Published in Towards Data Science

Will Koehrsen Follow

Oct 4, 2018 · 13 min read · Listen

(Source)

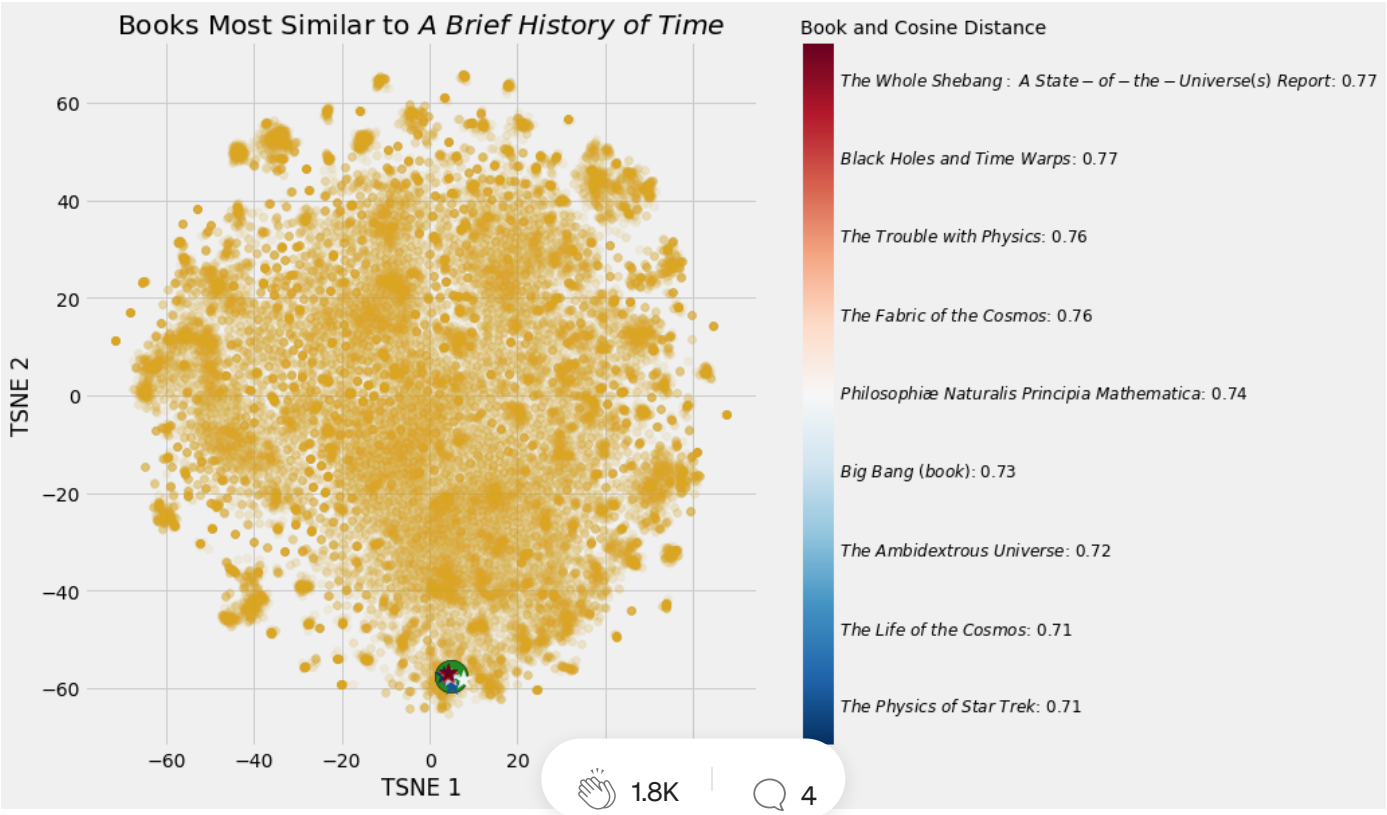
Building a Recommendation System Using Neural Network Embeddings

How to use deep learning and Wikipedia to create a book recommendation system

Deep learning can do some incredible things, but often the uses are obscured in academic papers or require computing resources available only to large corporations

Wikipedia articles on books.

Our recommendation system will be built on the idea that books which link to similar Wikipedia pages are similar to one another. We can represent this similarity and hence make recommendations by *learning embeddings* of books and Wikipedia links using a neural network. The end result is an effective recommendation system and a practical application of deep learning.



Most Similar Books to Stephen Hawking's A Brief History of Time

The complete code for this project is available as a Jupyter Notebook on GitHub. If you don't have a GPU, you can also find the notebook on Kaggle where you can train your neural network with a GPU for free.This article will focus on the implementation, with the concepts of neural network embeddings covered in an earlier article. (To see how to retrieve the data we'll use — all book articles on Wikipedia — take a look at this article.)

Neural Network Embeddings

Embeddings are a way to represent discrete — categorical — variables as continuous vectors. In contrast to an encoding method like one-hot encoding, neural network embeddings are *low-dimensional and learned*, which means they place similar entities closer to one another in the embedding space.

In order to create embeddings, we need a neural network embedding model and a supervised machine learning task. The end outcome of our network will be a representation of each book as a vector of 50 continuous numbers.

While the embeddings themselves are not that interesting — they are just vectors — they can be used for three primary purposes:

1. Finding nearest neighbors in the embedding space
2. As input to a machine learning model
3. Visualization in low dimensions

