape.gradient(loss_value, self.model.trainable_weights)
    optimizer = self.config['opt']
    optimizer.apply_gradients(zip(grads, self.model.trainable_weights))
    self.train_acc_metric.update_state(y, lat)
    return loss_value
```

Two main classes of optimisers were used in the models, both in the autoencoder and in the full model training. Stochastic Gradient Descent (SGD), and Adam. Adam (Kingma and Ba, 2014) was found to give generally better results, likely due to its adaptive nature - Adam maintains a learning rate which is both adaptive (it is modified for optimal learning over time) and also has a different rate per-parameter rather than a single rate. The combination of these two features is particularly useful for sparse data, such as the high-dimensional embeddings used in this project.

4.1.3 deep_cluster.py

This is the main module for the Deep Cluster model, inspired by and building upon Guo's implementation (Guo, no date) of Xie et al.'s DEC (Xie, Girshick and Farhadi, 2015).

The main class is DeepCluster, which, similar to DeepLatentCluster, manages the creation and training of the model for the DEC implementation. The model is shown in Figure 3.3 above, consisting of a paired encoder and decoder, making up the autoencoder, along with a clustering layer sitting as a second head atop the encoder.

The cluster layer, `ClusterLayer`, is a single dense layer with input dimension equal to the bottleneck in the autoencoder, and output dimension equal to the number of clusters to be found, configured in the model initialisation. The weights of the cluster layer are initialised, as per (Xie, Girshick and Farhadi, 2015), to the cluster centroids found by clustering a sample of the training data using the chosen clustering algorithm.

Training the model happens in two stages. Firstly the autoencoder is pre-trained on the full training dataset, with early stopping set as follows

```
EarlyStopping(monitor='loss', patience=5, verbose=1, min_delta=0.0003)
```

or, if the loss value has not deviated more than 0.0003 from its current value for the last 5 epochs, terminate. Loss for the autoencoder pretraining is MSE as described in 4.1.2 deep_latent.py above.

The main training is then executed: The model performs epochs of iterations, each iteration predicting, in batches, the clusters as the softmax of the clustering layer outputs, then calculating the loss from the KL divergence between the softmax cluster assignments q_i and the auxiliary distribution P, given by

$$p_{ic} = \frac{q_{ic}^2 / f_c}{\sum_{c'} q_{ic'}^2 / f_{c'}}$$ (Xie, Girshick and Farhadi, 2015)

for cluster c and sample i. Processing stops after the configured number of iterations, or when training converges, i.e. few cluster prediction changes between iterations:

```
# check stop criterion
delta_label = np.sum(y_pred != self.y_pred_last) / y_pred.shape[0]
if iteration > 0 and delta_label < tolerance:
    break
self.y_pred_last = y_pred
```

Three loss values are calculated, then the weighted sum is calculated as the overall iteration loss. The three losses are

- The KL divergence between P and Q as above.
- The MSE between autoencoder input x and output $\hat{x}$,
- The cluster accuracy loss between the true cluster and predicted cluster, given the true cluster assigned to a sample by spaCy for known entity types.

4.1.4 extract_bert_features.py

The component maintains a singleton transformer pipe, due to the Hugging Face distilBERT implementation having an 8s load time. The pipe operates thus:

```
text → tokenizer → distilBERT → feature extraction
```

Extracted bert features , i.e. the embeddings $\in \mathbb{R}^{768}$ are returned.

4.1.5 grid_search.py

This utility iterates over all combinations of parameters for model evaluation, given a configuration. It is based on the itertool `product` function to generate combinations:

```
combos = [
        {params[i]: v for (i,v) in enumerate(x) }\
                                        for x in it.product(*config.values())]
```

In the main notebooks, the config for an experiment is a simple dictionary that maps configurable parameters to test values which enables a grid search of hyperparameters to be executed by the `grid_search` method. For example:

```
config = {
    'train_size': [0],
    "radius": [6],
    'latent_weight': [0.9, 1.0, 1.1],
    'reconstr_weight': [0.65, 0.75, 0.85],
    'head': ['enc'],
    'cluster': ['OPTICS'],
    'noise_factor': [0.5]
}
grid_search(config, do_run)
```

will execute tests of method `do_run` against 9 configurations - 3 latent loss weights × 3 reconstruction loss weights, with all other parameters being a single element array and therefore the single value used.

4.1.6 cluster_metrics.py

This module is responsible for calculating and displaying cluster metrics.

The method `rearrange_clusters` will reorder the predicted classes for the clusters for analysis as per Algorithm 9 in 5.6 below, ensuring that, for example, the predicted class with the most PERSON predictions in it is given the prediction label PERSON. Importantly, this does not change the predictions, but enables better scoring and lets a useful confusion matrix be plotted.

The core clustering method, `do_clustering`, is in this module, applying the chosen clustering algorithm (K-Means, GMM, OPTICS or agglomerative clustering) to the provided sample.

`do_evaluation` is a utility function that takes a sample with 4 columns:

Column	Description
text	The full embedded text, i.e. the noun chunk +/- *radius* tokens
y_true	The chunk label from spaCy NER
y_pred	The cluster label from our model
shorts	The noun chunk text with *radius* extra tokens

Table 4. Sample structure for do_evaluation method

and then evaluates the clusters as above and saves the results as index.html with embedded images for word clouds for the larger clusters, plus the confusion matrix generated.

4.1.7 process_files.py

This file contains the routines to process a folder, find PDFs or other files within it, extract the text from those files, split the text on sentence boundaries and create a file for training containing one sentence per line. Document text extraction is done using the Apache Tika library via `tika-python` (Mattmann, no date).

This file can also be used from the command line, with full details provided in the `readme.md` file in the associated repo. Simple usage would be

```
> python process_files.py -a deep_latent -f my-file.txt
```

instructing the script to process files in ./pdfs and create an output folder ./results/my-file.txt, utilising the deep latent cluster model. For full details, consult the readme, or execute the script with the `-h` argument.

Finally the main notebooks

4.1.8 latent_represention.ipynb

For training and evaluation of the Latent Representation Network

4.1.9 deep_clustering.ipynb

For training and evaluation of the Deep Embedded Clustering Model

4.1.10 pdfs_notebook.ipynb

Demonstrating the use of process_files.py to process a set of PDFs, extracting the text and then preparing them for training or analysis.

4.2 Testing

4.2.1 Unit Tests

Unit testing of the software itself was performed using pytest, with a variety of specific test methods written for major system components, including

- the conll data ingestion (data_conll_test.py)
    - the correct data has been read
- the data ingestion classes (data_test.py)
    - the data read is of the expected shape and contents
- the DeepCluster class (DeepCluster_test.py)
    - data loading
    - model construction
    - clustering
    - prediction
- extracting BERT embeddings (extract_bert_features_test.py)
    - expected number and shape of embeddings
- metrics calculations (metrics_test.py)
    - acc function results
- spacy NER & data labelling (spacy_NER.py)
    - semantic components identified as expected

For example,

```
def test_predict(dc):
    # cluster prediction accuracy ok
    dc.y = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    dc.y_pred = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    assert dc.cluster_pred_acc() == 1.0
    dc.y = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    assert dc.cluster_pred_acc() == 0.9
```

This code, testing the `cluster_pred_acc()` method of the `DeepCluster` class, asserts that given a list of true labels and an identical list of predicted labels the accuracy is 1.0, whereas if 1 in 10 labels in the prediction list are changed to be erroneous, the accuracy will fall to 0.9. This is performed by creating a `DeepCluster` object, `dc`, then directly setting the internal values of that object y, the true values from spaCy labels. and y_pred, the predicted values of y derived by passing X through the model. The `cluster_pred_acc()` is then called given these values and the correct responses asserted.

Pytest is integrated into VS Code and executes all tests on demand.

4.2.2 Functional tests

Integrated functional testing was carried out ad-hoc in the interactive Jupyter Notebook environment, primarily concerned with identifying run-time errors and logical errors. As part of this process, python type hinting (van Rossum, Lehtosalo and Langa, 2014) was usedto improve static analysis by the linters, and to reduce the errors commonly found in Python where the type of an object is poorly disciplined and unexpected results occur depending on the execution path. Python data classes (Smith, 2017) were also found to be useful, further tightening the ability of the linters to detect logical errors of this kind. For example, the code describing a cluster:

```
clusters = {}
entry = {
    'freqs': freqs,
    'freqs_unknown': freqs_unknown,
    'class_freqs': class_freqs,
    'frac': frac,
    'n': len(cluster),
    'label': prob_lbl,
    'entity_id': prob_ent,
    'clus_no': clus_no,
}
clusters.append(entry)
```

The problem with this code is that the structure of the record is not enforced, nor are the types of the properties within it. This led to run-time problems being identified during the execution of the tests, rather than being identified in advance.

This was replaced with code utilising data classes and class initialisers:

```
@dataclass
class Cluster:
    """
    Describes a single cluster of entities.
    """
    freqs: dict
    freqs_unknown: dict
    class_freqs: dict
    frac: float
    n: int
    label: str
    entity_id: int
    clus_no: int
    name: str

ClusterList = dict[str: Cluster]

