CSE 400/691 Final Project Report

RECOMMENDATION SYSTEMS

By,

Arjun Kalyanasundaram

Bhushan Patel

Gurinder Singh

Nikhil Shetty

TABLE OF CONTENT

INTRODUCTION

In this project, we will build a book recommendation system based on a simple principle: books with Wikipedia pages that link to similar Wikipedia pages are similar to each other. In order to create this representation of similar books, we'll use the concept of neural network entity embeddings, mapping each book and each Wikipedia link (Wikilink) to a 50-number vector.

The idea of entity embeddings is to map high-dimensional categorical variables to a low-dimensional *learned* representation that *places similar entities closer together in the embedding space*. If we were to one-hot-encode the books (another representation of categorical data) we would have a 37,000 dimension vector for each book, with a single 1 indicating the book. In a one-hot encoding, similar books would not be "closer" to one another. By training a neural network to learn entity embeddings, we not only get a reduced dimension representation of the books, but we also get a representation that *keeps similar books closer to each other*.

Therefore, the basic approach for a recommendation system is to create entity embeddings of all the books, and then for any book, find the closest other books in the embedding space. From the previous systems, we have access to every single book article on Wikipedia, which will let us create an effective recommendation system.

APPLICATION

Approach

To create entity embeddings, we need to build an embedding neural network and train it on a supervised machine learning task that will result in similar books (and similar links) having closer representations in embedding space. The parameters of the neural network - the weights - are the embeddings, and so during training, these numbers are adjusted to minimize the loss on the prediction problem. In other words, the network tries to accurately complete the task by changing the representations of the books and the links.

Once we have the embeddings for the books and the links, we can find the most similar book to a given book by computing the distance between the embedded vector for that book and all the other book embeddings. We'll use the cosine distance which measures the angle between two vectors as a measure of similarity (another valid option is the Euclidean distance). We can also do the same with the links, finding the most similar page to a given page. (I use links and wikilinks interchangeably in this notebook). The steps we will follow are:

1. Load in data and clean
2. Prepare data for supervised machine learning task
3. Build the entity embedding neural network
4. Train the neural network on the prediction task
5. Extract embeddings and find the most similar books and wikilinks
6. Visualize the embeddings using dimension reduction techniques

Supervised Machine Learning Task: Map Books to Links

For our machine learning task, we'll set up the problem as identifying whether or not a particular link was present in a book article. The training examples will consist of (book, link) pairs, with some pairs of true examples - actually in the data - and other negative examples - do not occur in the data. It will be the network's job to adjust the entity embeddings of the books and the links in order to accurately make this classification. Although we are training for a supervised machine learning task, our end objective is not to make accurate predictions on new data, but learn the best entity embeddings, so we do not use a validation or testing set. We use the prediction problem as a means to an end rather than the final outcome.

Input Data: Read in Data

The data is stored as json with a line for every book. This data contains every single book article on Wikipedia which was parsed in the Downloading and Parsing Wikipedia Data Notebook.

```
In [1]: from IPython.core.interactiveshell import InteractiveShell
# Set shell to show all lines of output
InteractiveShell.ast_node_interactivity = 'all'
```

```
In [2]:from keras.utils import get_file
x = get_file('found_books_filtered.ndjson', 'https://raw.githubusercontent.com/WillKoehrsen/wikipedia-data-science/master/data/found_books_filtered.ndjson')
import json
books = []
with open(x, 'r') as fin:
    # Append each line to the books
    books = [json.loads(l) for l in fin]
# Remove non-book articles
books_with_wikipedia = [book for book in books if 'Wikipedia:' in book[0]]
books = [book for book in books if 'Wikipedia:' not in book[0]]
print(f'Found {len(books)} books.')
```

```
---------------------------------------------------------------------------
Downloading data from https://raw.githubusercontent.com/WillKoehrsen/wikipedia-data-science/master/data/found_books_filtered.ndjson
58933248/58925764 [==============================] - 1s 0us/step
Found 37020 books.
```

Data Cleaning

There are a few articles that were caught which are clearly not books (feel free to check out these articles yourself).

```
In [3]: [book[0] for book in books_with_wikipedia][:5]
```

```
---------------------------------------------------------------------------
Out[3]: ['Wikipedia:Wikipedia Signpost/2014-06-25/Recent research',
 'Wikipedia:New pages patrol/Unpatrolled articles/December 2010',
 'Wikipedia:Templates for discussion/Log/2012 September 23',
 'Wikipedia:Articles for creation/Redirects/2012-10',
 'Wikipedia:Templates for discussion/Log/2012 October 4']
```