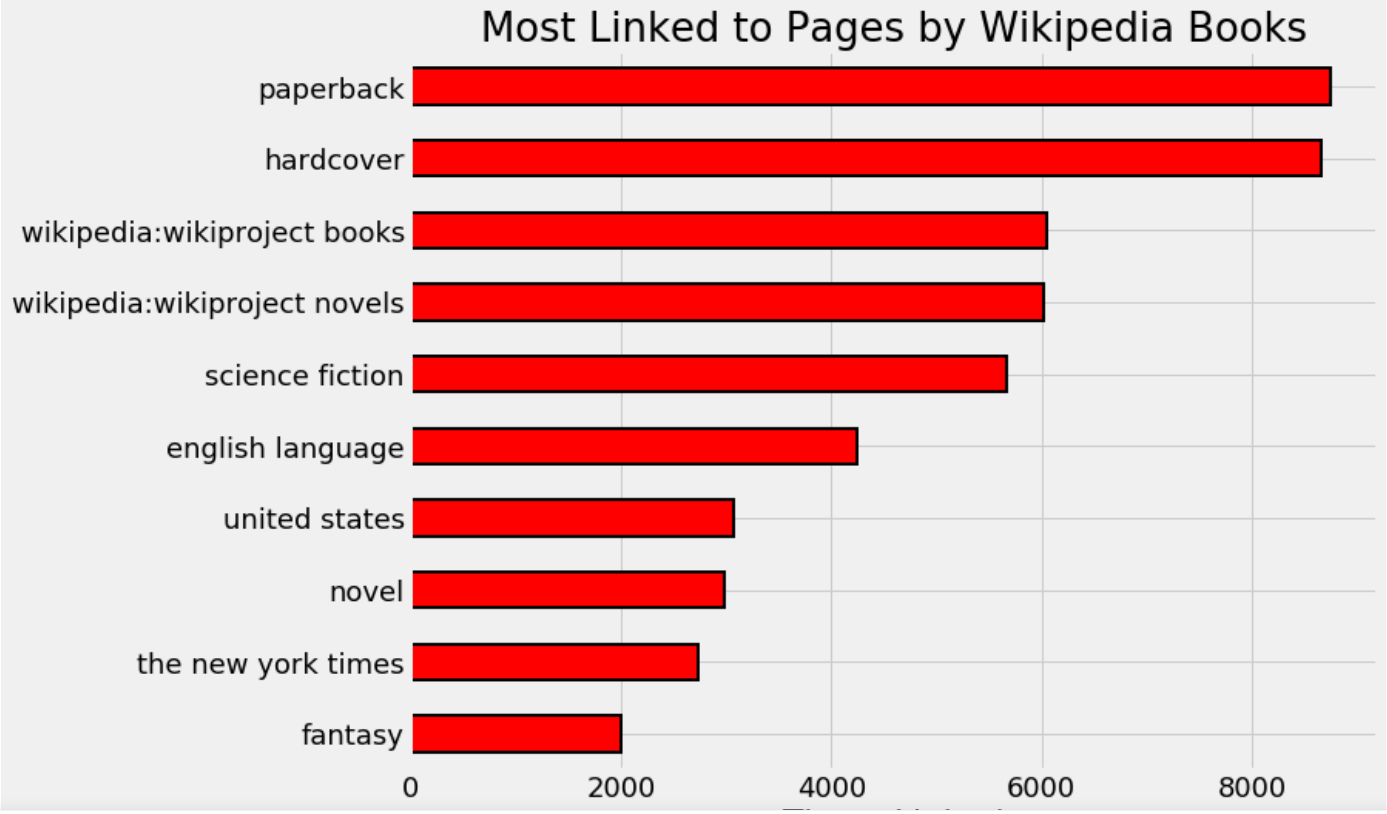
This project covers primarily the first use case, but we'll also see how to create visualizations from the embeddings. Practical applications of neural network embeddings include word embeddings for machine translation and entity embeddings for categorical variables.

Data: All Books on Wikipedia

As usual with a data science project, we need to start with a high-quality dataset. In this article, we saw how to download and process every single article on Wikipedia, searching for any pages about books. We saved the book title, basic information, links


```
Book Title: 'The Better Angels of Our Nature'

Wikilinks:

['Steven Pinker',
  'Nation state',
  'commerce',
  'literacy',
  'Influence of mass media',
  'Rationality',
  "Abraham Lincoln's first inaugural address",
  'nature versus nurture',
  'Leviathan']
```

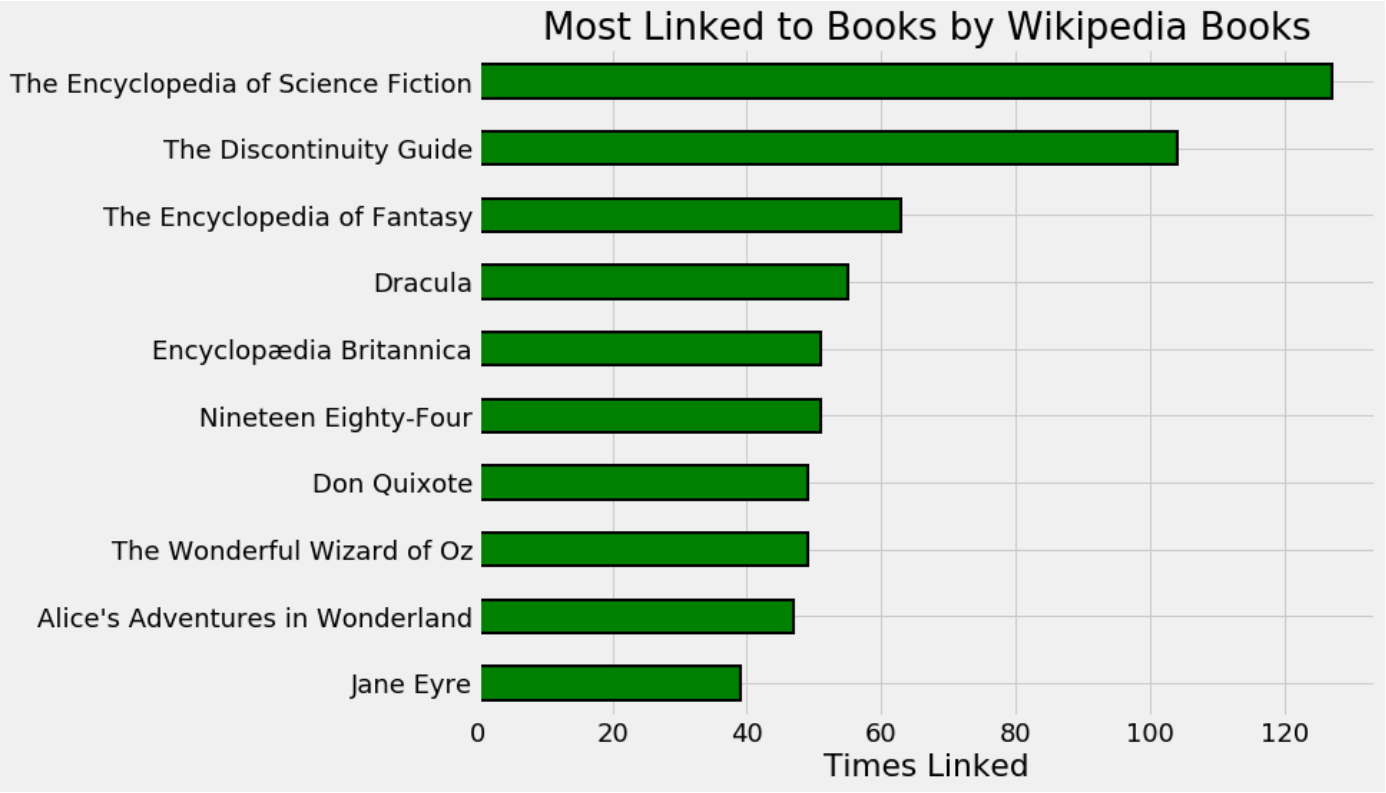
Even when working with a neural network, it's important to explore and clean the data, and in the notebook I make several corrections to the raw data. For example, looking at the most linked pages:



or `hardcover` does not allow us — or a neural network —to figure out the other books it is similar to. Therefore, we can remove these links to help the neural network distinguish between books.