# execution in a later method
entry = Cluster(
           freqs=freqs,
           freqs_unknown=freqs_unknown,
           class_freqs=class_freqs,
           frac=frac,
           n=len(cluster),
           label=prob_lbl,
           entity_id=prob_ent,
           clus_no=clus_no,
           name=prob_lbl)
```

4.3 Hyperparameter Optimisation

Various hyperparameters were trialled using grid search as described in 4.1.5 grid_search.py above.

The list of hyperparameters experimented with is at Appendix A below, but important examples were:

- the size and number of layers in each model, for which it was found the parameters suggested by Xie et al. and Dahal were optimal
- the size of the autoencoder bottleneck, which was optimised to 40
- the number of entity classes (Top-n) to be included from spaCy for data preparation, optimised to top 15
- the number of clusters to be found, optimised to 25, allowing for the 15 spaCy classes plus 10 novel classes.

4.4 Tools used

A variety of tools and third party components were used in this project, including

- Visual Studio Code as the IDE for development and debugging
- Jupyter Notebooks hosted in VS Code for model development and iteration.

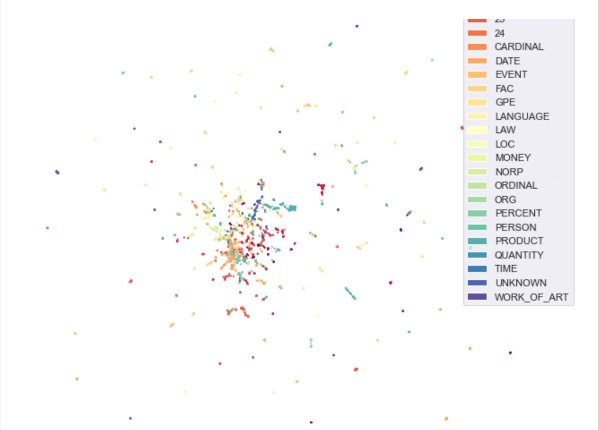
- Python 3.9, for all code. 3.9 is the highest version that CUDA supported for tensorflow / Windows / RTX 3090 at the time the project was started
- A modern home PC, optimised for ML tasks
    - Nvidia RTX 3090 GPU, for model execution and training
    - 64GB DDR5 RAM
    - M.2 SSD storage
    - Intel i9-12900K CPU
- Tensorflow 2.9 / Keras for most model code
- Hugging Face Transformers 4.20 for the distilBERT model and pipelines
- spaCy 3.4 for the entity recognition and noun chunk extraction
- Scikit-learn 1.1 for various processing libraries in the notebooks
- UMAP 0.5 for cluster visualisation

Parts of the code were built on top of the work of (Xie, Girshick and Farhadi, 2015) and (Dahal, 2018).

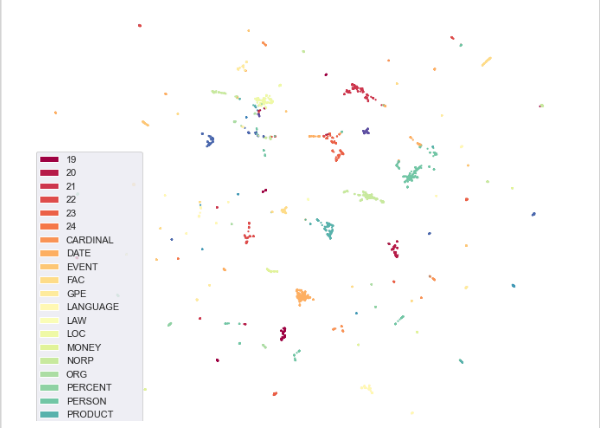
5 Evaluation & Results

5.1 Cluster Separation

The choice of the Student T distribution for the latent space was driven largely by the fact that it will, as applied here, increase the cluster separation.



UMAP visualisation of raw embeddings



UMAP visualisation of model output

This is demonstrated in the images above, both from the same sample. The left shows the k-means clustering applied to a sample of embeddings. The right image shows the same sample after prediction through the DeepClustering model. Cluster separation is markedly more apparent in the second image.

5.6 Rearranged F1 Measure

There are a variety of techniques for comparing clustering algorithms with each other, such as *circle segments(Achtert et al., 2012)*, however our aim is not to compare two clustering techniques but to measure the value to the use case of the overall model.

Our approach is to assess whether or not an individual cluster assignment is a positive result (i.e. correctly assigned) as follows. The author proposes the “Rearranged F1” metric to build upon the standard F1 measure. In summary, Rearranged F1 (RF1) is the averaged F1 score across all clusters found after rearranging the cluster labels for best fit. This is necessary as there is no direct mapping between the cluster IDs used for training, i.e. the spaCy entity labels for known classes, and the cluster numbers found by the clustering algorithm itself. There is no guaranteed ordering and, for example, “training cluster 7” derived from entity class 7 is not likely to be comparable with “found cluster 7” from K-means. To repeat the important point mentioned in 3.2.1 The fundamental concept above, by design, the number of clusters found will be greater than the number of spaCy entity classes being used, so one set of clusters each with a spaCy class label is found, and a further set of clusters of unknown class is also found. This latter set is found to be more interesting to the ontology engineer, as they might represent novel entity types.

The output of spaCy NER is taken to be the ground truth for those samples with an entity label (see 3.7), and then the model's predictions are clustered using the chosen clustering algorithm. Optimal Cluster Reassignment is then undertaken: cluster re-labelling based on the distribution of assignments applying each entity class label to the one cluster that best matches its members based on which cluster has the highest fraction of this class. Each assignment is made based on the correct assignment of an item to the cluster identified as the best candidate for the NER class derived by spaCy, both per cluster and globally. Note the clusters themselves are never altered, only the cluster label to be used for future analysis.

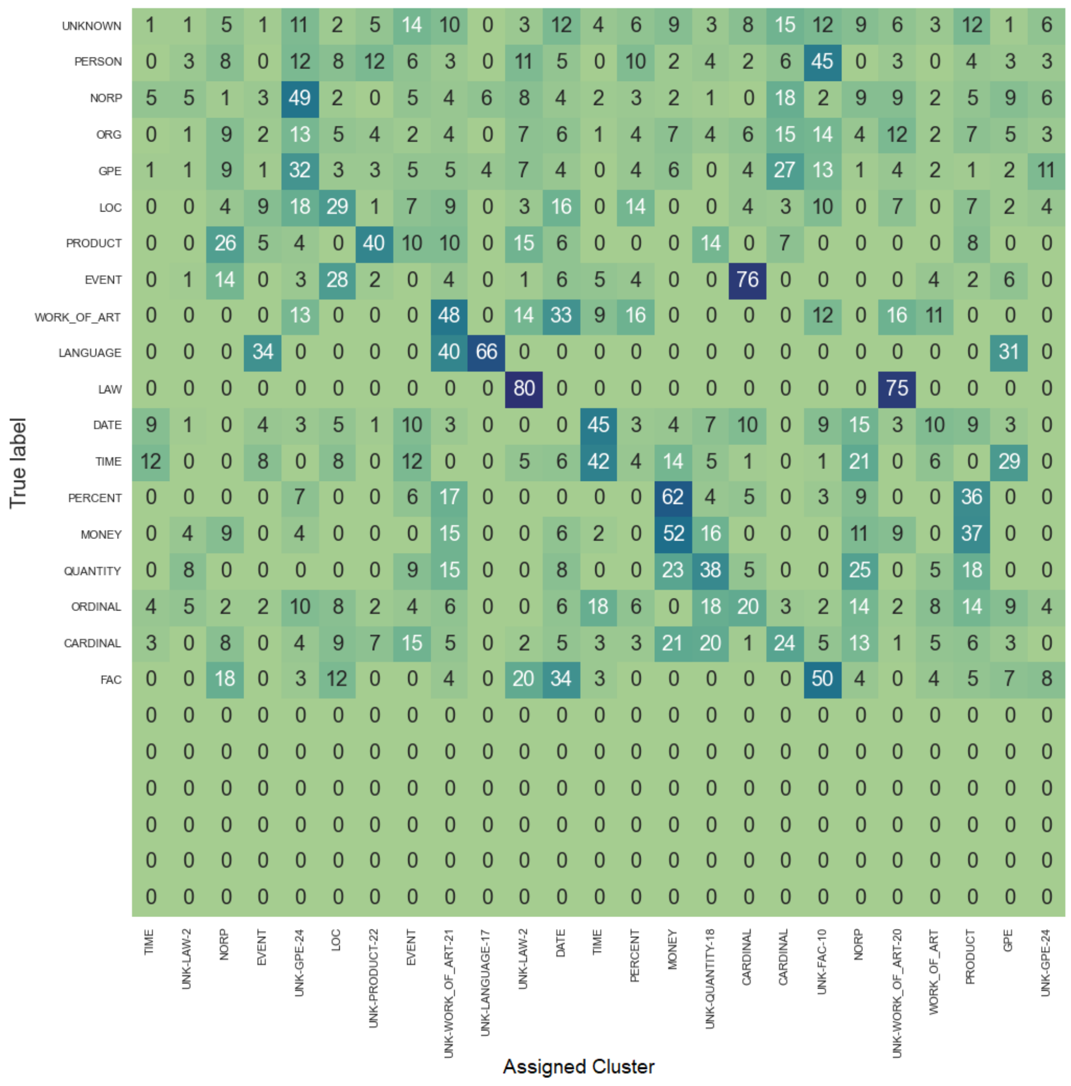
Algorithm 9 Optimal Cluster Reassignment

```
Input: Set of clusters samples $S$                              $S\colon s_i = (emb_i, c_i)$ with label $c \in C$

Output: Set of samples $\hat{S}$ labelled with cluster number   $\hat{S}\colon \hat{s}_i = (emb_i, c_i)$ with label $c \in C$

For each entity class e
  Compute the number of samples labelled in this cluster,           $|s{:}[y_{true} = e]|$
  Compute the vector of true positives for each cluster $c$         $\forall c,\ U_c = s{:}[y_{true} = e]$
  Compute F, the fraction of each class $y_{true}$ for $e$ per cluster,

Initialise entity-cluster mapping, $ECM$
For each class $c_j$ in $C, \equiv$ cluster $j$
   Compute the set of members of cluster $j$                        $S_c = \{s_i^c : c = j\}$

   Compute most common entity class in $S_c$,                       $\hat{c} = argmax(\sum_{i=1}^{n} \{s_i^c : c = j\})$