Each legitimate book contains the title, the information from the Infobox book template, the internal wikipedia links, the external links, the date of last edit, and the number of characters in the article (a rough estimate of the length of the article).

```
In [4]: n = 21
books[n][0], books[n][1], books[n][2][:5], books[n][3][:5], books[n][3][:5], books[n][4], books[n][5]
```

```
--------------------------------------------------------------------------------
Out[4]: ('Limonov (novel)',
 {'name': 'Limonov',
  'author': 'Emmanuel Carrère',
  'translator': 'John Lambert',
  'country': 'France',
  'language': 'French',
  'publisher': 'P.O.L.',
  'pub_date': '2011',
  'english_pub_date': '2014',
  'pages': '488',
  'isbn': '978-2-8180-1405-9'},
 ['Emmanuel Carrère',
  'biographical novel',
  'Emmanuel Carrère',
  'Eduard Limonov',
  'Prix de la langue française'],
 ['http://www.lefigaro.fr/flash-actu/2011/10/05/97001-20111005FILWWW00615-le-prix-de-la-langue-francaise-a-
e-carrere.php',
  'http://www.lexpress.fr/culture/livre/emmanuel-carrere-prix-renaudot-2011_1046819.html',
  'http://limonow.de/carrere/index.html',
  'http://www.tout-sur-limonov.fr/222318809'],
 ['http://www.lefigaro.fr/flash-actu/2011/10/05/97001-20111005FILWWW00615-le-prix-de-la-langue-francaise-a-
e-carrere.php',
  'http://www.lexpress.fr/culture/livre/emmanuel-carrere-prix-renaudot-2011_1046819.html',
  'http://limonow.de/carrere/index.html',
  'http://www.tout-sur-limonov.fr/222318809'],
 '2018-08-18T02:03:21Z',
 1437)
```

We will only use the wikilinks, which are saved as the third element for each book.

Neural Network

Different Layers:

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
book (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
link (InputLayer)               (None, 1)            0
__________________________________________________________________________________________________
book_embedding (Embedding)      (None, 1, 50)        1851000     book[0][0]
__________________________________________________________________________________________________
link_embedding (Embedding)      (None, 1, 50)        2087900     link[0][0]
__________________________________________________________________________________________________
dot_product (Dot)               (None, 1, 1)         0           book_embedding[0][0]
                                                                 link_embedding[0][0]
__________________________________________________________________________________________________
reshape_2 (Reshape)             (None, 1)            0           dot_product[0][0]
==================================================================================================
Total params: 3,938,900
Trainable params: 3,938,900
Non-trainable params: 0
__________________________________________________________________________________________________
```

Neural Network Embeddings

Neural Network embeddings have proven to be very powerful concepts both for modeling language and for representing categorical variables. For example, the Word2Vec word embeddings map a word to a vector based on training a neural network on millions of words. These embeddings can be used in any supervised model because they are just numerical representations of categorical variables. Much as we one-hot-encode categorical variables to use them in a random forest for a supervised task, we can also use entity embeddings to include categorical variables in a model. The embeddings are also useful because we can find entities that are close to one another in embedding space which might - as in a book recommendation system - allow us to find the most similar categories among tens of thousands of choices.

We can also use the Entity Embeddings to visualize words or categorical variables, such as creating a map of all books on Wikipedia. The entity embeddings typically are still high-dimensional - we'll use 50 numbers for each entity - so we need to use a dimension reduction technique such as TSNE or UMAP to visualize the embeddings in lower dimensions. (These are both manifold embedding methods so in effect we will embed the embeddings for visualization!) We'll take a look at doing this at the end of the notebook and later will upload the embeddings into an application custom-built for this purpose (projector.tensorflow.org). Entity embeddings are becoming more widespread thanks to the ease of development of neural networks in Keras and are a useful approach when we want to represent categorical variables with vectors that place similar categories close to one another. Other approaches for encoding categorical variables do not represent similar entities as being closer to one another, and entity embedding is a *learning-based method* for this important task.

Map Books to Integers

First, we want to create a mapping of book titles to integers. When we feed books into the embedding neural network, we will have to represent them as numbers, and this mapping will let us keep track of the books. We'll also create the reverse mapping, from integers back to the title.

```
In [5]: book_index = {book[0]: idx for idx, book in enumerate(books)}
index_book = {idx: book for book, idx in book_index.items()}
book_index['Anna Karenina']
Index_book[22494]
```

Out[5]:22494
Out[5]:'Anna Karenina'

Exploring Wikilinks

Although it's not our main focus, we can do a little exploration. Let's find the number of unique Wikilinks and the most common ones. To create a single list from a list of lists, we can use the itertools chain method.