> *Thinking about the end purpose can help in the data cleaning stage and this action alone significantly improves the recommendations.*

Out of pure curiosity, I wanted to find the books most linked to by other books on Wikipedia. These are the 10 "most connected" Wikipedia books:



Books on Wikipedia most often linked to by other books on Wikipedia.

This is a mix of reference works and classic books which makes sense.

After data cleaning, we have a set of **41758** unique wikilinks and **37020** unique books. Hopefully there is a book in there for everyone!

Once we're confident our data is clean, we need to develop a supervised machine

Supervised Learning Task

To *learn meaningful embeddings,* our neural network must be trained to accomplish an objective. Working from the guiding assumption of the project — that similar books link to similar Wikipedia pages — we can formulate the problem as follows: given a (book title, wikilink) pair, determine if the wikilink is present in the book's article.

We won't actually need to give the network the book article. Instead, we'll feed in hundreds of thousands of training examples consisting of book title, wikilink, and label. We give the network some *true examples* — actually present in the dataset — and some false examples, and eventually it learns embeddings to distinguish when a wikilink is on a book's page.

Expressing the supervised learning task is the most important part of this project. Embeddings are learned for a *specific* task and are relevant only to that problem. If our task was to determine which books were written by Jane Austen, then the embeddings would reflect that goal, placing books written by Austen closer together in embedding space. We hope that by training to tell if a book has a certain wikilink on its page, the network learns embeddings that places similar books — in terms of content — closer to one another.

Once we've outlined the learning task, we need to implement it in code. To get started, because the neural network can only accept integer inputs, we create a mapping from each unique book to an integer:

```
# Mapping of books to index and index to books
book_index = {book[0]: idx for idx, book in enumerate(books)}

book_index['Anna Karenina']
```

```
22494
```

We also do the same thing with the links. After this, to create a training set, we make a


```
pairs = []

# Iterate through each book
for book in books:

    title = book[0]
    book_links = book[2]

    # Iterate through wikilinks in book article
    for link in book_links:

        # Add index of book and index of link to pairs
        pairs.extend((book_index[title],
                      link_index[link]))
```

This gives us a total of **772798** true examples that we can sample from to train the model. To generate the false examples — done later — we'll simply pick a link index and book index at random, make sure it's not in the `pairs`, and then use it as a negative observation.

Note about Training / Testing Sets

While using a separate validation and testing set is a **must** for a normal supervised machine learning task, in this case, our primary objective is **not** to make the most accurate model, but to generate embeddings. The prediction task is just the means by which we train our network for those embeddings. At the end of training, we are not going to be testing our model on new data, so we don't need to evaluate the performance or use a validation set to prevent overfitting. To get the best embeddings, we'll use all examples for training.

Embedding Model

Although neural network embeddings sound technically complex, they are relatively

The embedding model has 5 layers:

1. **Input:** parallel inputs for the book and link
2. **Embedding:** parallel length 50 embeddings for the book and link
3. **Dot:** merges embeddings by computing dot product
4. **Reshape:** needed to shape the dot product to a single number
5. **Dense:** one output neuron with sigmoid activation

In an embedding neural network, the embeddings are the parameters — weights — of the neural network that are adjusted during training in order to minimize loss on the objective. The neural network takes in a book and a link as integers and outputs a prediction between 0 and 1 that is compared to the true value. The model is compiled with the `Adam` optimizer (a variant on Stochastic Gradient Descent) which, during training, alters the embeddings to minimize the `binary_crossentropy` for this binary classification problem.

Below is the code for the complete model:

```
from keras.layers import Input, Embedding, Dot, Reshape, Dense
from keras.models import Model

def book_embedding_model(embedding_size = 50, classification = False):
    """Model to embed books and wikilinks using the Keras functional API.
       Trained to discern if a link is present in on a book's page"""

    # Both inputs are 1-dimensional
    book = Input(name = 'book', shape = [1])
    link = Input(name = 'link', shape = [1])

    # Embedding the book (shape will be (None, 1, 50))
    book_embedding = Embedding(name = 'book_embedding',
                               input_dim = len(book_index)
```

```
                                input_dim = len(link_index),
                                output_dim = embedding_size)(link)

    # Merge the layers with a dot product along the second axis
    # (shape will be (None, 1, 1))
    merged = Dot(name = 'dot_product', normalize = True,
                 axes = 2)([book_embedding, link_embedding])

    # Reshape to be a single number (shape will be (None, 1))
    merged = Reshape(target_shape = [1])(merged)

    # Squash outputs for classification
    out = Dense(1, activation = 'sigmoid')(merged)
    model = Model(inputs = [book, link], outputs = out)

    # Compile using specified optimizer and loss
    model.compile(optimizer = 'Adam', loss = 'binary_crossentropy',
                  metrics = ['accuracy'])

    return model
```

book_embedding_model.py hosted with ❤ by GitHub view raw

This same framework can be used for many embedding models. The important point to understand is that the embeddings are the *model parameters (weights)* and also the final result we want. We don't really care if the model is accurate, what we want is relevant embeddings.

> *We're used to the weights in a model being a means to make accurate predictions, but in an embedding model, the weights are the objective and the predictions are a means to learn an embedding.*

There are almost 4 million weights as shown by the model summary:

```

_________________________________________________________________________________________________
Layer (type)                 Output Shape         Param #     Connected to
-------------------------------------------------------------------------------------------------
```

```
10  __________________________________________________________________________________________________
11  link_embedding (Embedding)      (None, 1, 50)        2087900     link[0][0]
12  __________________________________________________________________________________________________
13  dot_product (Dot)               (None, 1, 1)         0           book_embedding[0][0]
14                                                                   link_embedding[0][0]
15  __________________________________________________________________________________________________
16  reshape_1 (Reshape)             (None, 1)            0           dot_product[0][0]
17  ==================================================================================================
18  Total params: 3,938,900
19  Trainable params: 3,938,900
20  Non-trainable params: 0
```

book_embedding_model_summary.txt hosted with ❤ by GitHub view raw

With this approach, we'll get embeddings not only for books, but also for links which means we can compare all Wikipedia pages that are linked to by books.

Generating Training Samples

Neural networks are batch learners because they are trained on a small set of samples — observations — at a time over many rounds called epochs. A common approach for training neural networks is to use a generator. This is a function that `yields` (not `returns`) batches of samples so the entire result is not held in memory. Although it's not an issue in this problem, the benefit of a generator is that large training sets do not need to all be loaded into memory.