if frequency for $\hat{c}$ in this cluster $c_j >$ the current optimal,   if $F_{\hat{c}j} > ECM_{\hat{c}j}$
      Update $ECM$ accordingly                                          $ECM_{\hat{c}j} \Rightarrow F_{\hat{c}j}$
   All other clusters are labelled "unknown"
Return *result*
```

The output of Optimal Cluster Reassignment is a set of clusters, each of which is either labelled with its optimal entity label, or with an "unknown" label. Each individual sample in each cluster also has a true (spaCy) entity class. These clusters can then trivially be used to generate confusion matrices or F1 scores.

Per-cluster metrics assess an assignment as positive if the item has been assigned to the primary cluster for its spaCy entity class, then average these measures from each cluster based on the number of entities of the optimal class for the cluster. Per-cluster metrics do not assess clusters which have not been assessed as the optimal cluster for an entity class and thus reject many assessments and score lower than the global F-1 metrics.

The confusion matrix plotted from Optimal Cluster Reassignment will by design demonstrate a strong diagonal, given that the clusters have been rearranged with this aim:



Sample Confusion Matrix Showing Pronounced Diagonal[1]

Of note in the confusion matrix are: the pronounced diagonal; confusion between some classes such as PERCENT, MONEY, CARDINAL; and a series of clusters to the right of the diagonal representing novel semantic clusters identified.

For cluster c with label c^l, containing sample s, of assigned class s^c and true class s^y:

True Positives are labelled correctly for their class and have been placed in this cluster:	$TP_c = \sum_{i=0}^{i=\lvert c\rvert} \left[s_i^c = s_i^y,\ s_i^c = c^l \right]$
False Positives are labelled correctly for their class, which is not the class label of this cluster, but have been placed in this cluster:	$FP_c = \sum_{i=0}^{i=\lvert c\rvert} \left[s_i^c \neq s_i^y,\ s_i^c = c^l \right]$

[1] From test
`test-train_size=3000-radius=0-latent_weight=0.001-reconstr_weight=0.5-head=z-cluster=Kmeans-entity_count=15-noise_factor=0.1-`

False Negatives are labelled correctly for the class of this cluster, but placed in another cluster	$FN = \sum_{i=0}^{i=\lvert c \rvert}\left[s_i^c \neq c^l, s_i^y = c^l\right]$

Enabling us to calculate a sample-wide but per-cluster F1 score as the harmonic mean of precision and recall:

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$F_1 = 2\frac{Precision \times Recall}{Precision + Recall}$$

5.6.1 Global F1 Measure

Global metrics compare the ‘true’ NER class to the final predicted class across all clusters in the sample.

True Positives are labelled correctly for their cluster	$TP = \sum\left[s^c = s^y\right]$
False Positives are labelled incorrectly for their cluster	$FP = \sum_{c=0}^{c=\lvert C \rvert}\left[s^c \neq s^y,\ s^c = c\right]$
False Negatives are labelled correctly for this cluster, but placed in another cluster	$FN = \sum_{c=0}^{c=\lvert C \rvert}\left[s^c = s^y,\ s^c \neq c\right]$

Enabling a global F1 to be calculated as for the per-cluster metrics above.

NOTE: Cluster rearrangement is not used for DeepCluster models, as the cluster prediction is trained to give the actual cluster label, not an arbitrary ID.

5.7 Results

5.7.1 Benchmark

A benchmark was generated to compare the models. This consisted of the application of the full clustering component to raw distilBERT embeddings, without using the trained models to predict clusters, but with optimal cluster reassignment.

5.7.1 Scores across architectures

We use the methods above to score the results of several hundred runs, with the best of each architecture listed below. $\overline{F_1}$ is the average of the RF_1 described in 5.6 Rearranged F1 Measure and the global F_1 in 5.6.1.

	Dims	$\overline{F_1}$	RF_1	F_1 global	Precision	Recall
Benchmark	768	0.3768	0.1881	0.5569	**0.8387**	0.4795
GMM-*AE*	768	0.4168	0.3432	0.4905	0.8013	0.4361
K-means-*Z*	2000	**0.4989**	**0.4516**	**0.5462**	0.7277	**0.5097**
K-means-*Enc*	40	0.4480	0.3533	0.5428	0.8381	0.4946
K-means-*AE*	768	0.4217	0.3235	0.5200	0.8200	0.4681
OPTICS-*AE*	768	*0.5727*	*0.3982*	*0.7472*	*0.9549*	*0.7415*
Agg-*AE*	768	0.4217	0.3235	0.5200	0.8200	0.4681

Table 5. Key Results

Note 1: OPTICS is not used for the final model because although the scores are good on these metrics, the very large number of clusters generated makes it impractical for the use case.

Note 2: Agg (agglomerative clustering) and GMM (Gaussian Mixture Models) are not deployed by default in the final model due to poor performance compared to k-means.

Note 3: *Z*, *Enc*, and *AE* are the outputs of the latent representation network (*Z*), the encoder head (*enc*) and the autoencoder's decoder head (*AE*) respectively.

5.8 Metrics for the use case

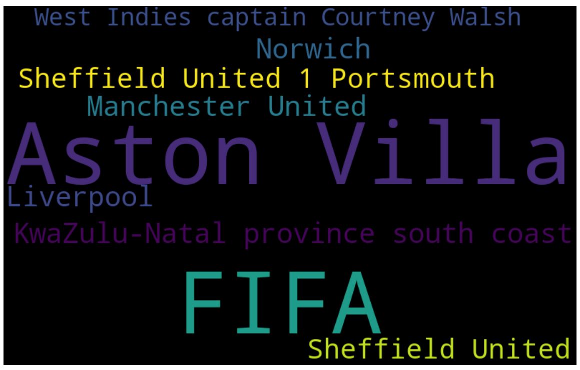
The application is designed to support the discovery of new ontological categories. These are not so amenable to automatic metrics but rather need a more subjective approach. For examples see the next section, 5.9. To generate quantitative metrics for this use case would necessitate human involvement, such as the manual labelling of a dataset with novel categories, then the scoring of how many of those categories end up strongly represented in the final clustering. While this approach would generate appropriate metrics and might be useful for developing the model further, it obviates the requirement for the model in the first place. See 7.3 Human scoring of new clusters below for more discussion.

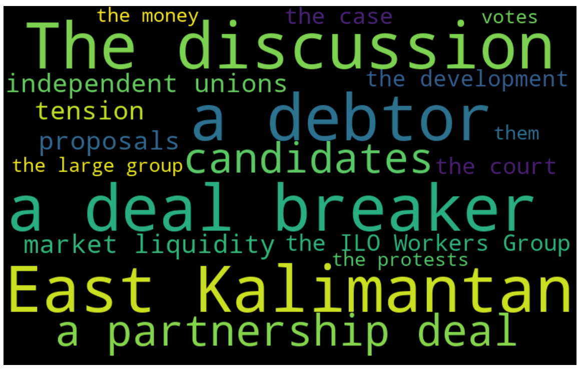
5.9 Qualitative results

The actual use case for this project is the identification of new ontological categories. This is largely a subjective matter, as different people may interpret clusters differently - see 7.3 for discussion. Examples of clear clusters identified by the model follow, along with possible, human-suggested cluster labels which might form the basis of new entity classes in the ontology.

5.9.1 Clusters from the conll test dataset

Hugging Face provides test and train datasets, both derived from the same Reuters corpus. The training dataset was used for the self-supervised training of the model. For overall model validation, the conll test dataset was used, previously unseen by the model, to generate clusters unsupervised and then rearrange semi-supervised with the optimal spaCy entity label. The additional clusters generated beyond those assessed as primarily belonging to the core spaCy / ON5 categories include some clusters with clear semantic coherence, as the following two examples show:


West Indies captain Courtney Walsh
Norwich
Sheffield United 1 Portsmouth
Manchester United
Aston Villa
Liverpool
KwaZulu-Natal province south coast
FIFA
Sheffield United
"Football" or "Football teams"



"Negotiations and Discussions" or "financial agreements"

5.9.2 Clusters from novel documents

For a task more representative of the use case, a set of PDFs were selected effectively at random[2] from a novel corpus, not related to news topics from which conll Reuters data is derived. The PDFs were then processed using the process_files.py utility and passed through the pretrained deep clustering network.

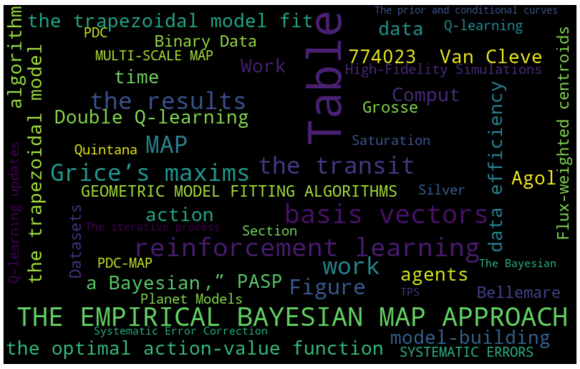
For reference, the titles are:

- Kepler Data Processing Handbook (Jenkins, 2017)
- Building machines that learn and think like people (Lake *et al.*, 2017)
- Adaptive nodes enrich nonlinear cooperative learning beyond traditional adaptation by links (Sardi *et al.*, 2018)
- Rainbow: Combining Improvements in Deep Reinforcement Learning (Hessel *et al.*, 2017)
- Neuroanatomy of Dopamine Reward and Addiction (Taber *et al.*, 2012)
- Representation Learning on Graphs: A Reinforcement Learning Application (Madjiheurem and Toni, no date)
- Mapping value based planning and extensively trained choice in the human brain (Wunderlich, Dayan and Dolan, 2012)

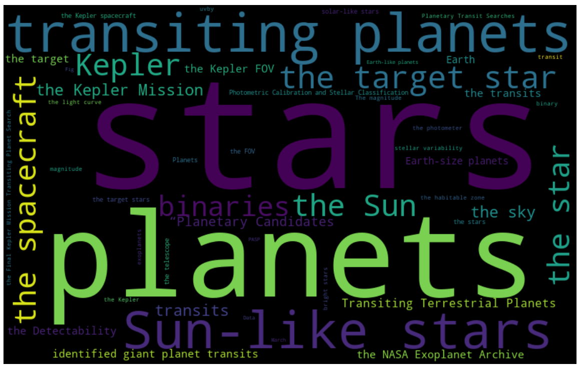
[2] These were the first 12 results from the author's stored papers collection, with little more in common than "academic topics I am interested in"

- Predictive Reward Signal of Dopamine Neurons (Schultz, 1998)
- Overview Of The Kepler Science Processing Pipeline (Jenkins *et al.*, 2010)
- Veiled Nonlocality and Cosmic Censorship (Kafatos and Kak, no date)
- Intentional action: Conscious experience and neural prediction (Haggard and Clark, 2003)
- Kepler Planet-Detection Mission: Introduction and First Results (Borucki *et al.*, 2010)

Two examples are provided here, with additional examples in Appendix B Sample Clusters Found and full outputs included in the github repository under the folder `\test-pdfs-dec`, ideally accessed from the file `index.html`.



"Algorithms"



"Astronomical terms"

6 Conclusions

To recall, section 3.1 Use Case states:

> The application for the project is the identification of clusters of terms which are not recognised by the NER system but are semantically coherent and indicative of a potential new ontological category present in the data.