Our generator takes in the training `pairs`, the number of positive samples per batch (`n_positive`), and the ratio of negative:positive samples per batch (`negative_ratio`). The generator yields a new batch of positive and negative samples each time it is called. To get positive examples, we randomly sample true pairs. For the negative examples, we randomly sample a book and link, make sure this pairing is not in the true pairs, and then add it to the batch.

```
random.seed(100)

def generate_batch(pairs, n_positive = 50, negative_ratio = 1.0):
    """Generate batches of samples for training.
       Random select positive samples
       from pairs and randomly select negatives."""

    # Create empty array to hold batch
    batch_size = n_positive * (1 + negative_ratio)
    batch = np.zeros((batch_size, 3))

    # Continue to yield samples
    while True:
        # Randomly choose positive examples
        for idx, (book_id, link_id) in enumerate(random.sample(pairs, n_positive)):
            batch[idx, :] = (book_id, link_id, 1)
        idx += 1

        # Add negative examples until reach batch size
        while idx < batch_size:

            # Random selection
            random_book = random.randrange(len(books))
            random_link = random.randrange(len(links))

            # Check to make sure this is not a positive example
            if (random_book, random_link) not in pairs_set:

                # Add to batch and increment index
                batch[idx, :] = (random_book, random_link, neg_label)
                idx += 1

        # Make sure to shuffle order
        np.random.shuffle(batch)
        yield {'book': batch[:, 0], 'link': batch[:, 1]}, batch[:, 2]
```

generate_book_batch.py hosted with ❤ by GitHub view raw

Each time we call `next` on the generator, we get a new training batch.

```
({'book': array([ 6895., 29814., 22162.,  7206., 25757., 28410.]),
  'link': array([  260., 11452.,  5588., 34924., 22920., 33217.])},
  array([ 1., -1.,  1., -1., -1., -1.]))
```

With a supervised task, a training generator, and an embedding model, we're ready to learn book embeddings.

Training Model

There are a few training parameters to select. The first is the number of positive examples in each batch. Generally, I try to start with a small batch size and increase it until performance starts to decline. Also, we need to choose the number of negative examples trained for each positive example. I'd recommend experimenting with a few options to see what works best. Since we're not using a validation set to implement early stopping, I choose a number of epochs beyond which the training loss does not decrease.

```
n_positive = 1024

gen = generate_batch(pairs, n_positive, negative_ratio = 2)

# Train
h = model.fit_generator(gen, epochs = 15,
                        steps_per_epoch = len(pairs) // n_positive)
```

(If the training parameters seem arbitrary, in a sense they are, but based on best practices outlined in *Deep Learning*. Like most aspects of machine learning, training a neural network is largely empirical.)

Once the network is done training, we can extract the embeddings.

```
book_weights = book_layer.get_weights()[0]
```

Applying Embeddings: Making Recommendations

The embeddings themselves are fairly uninteresting: they are just 50-number vectors for each book and each link:

title	dim-0	dim-1	dim-2	dim-3	dim-4	...	dim-45	dim-46	dim-47	dim-48	dim-49
War and Peace	-0.279165	-0.107367	0.114153	0.143709	-0.141921	...	-0.067178	0.230711	-0.230550	0.199285	-0.099167
Anna Karenina	-0.248443	-0.000578	0.150472	0.151845	0.000908	...	-0.141615	0.178011	-0.230794	0.042102	-0.189196
The Hitchhiker's Guide to the Galaxy (novel)	-0.190761	-0.060406	0.115548	-0.249868	-0.120824	...	-0.038944	0.084992	-0.047035	-0.054157	-0.209883

What War and Peace Looks Like as a Vector.

However, we can use these vectors for two different purposes, the first of which is to make our book recommendation system. To find the closest book to a query book in the embedding space, we take the vector for that book and find its dot product with the vectors for all other books. If our embeddings are normalized, then the dot product between the vectors represents the cosine similarity, ranging from -1, most dissimilar, to +1, most similar.

Querying the embeddings for the classic *War and Peace* by Leo Tolstoy yields:

```
Books closest to War and Peace.

Book: Anna Karenina                Similarity: 0.92
Book: The Master and Margarita     Similarity: 0.92
Book: Demons (Dostoevsky novel)    Similarity: 0.91
Book: The Idiot                    Similarity: 0.9
Book: Crime and Punishment         Similarity: 0.9
```

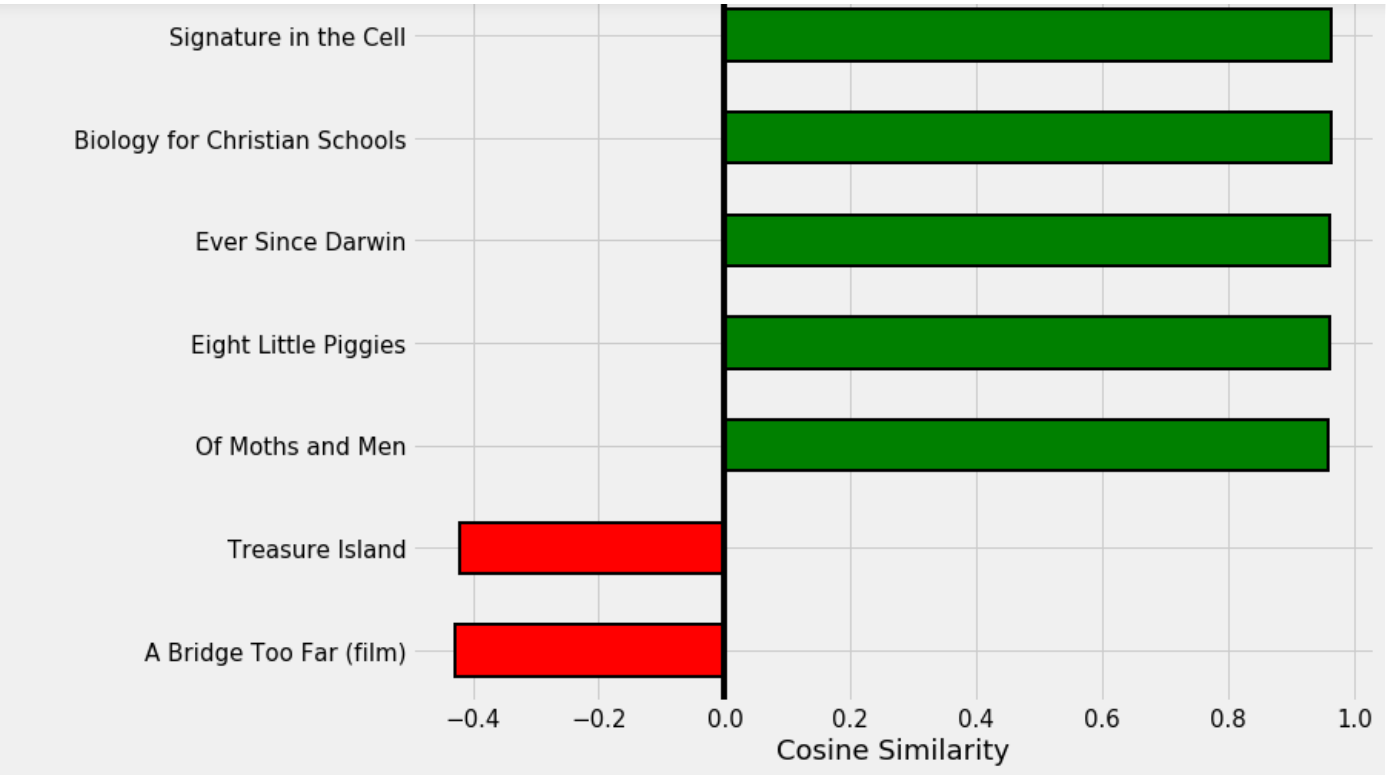
The recommendations make sense! These are all classic Russian novels. Sure we could

```
Books closest to The Fellowship of the Ring.

Book: The Return of the King       Similarity: 0.96
Book: The Silmarillion             Similarity: 0.93
Book: Beren and Lúthien            Similarity: 0.91
Book: The Two Towers               Similarity: 0.91
```