The author suggests that the results obtained in section 5.9.2 Clusters from novel documents above clearly demonstrate the value of the approach. A series of semantically coherent clusters have been generated which are not derived from the spaCy / Ontonotes ontology and would likely be immediate pointers to new entity classes for an organisation for whom the 12 PDFs were representative. The model would then act as a poor-quality entity classifier for entities of these categories, which may assist in the generation of labelled training sets for the actual fine-tuning of the spaCy (or other) NER classification model.

7 Weaknesses & Future Work

7.1 Fine Tuning BERT

BERT is commonly fine-tuned for specific tasks (Howard and Ruder, 2018). This typically consists of freezing the BERT layer weights, then adding one or more extra layers on top of the last BERT layer and training for the task at hand. To apply this approach to the task at hand, an alternative architecture could be considered: apply the clustering layer directly on top of BERT to learn the clusters directly, as per (Xie, Girshick and Farhadi, 2015).

7.2 Better training datasets

The dataset used was relatively small at ~14k sentences. This increases the risk of overfitting or memorisation in a complex model of ~6M parameters, especially when compared to the complexity of the language model used: "LMs overfit to small datasets and suffered catastrophic forgetting when fine-tuned with a classifier." (Howard and Ruder, 2018). To reduce this risk, a larger dataset should be obtained or generated. This is not an excessively onerous task, given the initial entity labelling is automated. For example, repeated crawling of the BBC news or similar websites will generate large amounts of material, which can then be pre-processed with BERT for embeddings and spaCy for labelled data generation.

7.3 Human scoring of new clusters

Discovery of relevant new clusters was undertaken by intuitive human analysis in this project. This reduces the value of the results. Future work might include the identification of clusters prior to training, based on human knowledge, and then the training of the system to find such clusters. Alternatively, humans could score the identified clusters against a defined scoring schema to allow the model success to be better evaluated. Given the use case of finding relevant clusters for a given industry or organisation, appropriate tagging of such items in advance would support the evaluation of the applicability to that domain.

7.4 Other languages

Both models used in data preparation were trained on English-only corpora: the distilBERT language model (*distilbert-base-uncased · Hugging Face*, no date), derived from the pre-trained Hugging Face BERT model; as well as the spaCy (`en_core_web_sm`) model. The conll training data used was all English, from a Reuters news corpus. The models generated were therefore heavily optimised for the English language and should not be relied upon in any way for non-English texts. To improve performance on other languages may simply be a matter of replacing both these models with versions pre-trained on the appropriate language, then retraining the `DeepLatentCluster` model on this dataset.

7.5 Deep clustering the latent space

The two models inspiring this project, (Xie, Girshick and Farhadi, 2015) and (Dahal, 2018), have different strengths. Xie et al.'s Deep Embedded Clustering directly clusters the data offering computational efficiency, whereas Dahal's Latent Representation Network learns the latent space alongside the encoder to separate clusters out from each. A combination of the two approaches was not completed for this project, but could be achieved by adding a clustering layer either on top of the decoder, or on top of the latent space. This will complicate the calculation of loss, necessitating multiple loss functions, and therefore require a large training set, but may lead to an optimal solution.

8 References

Achtert, E. *et al.* (2012) *Evaluation of clusterings - metrics and visual support*. Available at: https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=0114DF890EC2A922A9EEDFB84C4764D8?doi=10.1.1.701.4339&rep=rep1&type=pdf (Accessed: 26 August 2022).

Aggarwal, C.C., Hinneburg, A. and Keim, D.A. (2001) 'On the Surprising Behavior of Distance Metrics in High Dimensional Space', in *Database Theory — ICDT 2001*. Springer Berlin Heidelberg, pp. 420–434.

Ankerst, M. *et al.* (1999) 'OPTICS: ordering points to identify the clustering structure', *SIGMOD Rec.*, 28(2), pp. 49–60.

Bahdanau, D., Cho, K. and Bengio, Y. (2014) 'Neural Machine Translation by Jointly Learning to Align and Translate', *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/1409.0473.

Borucki, W.J. *et al.* (2010) 'Kepler planet-detection mission: introduction and first results', *Science*, 327(5968), pp. 977–980.

Chawla, N.V. *et al.* (2002) 'SMOTE: Synthetic Minority Over-sampling Technique', *Journal of Artificial Intelligence Research*, 16, pp. 321–357.

Chidananda Gowda, K. and Krishna, G. (1978) 'Agglomerative clustering using the concept of mutual nearest neighbourhood', *Pattern recognition*, 10(2), pp. 105–112.

Craw, S. (2010) 'Manhattan Distance', in Sammut, C. and Webb, G.I. (eds) *Encyclopedia of Machine Learning*. Boston, MA: Springer US, pp. 639–639.

Dahal, P. (2018) 'Learning Embedding Space for Clustering From Deep Representations', in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 3747–3755.

Dahal, P. (2019) *deepclustering: Author implementation of deep clustering model from the paper 'Learning Embedding Space for Clustering From Deep Representations' 2018 IEEE BigData*. Github. Available at: https://github.com/parasdahal/deepclustering (Accessed: 18 September 2022).

Devlin, J. *et al.* (2019) 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding', in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186.

distilbert-base-uncased · Hugging Face (no date). Available at: https://huggingface.co/distilbert-base-uncased (Accessed: 11 September 2022).

Ester *et al.* (no date) 'A density-based algorithm for discovering clusters in large spatial databases with noise', *KDD: proceedings / International Conference on Knowledge Discovery & Data Mining. International Conference on Knowledge Discovery & Data Mining* [Preprint]. Available at: https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf?source=post_page.

Fonseca, F. (2007) 'The Double Role of Ontologies in Information Science Research', *Journal of the American Society for Information Science and Technology*, 58(6), pp. 786–793.

Grishman, R. and Sundheim, B.M. (1996) 'Message understanding conference-6: A brief history', in *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*. Available at: https://aclanthology.org/C96-1079.pdf.

Guo, X. (no date) *DEC-keras: Keras implementation for Deep Embedding Clustering (DEC)*. Github. Available at: https://github.com/XifengGuo/DEC-keras (Accessed: 18 September 2022).

Haggard, P. and Clark, S. (2003) 'Intentional action: conscious experience and neural prediction', *Consciousness and cognition*, 12(4), pp. 695–707.

Hessel, M. *et al.* (2017) 'Rainbow: Combining Improvements in Deep Reinforcement Learning', *arXiv [cs.AI]*. Available at: http://arxiv.org/abs/1710.02298.

Howard, J. and Ruder, S. (2018) 'Universal language model fine-tuning for text classification', in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Stroudsburg, PA, USA: Association for Computational Linguistics. doi:10.18653/v1/p18-1031.

Jenkins, J.M. *et al.* (2010) 'OVERVIEW OF THE KEPLER SCIENCE PROCESSING PIPELINE', *The Astrophysical Journal Letters*, 713(2), p. L87.

Jenkins, J.M. (ed.) (2017) *Kepler Data Processing Handbook: KSCI-19081-002*. NASA Ames Research Center.

Jin, X. and Han, J. (2010) 'K-Means Clustering', in Sammut, C. and Webb, G.I. (eds) *Encyclopedia of Machine Learning*. Boston, MA: Springer US, pp. 563–564.

Kafatos, M. and Kak, S. (no date) 'Veiled Nonlocality and Cosmic Censorship'.

Kingma, D.P. and Ba, J. (2014) 'Adam: A Method for Stochastic Optimization', *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/1412.6980.

Koziarski, M., Woźniak, M. and Krawczyk, B. (2020) 'Combined Cleaning and Resampling algorithm for multi-class imbalanced data with label noise', *Knowledge-Based Systems*, 204, p. 106223.

Lake, B.M. *et al.* (2017) 'Building machines that learn and think like people', *The Behavioral and brain sciences*, 40, p. e253.

Lemaître, G., Nogueira, F. and Aridas, C.K. (2017) 'Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning', *Journal of machine learning research: JMLR*, 18(17), pp. 1–5.

Liang, J. and Liu, R. (2015) 'Stacked denoising autoencoder and dropout together to prevent overfitting in deep neural network', in *2015 8th International Congress on Image and Signal Processing (CISP)*, pp. 697–701.

MacKay, D.J.C., Kay, D.J.C.M. and Press, C.U. (2003) *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