In addition to embedding the books, we also embedded the links which means we can find the most similar links to a given Wikipedia page:

```
Pages closest to steven pinker.

Page: the blank slate           Similarity: 0.83
Page: evolutionary psychology   Similarity: 0.83
Page: reductionism              Similarity: 0.81
Page: how the mind works        Similarity: 0.79
```

Currently, I'm reading a fantastic collection of essays by Stephen Jay Gould called *Bully for Brontosaurus*. What should I read next?

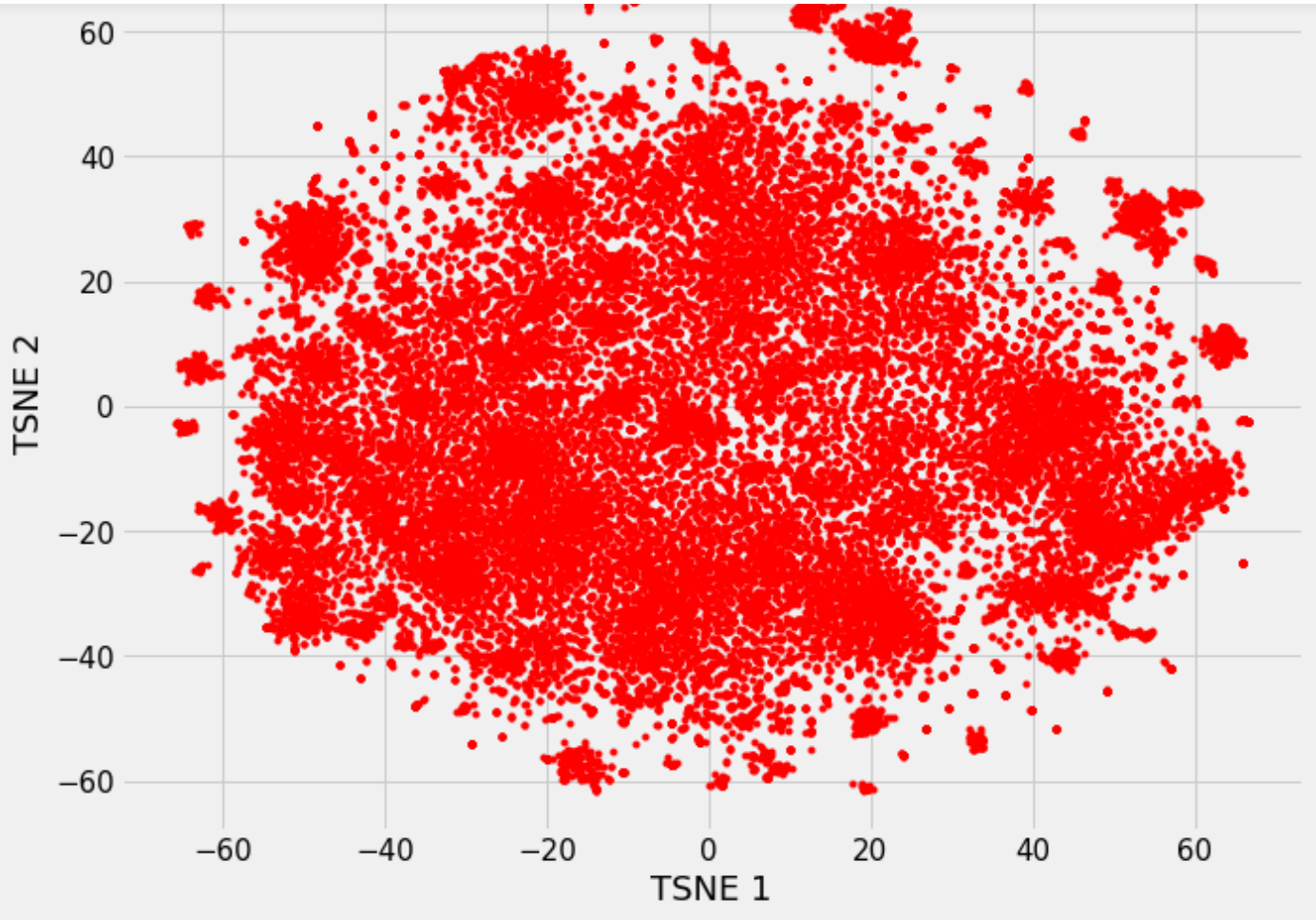



Recommendations for my next book.

Visualizations of Embeddings

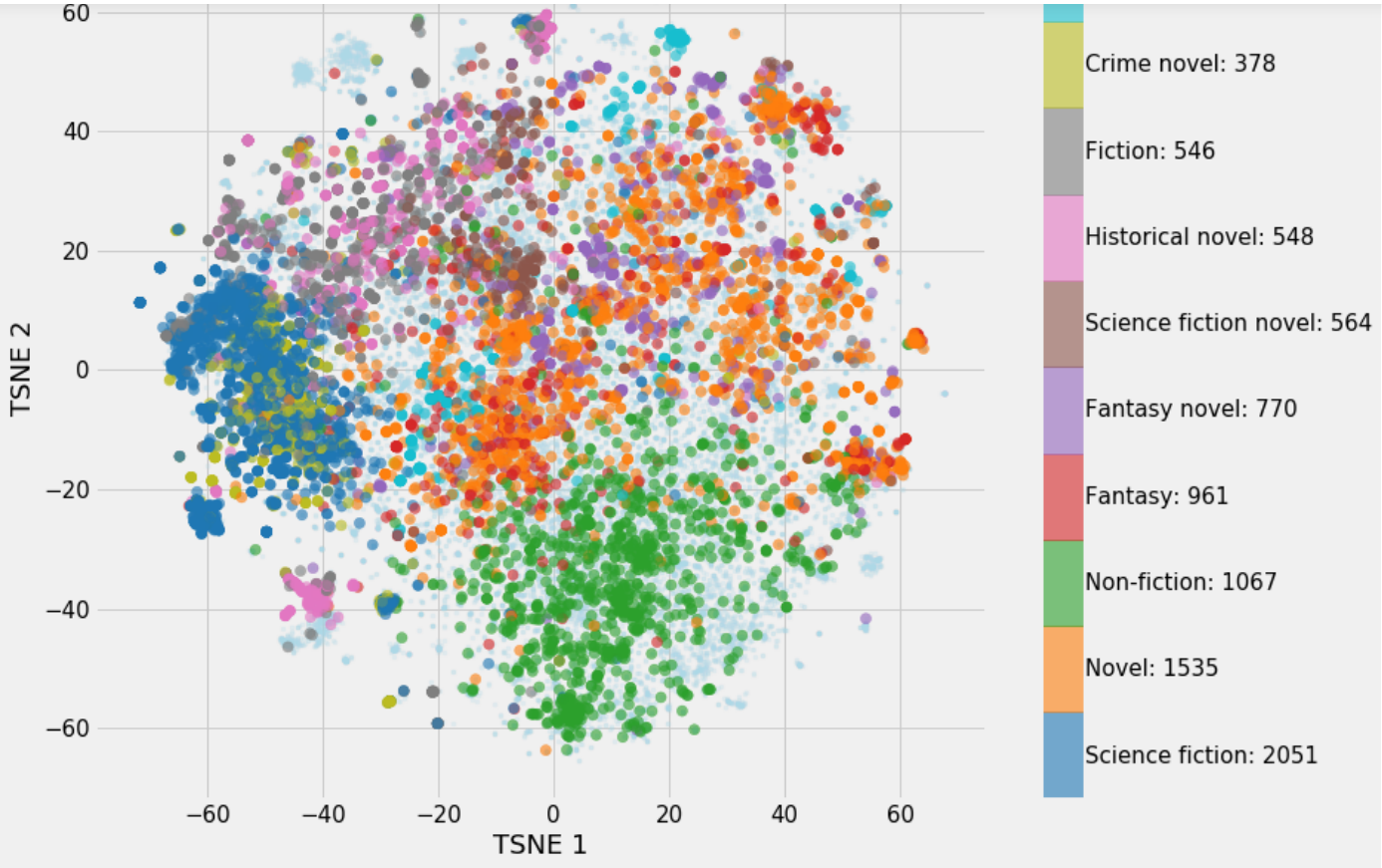
One of the most intriguing aspects of embeddings are that they can be used to visualize concepts such as *novel* or *nonfiction* relative to one another. This requires a further dimension reduction technique to get the dimensions to 2 or 3. The most popular technique for reduction is another embedding method: t-Distributed Stochastic Neighbor Embedding (TSNE).

Starting with the 37,000-dimensional space of all books on Wikipedia, we map it to 50 dimensions using embeddings, and then to just 2 dimensions with TSNE. This results in the following image:



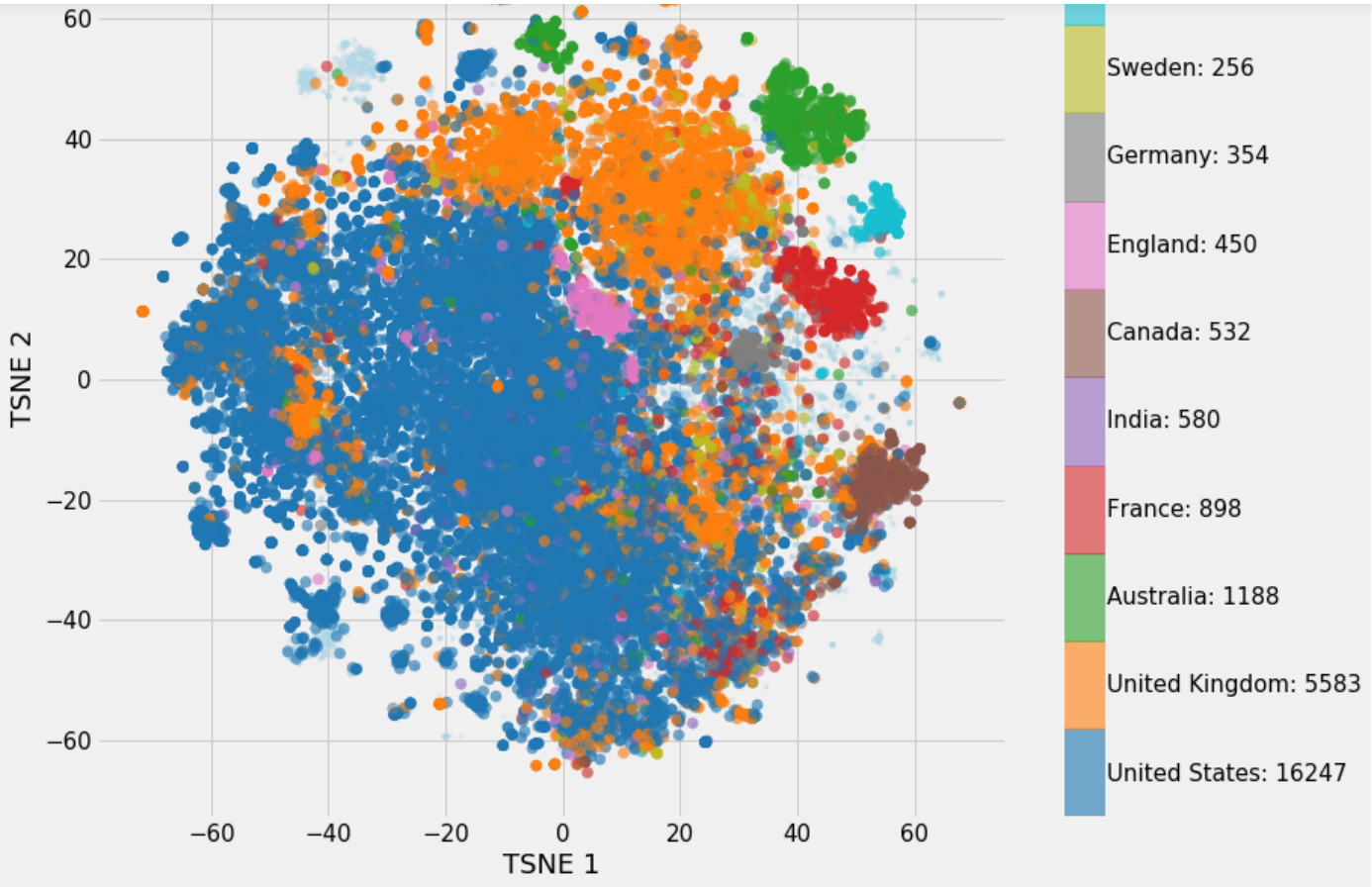
Embeddings of all books on Wikipedia.