Madjiheurem, S. and Toni, L. (no date) 'Representation Learning on Graphs: A Reinforcement Learning Application'.

Mattmann, C. (no date) *tika-python: Tika-Python is a Python binding to the Apache Tika^{TM} REST services allowing Tika to be called natively in the Python community*. Github. Available at: https://github.com/chrismattmann/tika-python (Accessed: 16 September 2022).

Nielsen, F. (2016) *Introduction to HPC with MPI for Data Science*. Springer International Publishing, pp. 203–204.

Ontology (2017). Available at:

https://warwick.ac.uk/fac/soc/ces/research/current/socialtheory/maps/ology/ (Accessed: 29 August 2022).

Papineni, K. *et al.* (2002) 'Bleu: a Method for Automatic Evaluation of Machine Translation', in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, pp. 311–318.

Reimers, N. and Gurevych, I. (2019) 'Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks', *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/1908.10084.

van Rossum, G., Lehtosalo, J. and Langa, Ł. (2014) *PEP 484 – Type Hints*. Available at: https://peps.python.org/pep-0484/ (Accessed: 11 September 2022).

Sardi, S. *et al.* (2018) 'Adaptive nodes enrich nonlinear cooperative learning beyond traditional adaptation by links', *Scientific reports*, 8(1), p. 5100.

Schultz, W. (1998) 'Predictive reward signal of dopamine neurons', *Journal of neurophysiology*, 80(1), pp. 1–27.

sklearn.mixture.GaussianMixture (no date) *scikit-learn*. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html (Accessed: 1 September 2022).

Smith, E.V. (2017) *PEP 557 – Data Classes*. Available at: https://peps.python.org/pep-0557/ (Accessed: 11 September 2022).

spaCy (no date) *spaCy · Industrial-strength Natural Language Processing in Python*. Available at: https://spacy.io/ (Accessed: 13 February 2022).

Taber, K.H. *et al.* (2012) 'Neuroanatomy of Dopamine: Reward and Addiction', *J Neuropsychiatry Clin Neurosci*. Edited by R.A. Hurley, A. Hayman, and K.H. Taber, 24(1).

Taboga, M. (2017) 'Gaussian mixture - Maximum likelihood estimation', in *Lectures on Probability Theory and Mathematical Statistics - 3rd Edition*. Kindle Direct Publishing, p. Online appendix.

The HF Datasets community (no date) 'conll2003'. Available at: https://huggingface.co/datasets/conll2003 (Accessed: 1 September 2022).

Tjong Kim Sang, E. and De Meulder, F. (2003) 'Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition', *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/cs/0306050.

Van Maaten and Hinton (no date) 'Visualizing data using t-SNE', *Journal of machine learning research: JMLR* [Preprint]. Available at: https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf?fbcl.

Vaswani, A. *et al.* (2017) 'Attention Is All You Need', *arXiv [cs.CL]*. Available at: http://arxiv.org/abs/1706.03762.

Vincent, P. *et al.* (2008) 'Extracting and composing robust features with denoising autoencoders', in *Proceedings of the 25th international conference on Machine learning - ICML '08. the 25th international conference*, New York, New York, USA: ACM Press. doi:10.1145/1390156.1390294.

Wang, S. and Yao, X. (2012) 'Multiclass imbalance problems: Analysis and potential solutions', *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics: a publication of the IEEE Systems, Man, and Cybernetics Society*, 42(4), pp. 1119–1130.

Wunderlich, K., Dayan, P. and Dolan, R.J. (2012) ‘Mapping value based planning and extensively trained choice in the human brain’, *Nature neuroscience*, 15(5), pp. 786–791.

Xie, J., Girshick, R. and Farhadi, A. (2015) ‘Unsupervised Deep Embedding for Clustering Analysis’, *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/1511.06335.

Xue, N. (2012) ‘OntoNotes Release 5.0’. Available at: https://catalog.ldc.upenn.edu/docs/LDC2013T19/OntoNotes-Release-5.0.pdf.

Appendix A - Hyperparameters Optimised

`train_size`: the training set size. Varied from 100 (for speed), 100, 3000, 10000 and full dataset. Better results were obtained with larger datasets, but training is slower. Optimal value: Full

`radius`: the number of tokens either side of the noun chunk to be included in the embeddings. Values tried were 0, 2, 4, 6, 8, 10, with optimal results at both 0 and 6. Optimal value: 0 or 6

`latent_weight`: the weight applied to the value of the latent_weight function. Values trialled by order of magnitude from 10^{-5} to 1.0 in steps of 10×.
Optimal value:

`recontstr_weight`: the weight applied to the autoencoder reconstruction loss in the full model training. Ranges between 0.1 and 1.0 in steps of 0.1.
Optimal value:

`head`: which head is used to generate the predictions for the clusterer during the evaluation stage. Could be the output of the latent representation network, of the reconstructed autoencoder output, or of the encoder.
Optimal value: Z-head, the latent representation network.

`cluster`: which cluster algorithm to use, from k-means, Gaussian mixtures method, OPTICS or agglomerative clustering. OPTICS generated high scores on the metrics but clusters of little value to the use case, as it tends to produce a large number of small clusters.
Optimal value: K-means or GMM

`noise_factor`: how much gaussian noise to apply, on a scale 0.0 to 1.0 in steps of 0.1
Optimal value: 0.1

`min_cluster_size`: for OPTICS clustering. It was found that higher values decreased the F1 score dramatically. options 5,10,15,20.
Optimal value: 5

`entity_count`: which entities to include in the training set, taking the computed list of entity frequency in the data. entity_count of 10 means the top 10 most common entities. Ranged from 3, 5, 10, 15, All.
Optimal value: 15

`alpha1`: the α factor to be applied to the Student's distribution for the P-distribution for $\hat{y}$. Trialled over 20, 40, 60, 80, 100

`alpha2`: the α factor to be applied to the Student's distribution for the Q-distribution for Z, the latent representation. Trialled over 0.5, 0.75, 1.0, 1.25, 1.5, 2.0

Appendix B - Sample Clusters Found

B.1 Clusters from the conll test dataset

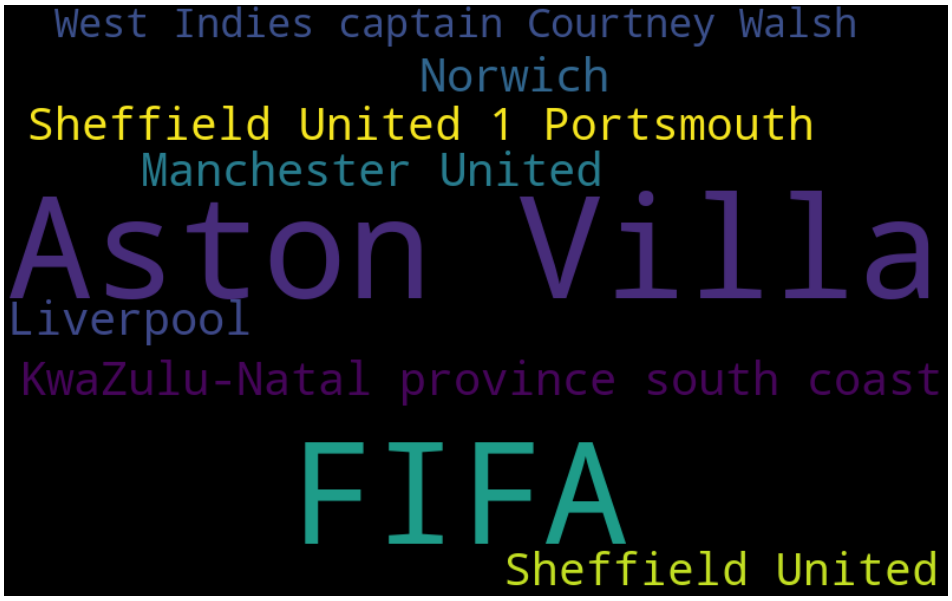

West Indies captain Courtney Walsh
Norwich
Sheffield United 1 Portsmouth
Manchester United
Aston Villa
Liverpool
KwaZulu-Natal province south coast
FIFA
Sheffield United

Figure B.1. This first cluster might be "Sports" or "Sporting Teams"

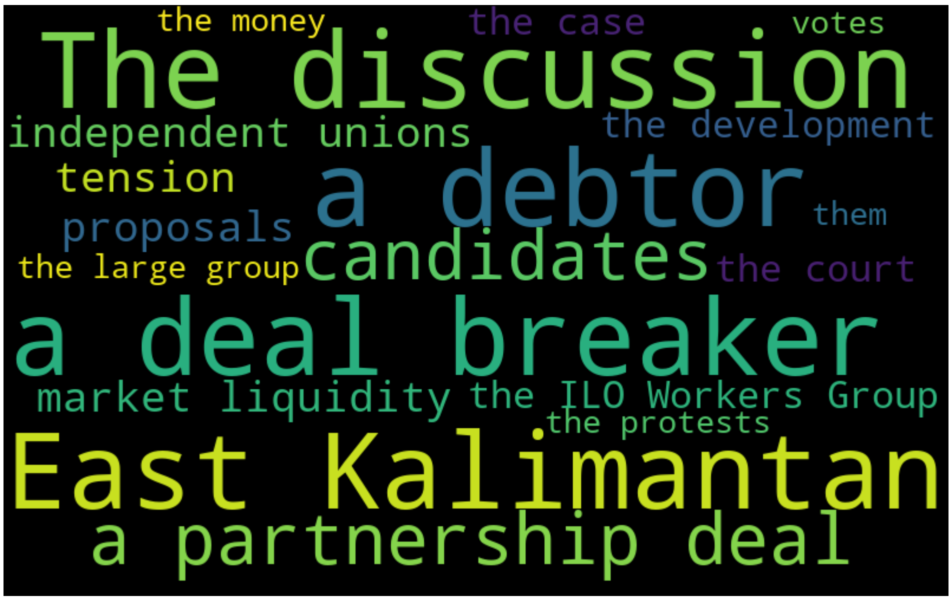


Figure B.2. "Negotiations and Discussions" or "financial agreements"

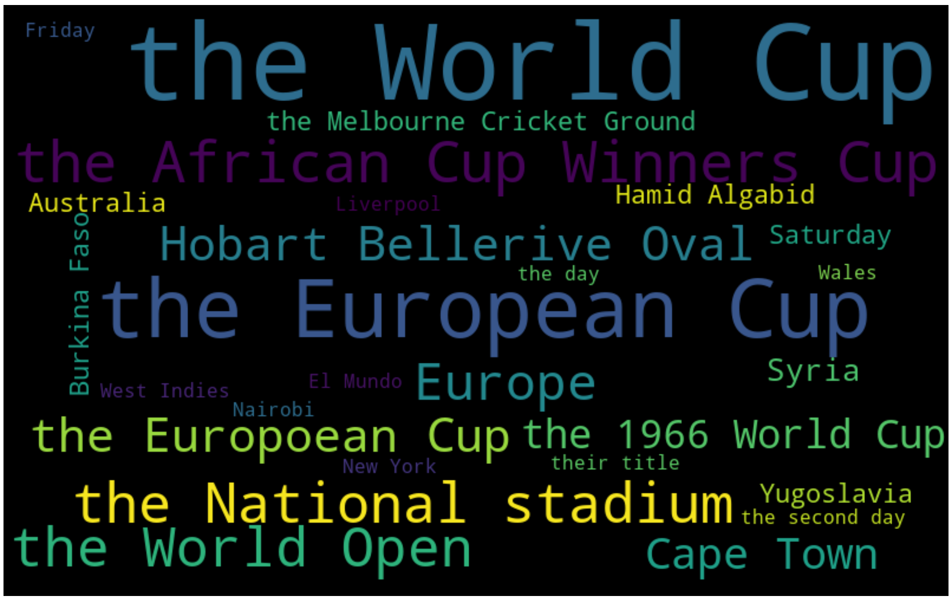


Figure B.3. "Sporting competitions"

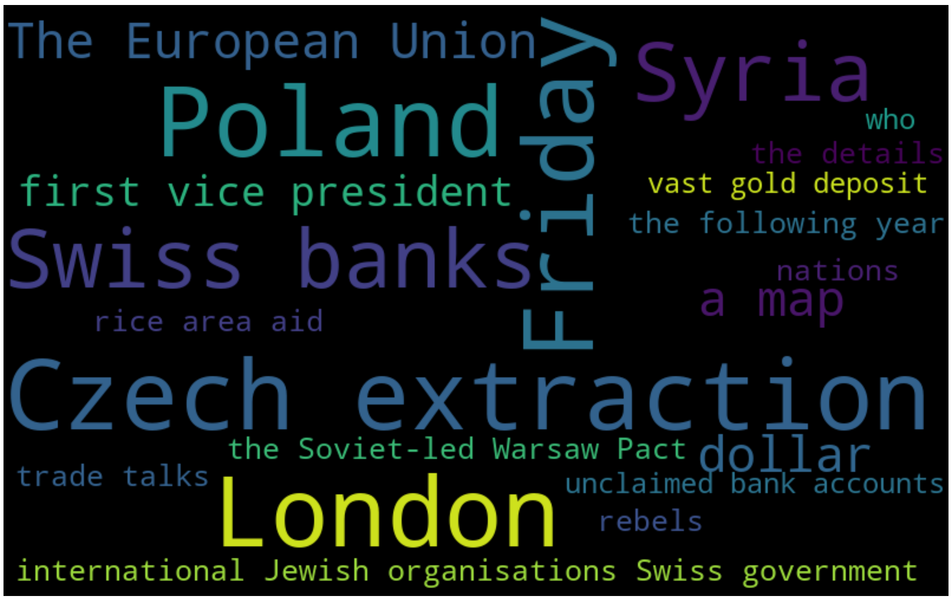


Figure B.4. "Foreign affairs"

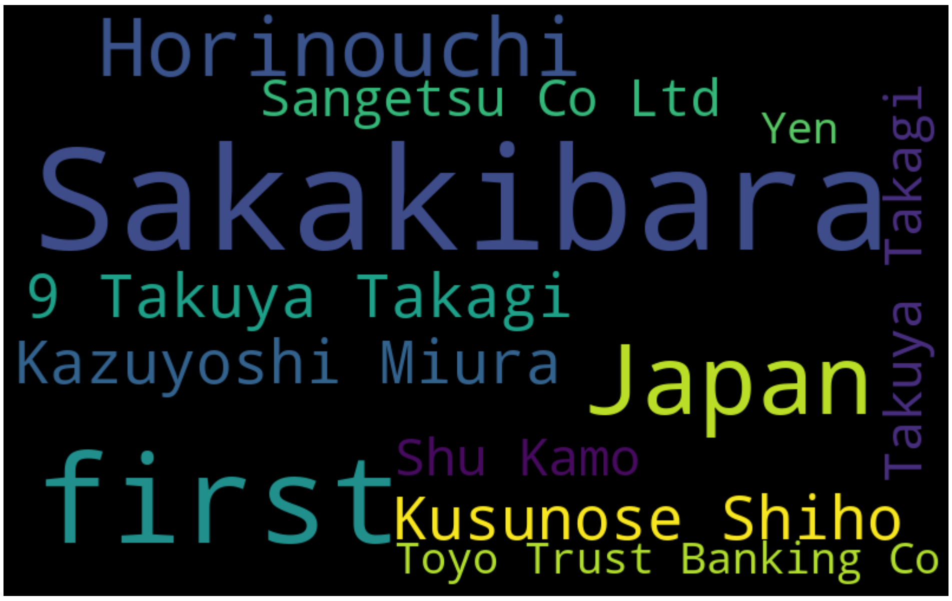


Figure B.5. "Japanese Topics"

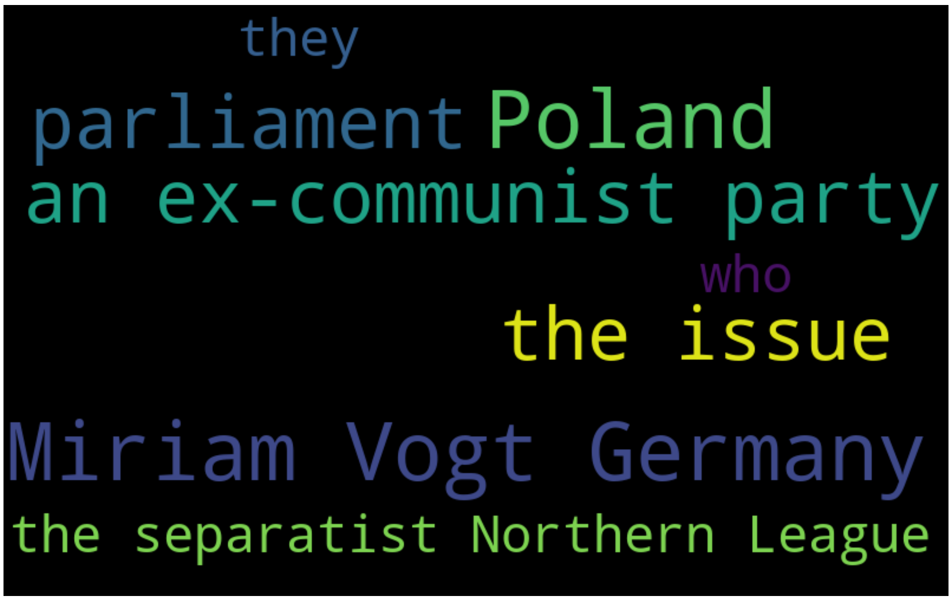


Figure B.6. "Eastern-European Politics"

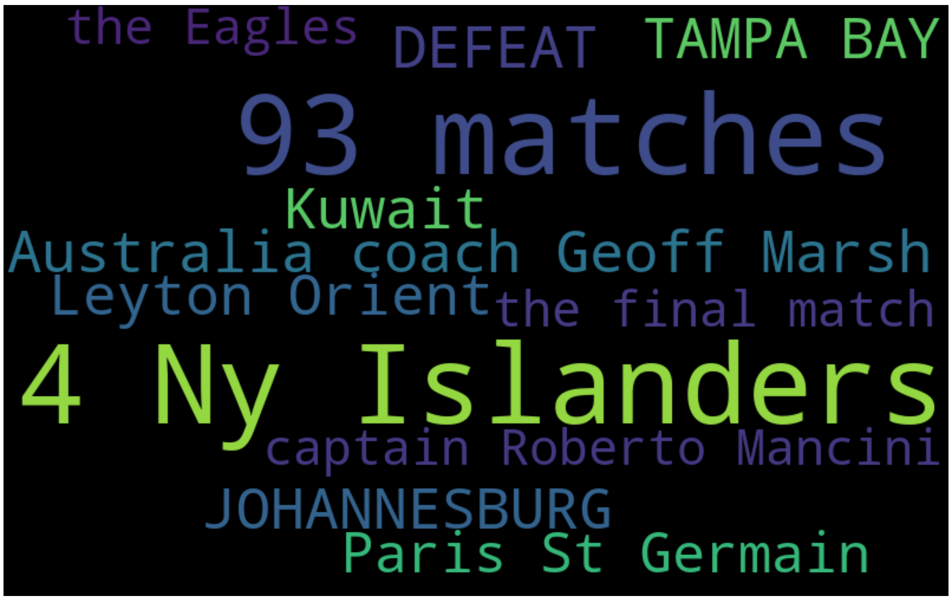


Figure B.7. "Sport matches"

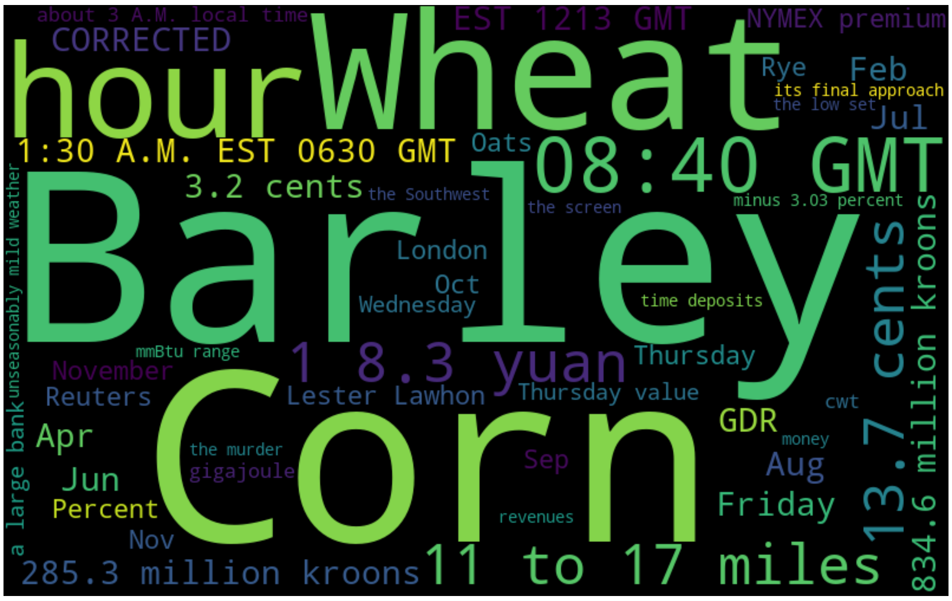


Figure B.8. "The business of farming"

B.2 Clusters from novel documents

For the a more representative tasks given the use case, a set of PDFs were selected effectively at random[3] from a novel corpus, not related to news topics from which conll Reuters data is derived. The PDFs were then processed using the process_files.py utility, and passed through the pretrained deep clustering network.

For reference, the titles are:

- Kepler Data Processing Handbook (Jenkins, 2017)
- Building machines that learn and think like people (Lake *et al.*, 2017)
- Adaptive nodes enrich nonlinear cooperative learning beyond traditional adaptation by links (Sardi *et al.*, 2018)
- Rainbow: Combining Improvements in Deep Reinforcement Learning (Hessel *et al.*, 2017)
- Neuroanatomy of Dopamine Reward and Addiction (Taber *et al.*, 2012)
- Representation Learning on Graphs: A Reinforcement Learning Application (Madjiheurem and Toni, no date)
- Mapping value based planning and extensively trained choice in the human brain (Wunderlich, Dayan and Dolan, 2012)
- Predictive Reward Signal of Dopamine Neurons (Schultz, 1998)
- Overview Of The Kepler Science Processing Pipeline (Jenkins *et al.*, 2010)
- Veiled Nonlocality and Cosmic Censorship (Kafatos and Kak, no date)
- Intentional action: Conscious experience and neural prediction (Haggard and Clark, 2003)
- Kepler Planet-Detection Mission: Introduction and First Results (Borucki *et al.*, 2010)

These outputs are included in the github repository under the folder `\test-pdfs-dec`, ideally accessed from the file `index.html`.

Figures are titled with possible, manually-generated labels

[3] These were the first 12 results from the author's stored papers collection, with little more in common than "academic topics I am interested in"

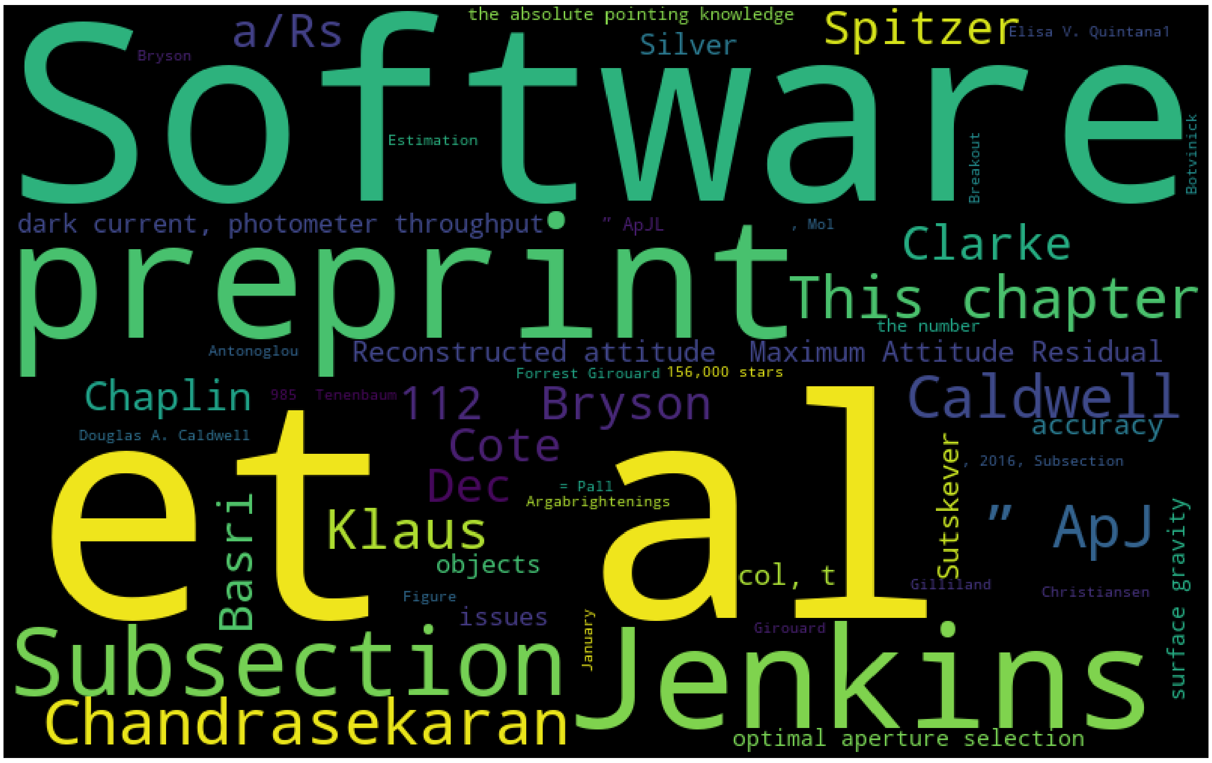


Figure B.9. The first cluster might be "Document structure"

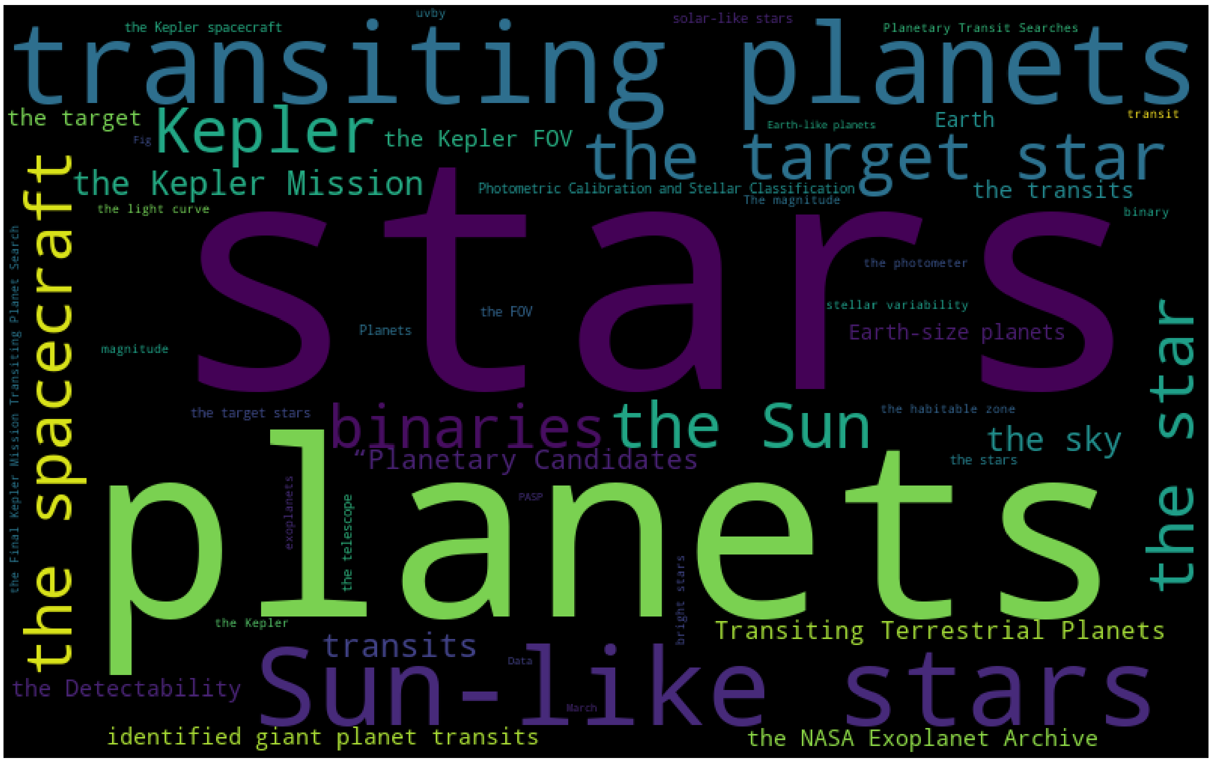


Figure B.10. "Astronomical bodies"

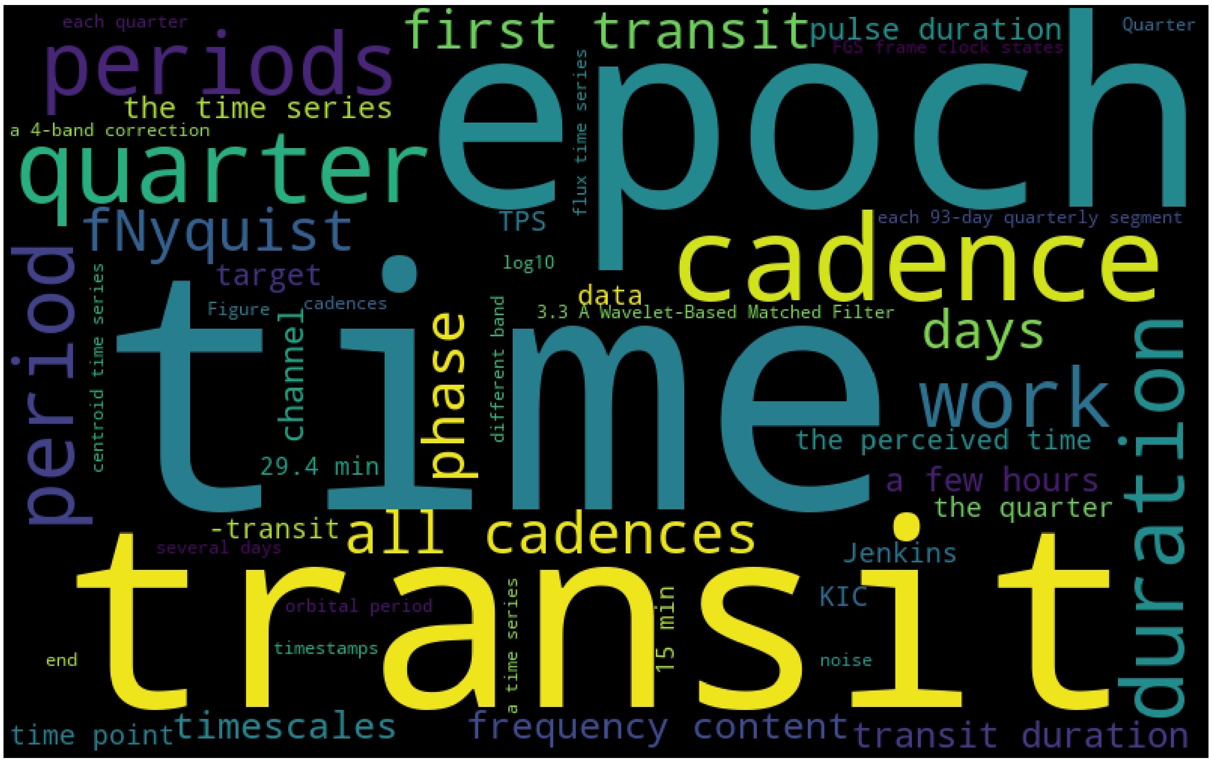


Figure B.11. "Time-related astronomy concepts"

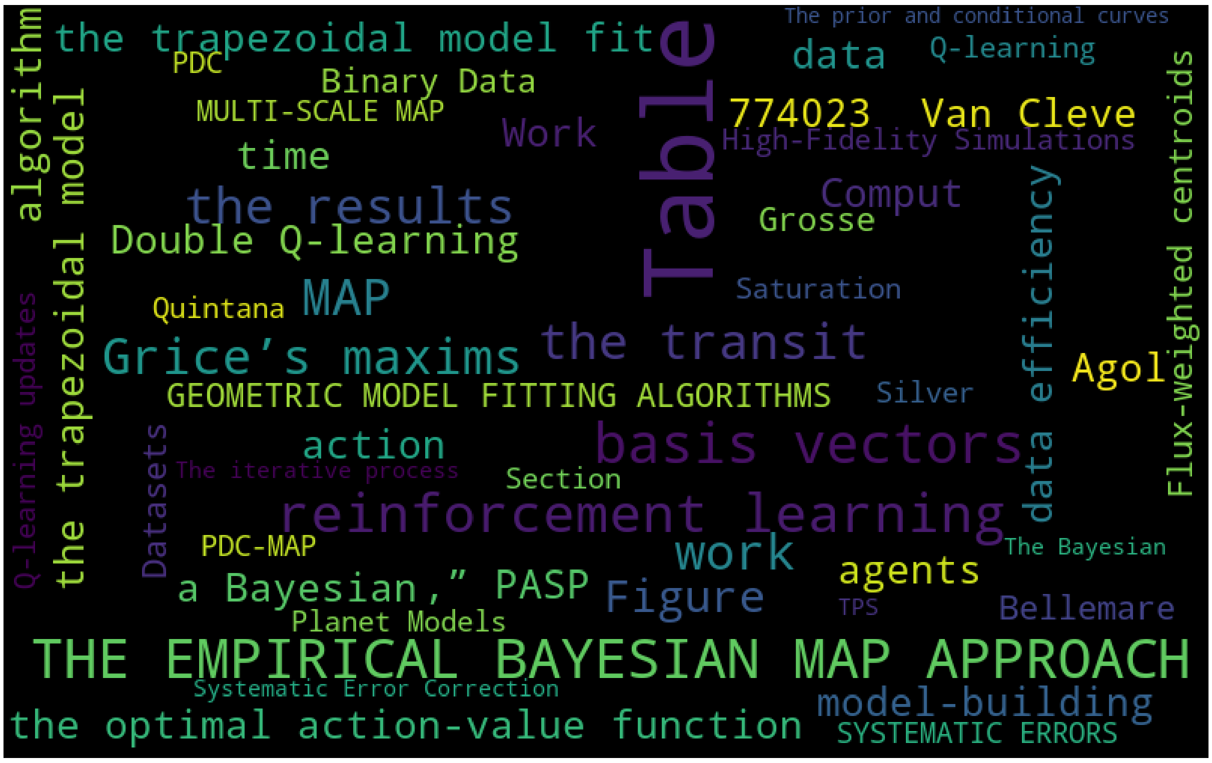


Figure B.12. "Algorithms"

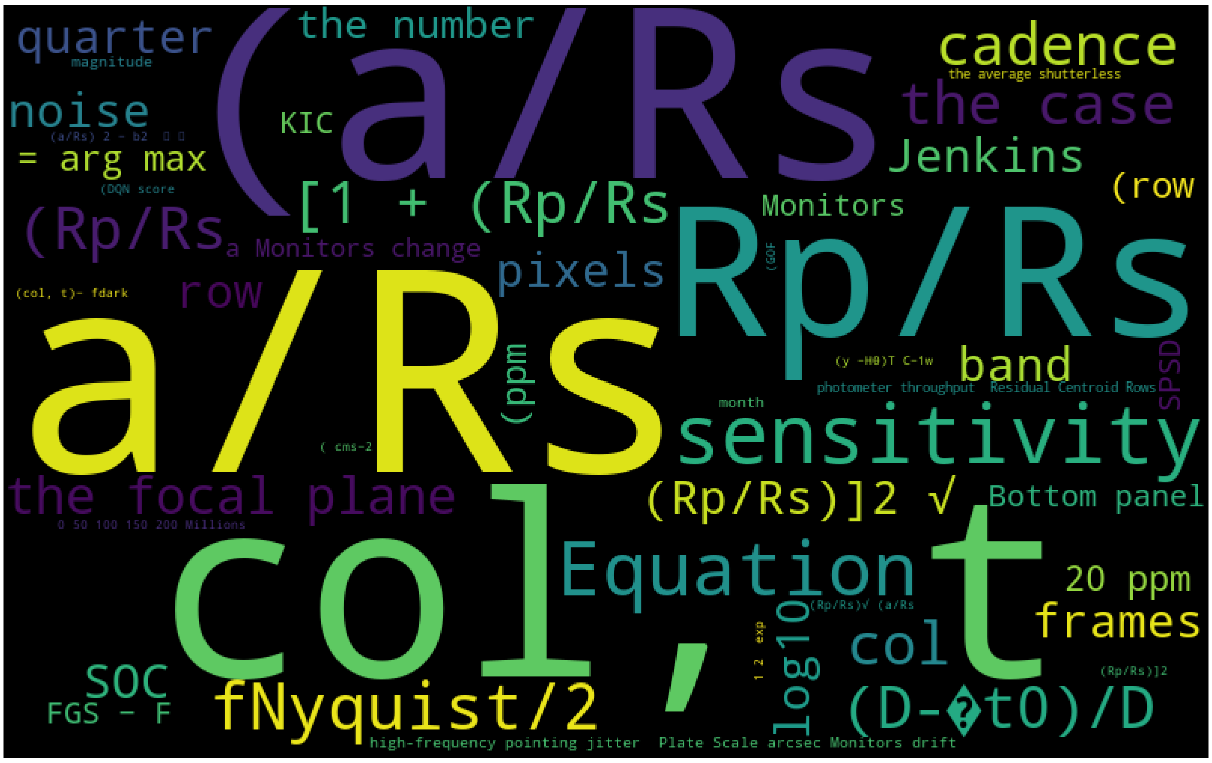


Figure B.13. "Equations and formulae"

Appendix C - Model Descriptions

C.1 DeepCluster

Tensorflow model summary:

```
 Layer (type)                   Output Shape         Param #     Connected to
==============================================================================
 input (InputLayer)             [(None, 768)]        0           []

 encoder_0 (Dense)              (None, 500)          384500      input
 encoder_1 (Dense)              (None, 500)          250500      encoder_0
 encoder_2 (Dense)              (None, 2000)         1002000     encoder_1
 encoder_3 (Dense)              (None, 40)           80040       encoder_2
 decoder_3 (Dense)              (None, 2000)         82000       encoder_3
 decoder_2 (Dense)              (None, 500)          1000500     decoder_3
 decoder_1 (Dense)              (None, 500)          250500      decoder_2
 clustering (ClusteringLayer)   (None, 25)           1000        encoder_3
 decoder_0 (Dense)              (None, 768)          384768      decoder_1
==============================================================================
Total params: 3,435,808
Trainable params: 3,435,808
Non-trainable params: 0
```

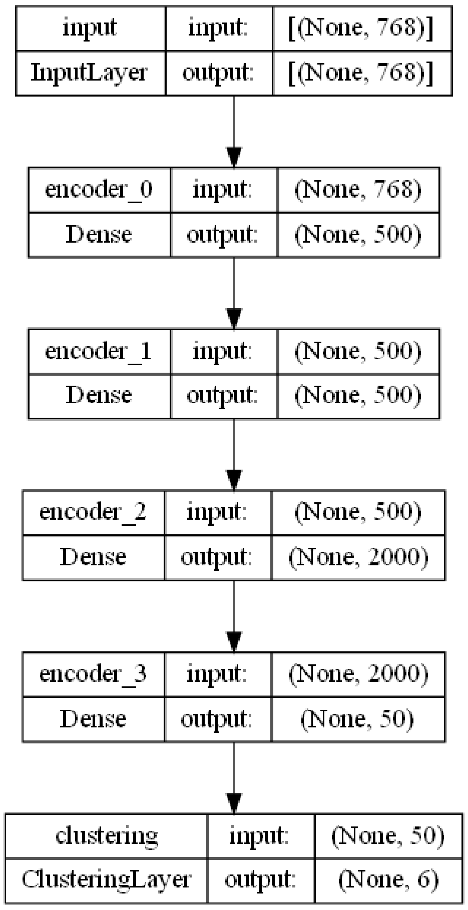
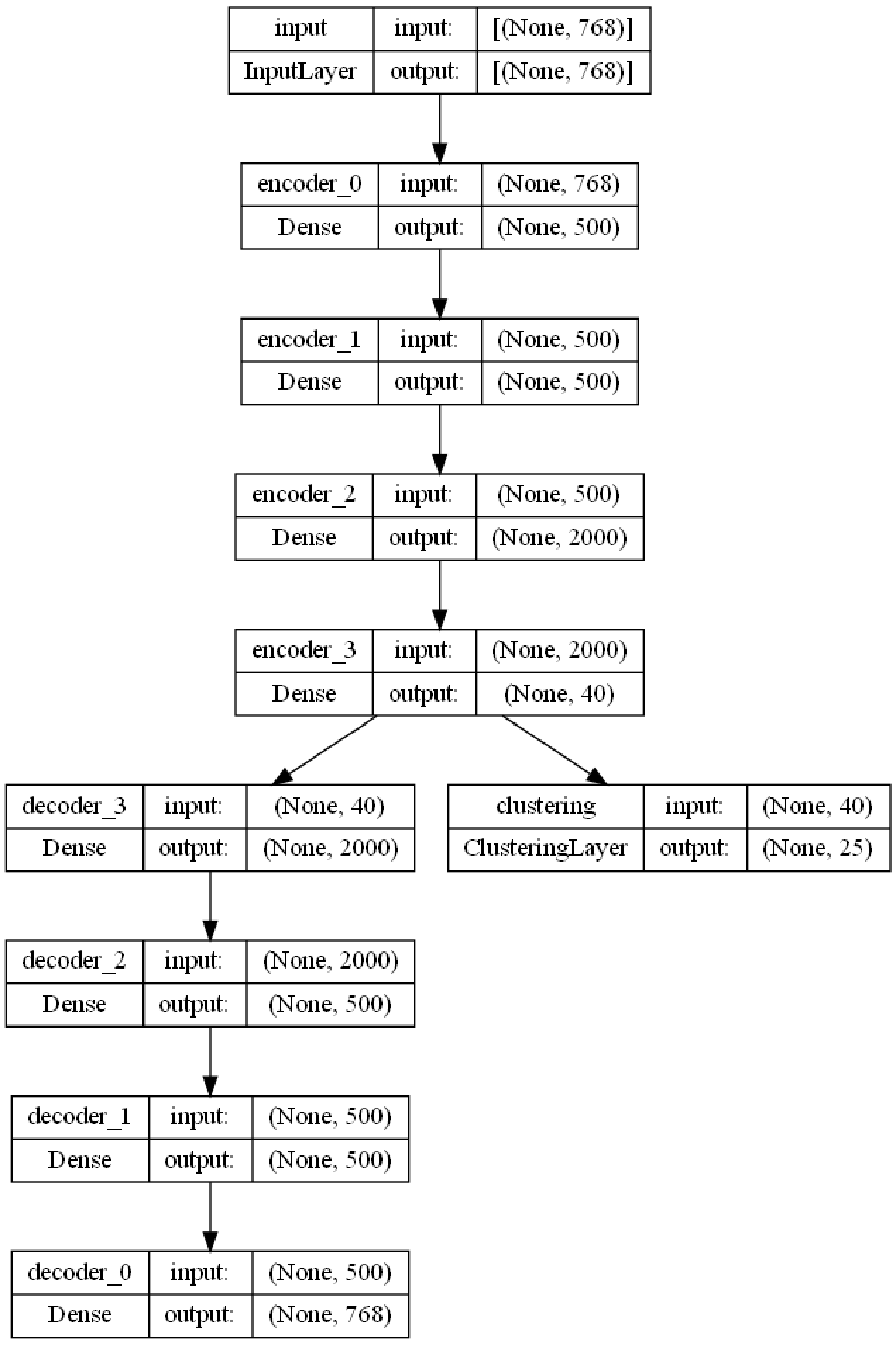


Figure C.1. Deep Cluster Inference Architecture


input | input: | [(None, 768)]
InputLayer | output: | [(None, 768)]
encoder_0 | input: | (None, 768)
Dense | output: | (None, 500)
encoder_1 | input: | (None, 500)
Dense | output: | (None, 500)
encoder_2 | input: | (None, 500)
Dense | output: | (None, 2000)
encoder_3 | input: | (None, 2000)
Dense | output: | (None, 40)
decoder_3 | input: | (None, 40)
Dense | output: | (None, 2000)
clustering | input: | (None, 40)
ClusteringLayer | output: | (None, 25)
decoder_2 | input: | (None, 2000)
Dense | output: | (None, 500)
decoder_1 | input: | (None, 500)
Dense | output: | (None, 500)
decoder_0 | input: | (None, 500)
Dense | output: | (None, 768)
Figure C.2. Deep Cluster Training Architecture

C.2 DeepLatentCluster

Tensorflow model summary:

```
______________________________________________________________________________
 Layer (type)                 Output Shape         Param #     Connected to
==============================================================================
 input (InputLayer)           [(None, 768)]        0           []
 encoder_0 (Dense)              (None, 500)          384500      input
 encoder_1 (Dense)              (None, 500)          250500      encoder_0
 encoder_2 (Dense)              (None, 2000)         1002000     encoder_1
 encoder_3 (Dense)              (None, 40)           80040       encoder_2
 decoder_3 (Dense)              (None, 2000)         82000       encoder_3
 decoder_2 (Dense)              (None, 500)          1000500     decoder_3
 decoder_1 (Dense)              (None, 500)          250500      decoder_2
 decoder_0 (Dense)              (None, 768)          384768      decoder_1
 latent_0 (Dense)               (None, 2000)         82000       encoder_3
 latent_1 (Dense)               (None, 500)          1000500     latent_0
 latent_2 (Dense)               (None, 500)          250500      latent_1
 latent_out (Dense)             (None, 2000)         1002000     latent_2
==============================================================================
Total params: 5,769,808
Trainable params: 5,769,808
Non-trainable params: 0
______________________________________________________________________________
```

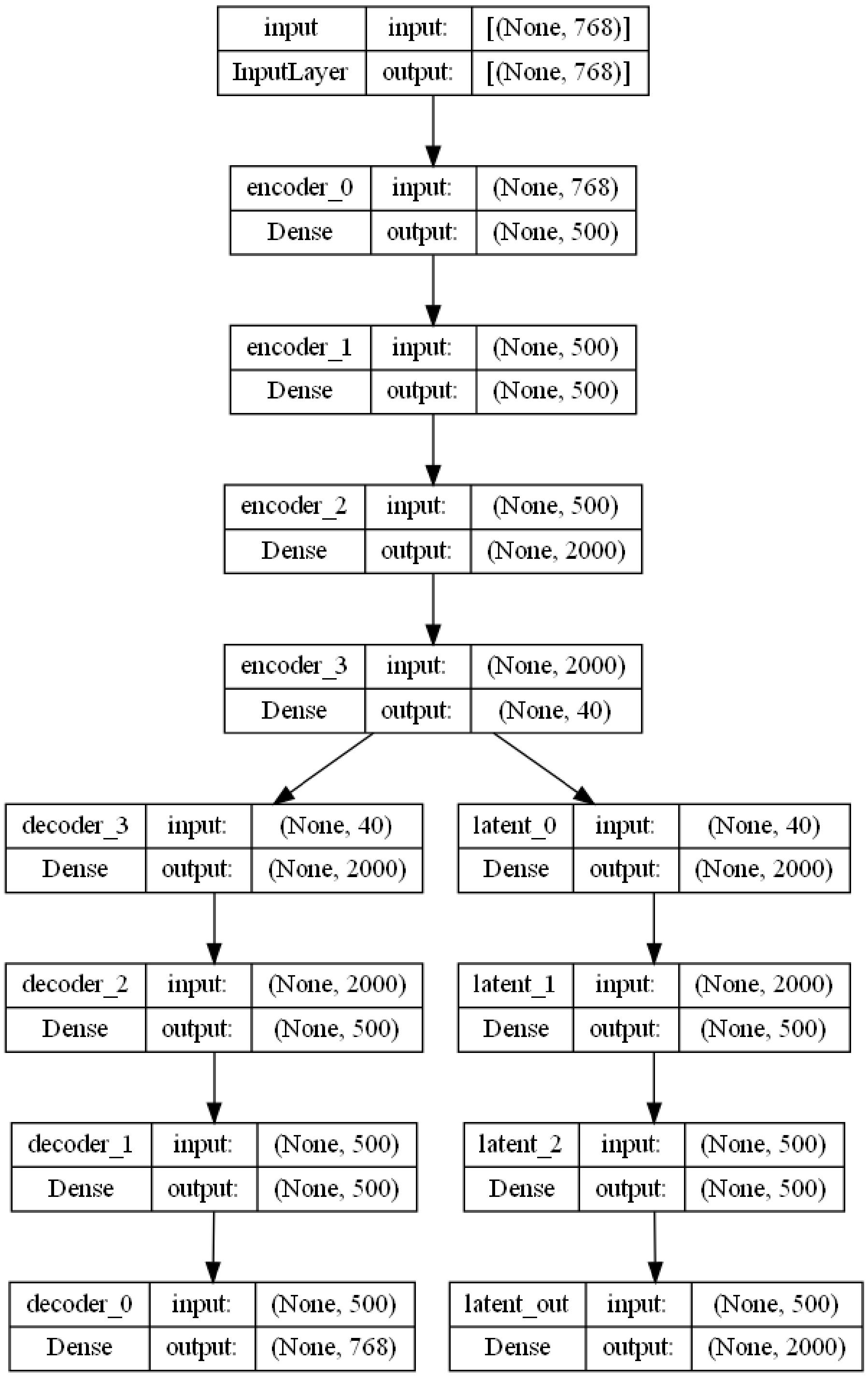

input | input: | [(None, 768)]
InputLayer | output: | [(None, 768)]
encoder_0 | input: | (None, 768)
Dense | output: | (None, 500)
encoder_1 | input: | (None, 500)
Dense | output: | (None, 500)
encoder_2 | input: | (None, 500)
Dense | output: | (None, 2000)
encoder_3 | input: | (None, 2000)
Dense | output: | (None, 40)
decoder_3 | input: | (None, 40)
Dense | output: | (None, 2000)
latent_0 | input: | (None, 40)
Dense | output: | (None, 2000)
decoder_2 | input: | (None, 2000)
Dense | output: | (None, 500)
latent_1 | input: | (None, 2000)
Dense | output: | (None, 500)
decoder_1 | input: | (None, 500)
Dense | output: | (None, 500)
latent_2 | input: | (None, 500)
Dense | output: | (None, 500)
decoder_0 | input: | (None, 500)
Dense | output: | (None, 768)
latent_out | input: | (None, 500)
Dense | output: | (None, 2000)

Figure C.3. Deep Latent Cluster Architecture

Appendix D - Instructions for using the software

To process a set of files:

Make a folder `pdfs` under the top-level project folder.

Place files in that folder. They can be any of the ~2000 types supported by Apache Tika: https://tika.apache.org/0.9/formats.html

Create an environment from requirements.txt

Run the `process_files.py` script from the project `src` folder. For example, to use the approach of the DeepLatent model on the files in `./pdfs`, saving the results to `./results/my-file.text.run`