By itself this image is not that illuminating, but once we start coloring it by book characteristics, we start to see clusters emerge:



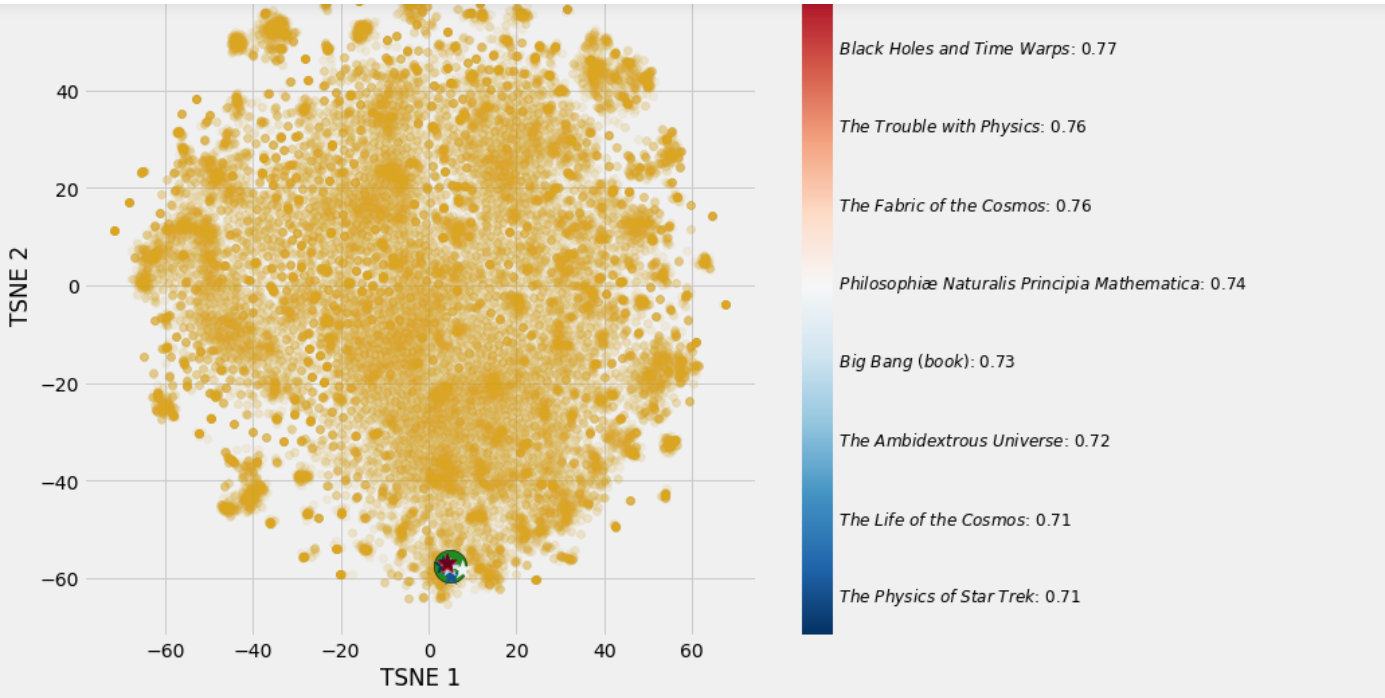
There are some definite clumps (only the top 10 genres are highlighted) with non-fiction and science fiction having distinct sections. The novels seem to be all over the place which makes sense given the diversity in novel content.

We can also do embeddings with the country:



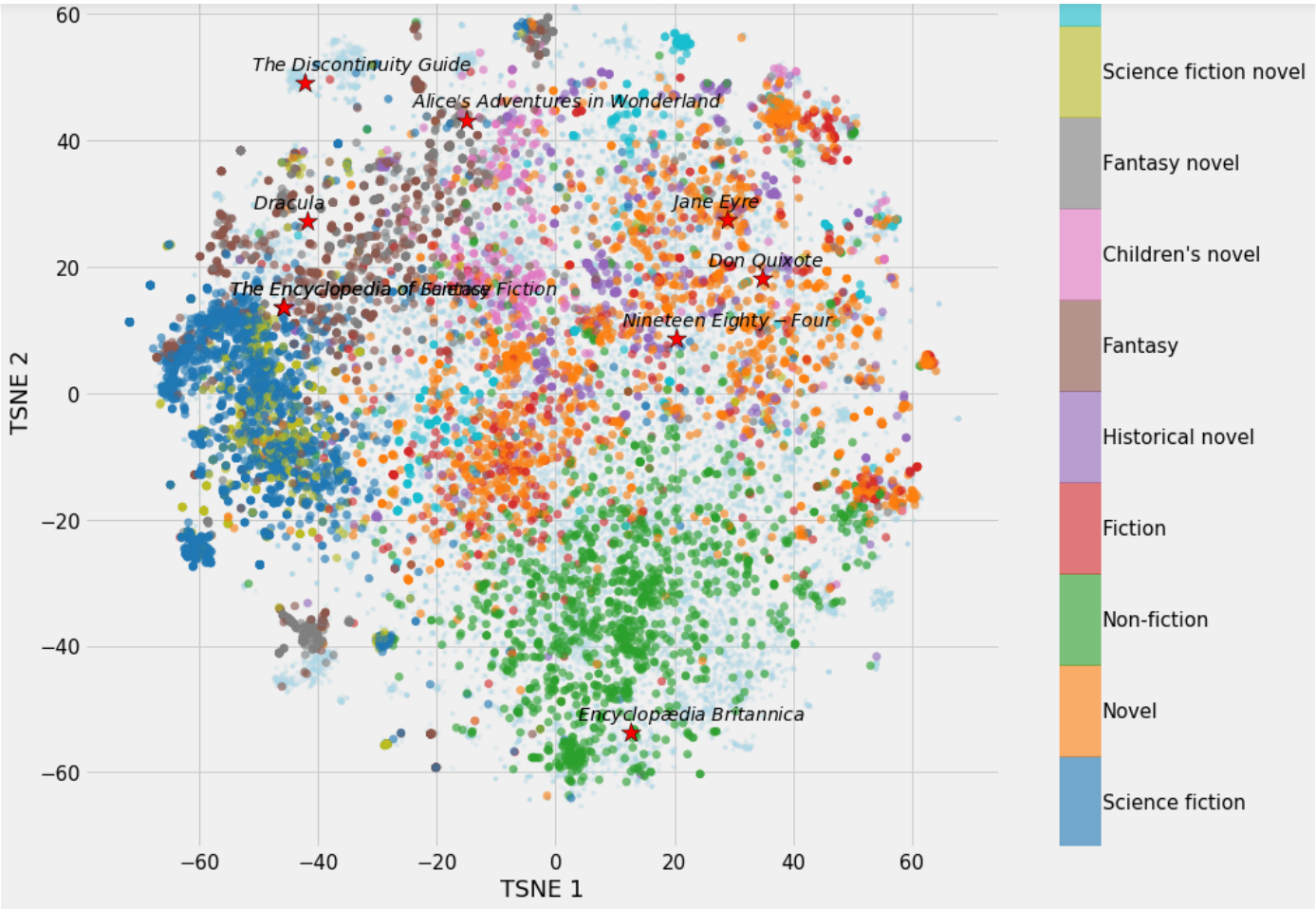
I was a little surprised at how distinctive the countries are! Evidently Australian books are quite unique.