```
> python process_files.py -a deep_latent -f my-file.txt
```

or, to run from the conda environment `tf-27`

```
> conda run -n tf-27 python process_files.py -a deep_cluster -f out-file.txt
```

```
usage: process_files.py [-h]    --approach {deep_cluster,deep_latent}
                                [--source SOURCE] [--output OUTPUT]
                                [--results RESULTS][--pretrained PRETRAINED]
                                [--filename FILENAME]
                                [--clusterer {kmeans,optics,agg,gmm}]
```

optional arguments:

```
 -h, --help
```
Show this help message and exit
```
 --source SOURCE, -s SOURCE
```
Folder containing files
```
 --output OUTPUT, -o OUTPUT
```
File where output text lines are stored
```
 --results RESULTS, -r RESULTS
```
Folder where model results are stored
```
 --pretrained PRETRAINED, -p PRETRAINED
```
Use the weights files in this result folder from a previous run
```
 --filename FILENAME, -f FILENAME
```
The name of the output text file.

The results folder will be FILENAME with `.run` appended.
```
 --approach {deep_cluster,deep_latent}, -a {deep_cluster,deep_latent}
```

```
                    Use Deep Clustering or learn the Latent Representation
  --clusterer {kmeans,optics,agg,gmm}, -c {kmeans,optics,agg,gmm}
                    Which Clusterer to use:
                          K-means, OPTICS, Gaussian Mixture Model or agglomerative
```

You will now have a file, `lines.txt` in the chosen folder. If needed, copy or move it to the folder `./src/pdfs`, relative to the project folder.

Note that if you run again with the same file name, it will assume the old lines file is correct and not regenerate it, but will re-run the analysis

For show the above information, execute

```
> python process_files.py --help
```

Note the `-a, --approach` argument *must* be supplied to specify the algorithm to be used

D.2 Deep Cluster

To use from code, as per the sample code in `pdfs_notebook.ipynb`, run the following code:

```
from deep_cluster import DeepCluster

dc = DeepCluster(
        run_name='results-file',
        entity_count=15,
        train_size=0,
        num_clusters=25,
        maxiter=2000)
dc.train_and_evaluate_model(10000, verbose=1, folder="pdfs/lines.txt")
```

This will process the text lines found in `pdfs/lines.txt` to generate an analysis in `results/results-file` using the model `DeepCluster`

Execution options include:

- `run_name`: the folder name for saving results
- `train_size`: how many samples to use for training
- `num_clusters`: how many clusters do we want produced?
- `cluster`: the clustering algorithm for seeding
    - `Kmeans` for k-means

  - `GMM` for Gaussian Mixture Methods
  - `OPTICS` for OPTICS
  - `agg` for agglomerative clustering
- `entities`: optional list of spaCy entity labels(strings). Others will be ignored.
- `entity_count`: how many entity classes to find. Don't use with `entities`
- `dims`: optional list of numbers - the sizes of each layer in the encoder.
  - First item must be 768 for BERT embeddings
  - Last item is the size of the bottleneck
- `loss_weights`: optional list of 3 weights [0.0-1.0] for the losses used
  - KL divergence
  - Mean Squared Error
  - Cluster accuracy loss
- `maxiter`: how many iterations to train per epoch
- `verbose`: 0 is quiet, 1 is verbose, 2 is very verbose

D.2 Deep Latent Representation

To use the latent representation network:

```
from deep_latent import DeepLatentCluster

dc = DeepLatentCluster(
    run_name='test-latent-results',
    config={
        'train_size':0,
        "cluster": "Kmeans"
    })
dc.make_model()
dc.train_model()
dc.evaluate_model('test-latent-results', sample_size=4000)
```

This will process the text lines found in `pdfs/lines.txt` to generate an analysis in `results/test-latent-results` using the model `DeepLatentCluster`, using all lines(0) for the training and a random sample of 4000 entities for the analysis and clustering.

DeepLatentCluster has many configuration options that can be supplied to the config param in dictionary form. For a comprehensive list, see the class `__init__` method, but common ones include:

- `train_size`: how many samples to include
- `reconstr_weight`: weight applied to loss from the autoencoder reconstruction
- `latent_weight`: weight applied to loss from the latent representation network
- `cluster`: the clustering algorithm for seeding
    - `Kmeans` for k-means
    - `GMM` for Gaussian Mixture Methods
    - `OPTICS` for OPTICS
    - `agg` for agglomerative clustering
- `opt`: the optimizer function
- `noise_factor`: [0.0 - 1.0] How much gaussian noise to add to training data
- `entity_count`: how many spaCy entity classes
- `num_clusters`: target number of identified cluster, including the spaCy entities
- `max_iter`: how many training iterations per epoch
- `epochs`: how many training epochs
- `tolerance`: [float] the fraction of change of cluster allocations needed to continue training

Results will be stored in `./results/[run_name]` for the run name supplied to the class inititialiser.