Furthermore, we can highlight certain books in the Wikipedia map:



The corner of Wikipedia with books about the entire Universe

There are a lot more visualizations in the notebook and you make your own. I'll leave you with one more showing the 10 "most connected" books:

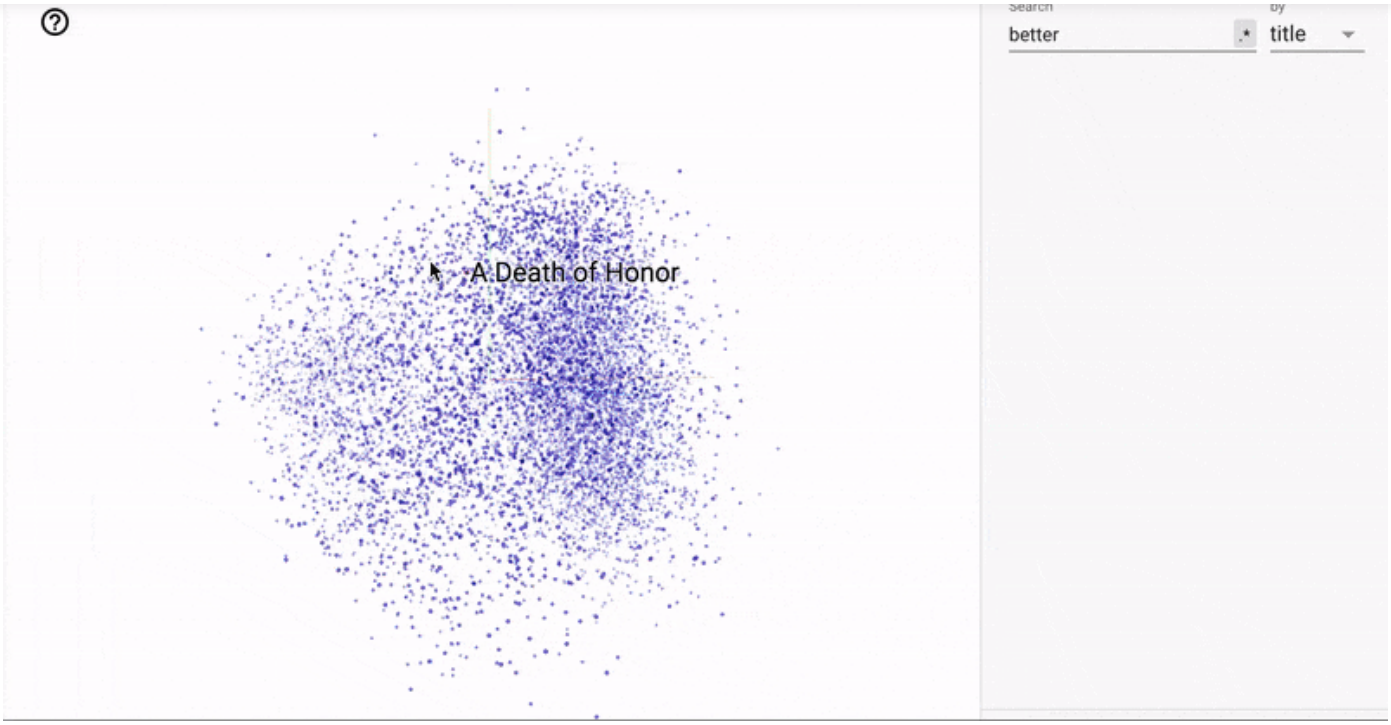


Book embeddings with 10 most linked to books and genres.

One thing to note about TSNE is that it tries to preserve distances between vectors in the original space, but because it reduces the number of dimensions, it may distort the original separation. Therefore, books that are close to one another in the 50-dimensional embedding space may not be closest neighbors in the 2-dimensional TSNE embedding.

Interactive Visualizations

These visualizations are pretty interesting, but we can make stunning interactive figures with TensorFlow's projector tool specifically designed for visualizing neural network embeddings. I plan on writing an article on how to use this tool, but for now here are some of the results:



Interactive visualizations made with projector

To explore a sample of the books interactively, head here.

Potential Other Projects

Data science projects aren't usually invented entirely on their own. A lot of the projects I work on are ideas from other data scientists that I adapt, improve, and build on to make a unique project. (This project was inspired by a similar project for movie recommendations in the Deep Learning Cookbook.)

With that attitude in mind, here are a few ways to build on this work:

1. Create embeddings using the *external links* instead of wikilinks. These are to web pages outside Wikipedia and might produce different embeddings.
2. Use the embeddings to train a supervised machine learning model to predict the book characteristics which include genre, author, and country.
3. Pick a topic category on Wikipedia and create your own recommendation system. You could use people, landmarks, or even historical events. You can use this

about it!

Conclusions

Neural network embeddings are a method to represent discrete categorical variables as continuous vectors. As a *learned low-dimensional* representation, they are useful for finding similar categories, as input into a machine learning model, or to visualize maps of concepts. In this project, we used neural network embeddings to create an effective book recommendation system built on the idea that books which link to similar pages are similar to each other.

The steps for creating neural network embeddings are:

1. Gather data. Neural networks require many training examples.
2. Formulate a supervised task to learn embeddings that reflect the problem.
3. Build and train an embedding neural network model.
4. Extract the embeddings for making recommendations and visualizations.

The details can be found in the notebook and I'd encourage anyone to build on this project. While deep learning may seem overwhelming because of technical complexity or computational resources, this is one of many applications that can be done on a personal computer with a limited amount of studying. Deep learning is a constantly evolving field, and this project is a good way to get started by building a useful system. And, when you're not studying deep learning, now you know what you should be reading!

As always, I welcome feedback and constructive criticism. I can be reached on Twitter