```
In [6]: from itertools import chain

wikilinks = list(chain(*[book[2] for book in books]))
print(f"There are {len(set(wikilinks))} unique wikilinks.")
```

There are 311276 unique wikilinks.

```
In [7]: wikilinks_other_books = [link for link in wikilinks if link in book_index.keys()]
print(f"There are {len(set(wikilinks_other_books))} unique wikilinks to other books.")
```

There are 17032 unique wikilinks linked to other books.\

Wikilinks to Index

As with the books, we need to map the Wikilinks to integers. We'll also create the reverse mapping.

```
In [16]: link_index = {link: idx for idx, link in enumerate(links)}
index_link = {idx: link for link, idx in link_index.items()}

link_index['the economist']
index_link[300]
print(f'There are {len(link_index)} wikilinks that will be used.')
```

Out[16]: 300
Out[16]:'the economist'
There are 41758 wikilinks that will be used.

LEARNING & TESTING

Supervised Machine Learning Task

Now that we have clean data, we'll move on to the second step: developing a supervised machine learning task to train an embedding neural network. As a reminder, we'll state the problem as: given a book title and a link, identify if the link is in the book's article.

Build a Training Set

In order for any machine learning model to learn, it needs a training set. We are going to treat this as a supervised learning problem: given a pair (book, link), we want the neural network to learn to predict whether this is a legitimate pair - present in the data - or not.

To create a training set, for each book, we'll iterate through the wikilinks on the book page and record the book title and each link as a tuple. The final pairs list will consist of tuples of every (book, link) pairing on all of Wikipedia.

```
In [17]:pairs = []
# Iterate through each book
for book in books:
    # Iterate through the links in the book
    pairs.extend((book_index[book[0]], link_index[link.lower()]) for link in book[2] if link.lower() in links)
len(pairs), len(links), len(books)
pairs[5000]
```

Out[17]: (772798, 41758, 37020)
Out[17]: (321, 232)

We now have over 770,000 positive examples on which to train! Each pair represents one Wikilink for one book. Let's look at a few examples.

```
In [18]: index_book[pairs[5000][0]], index_link[pairs[5000][1]]
```

Out[18]:('Slaves in the Family', 'category:american biographies')

```
In [19]:index_book[pairs[900][0]], index_link[pairs[900][1]]
```

Out[19]:('The Man Who Watched the Trains Go By (novel)', 'category:belgian novels adapted into films')

Later on we'll create the negative examples by randomly sampling from the links and the books and making sure the resulting pair is not in pairs.

```
In [20]:pairs_set = set(pairs)
```

Just for fun, let's look at the (book, link) pairs that are represented most often in the data.

```
In [21]: x = Counter(pairs)
sorted(x.items(), key = lambda x: x[1], reverse = True)[:5]
```

Out[21]: [((13337, 31111), 85),
((31899, 65), 77),
((25899, 8850), 61),
((1851, 2629), 57),
((25899, 30465), 53)]

```
In [22]: index_book[13337], index_link[31111]
index_book[31899], index_link[65]
```

```
index_book[25899], index_link[30465]
--------------------------------------------------------------------------------
Out[22]: ("France's Songs of the Bards of the Tyne - 1850", 'joseph philip robson')
Out[22]: ('The Early Stories: 1953–1975', 'the new yorker')
Out[22]: ('Marthandavarma (novel)', 'kerala sahitya akademi')
--------------------------------------------------------------------------------
```

There's nothing wrong with books that link to the same page many times. They are just more likely to be trained on since there are more of them.

Note about Training / Testing Set

To compute the embeddings, we are not going to create a separate validation or testing set. While this 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 the best embeddings. The prediction task is just the method through which we train our network to make the 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. Instead of testing on new data, we'll look at the embeddings themselves to see if books that we think are similar have embeddings that are close to each other.

If we kept a separate validation/testing set, then we would be limiting the amount of data that our network can use to train. This would result in less accurate embeddings. Normally with any supervised model, we need to be concerned about overfitting, but again, because we do not need our model to generalize to new data and our goal is the embeddings, we will make our model as effective as possible by using all the data for training. In general, always have a separate validation and testing set (or use cross-validation) and make sure to regularize your model to prevent overfitting.

Generator for Training Samples

We need to generate positive samples and negative samples to train the neural network. The positive samples are simple: pick a pair from pairs and assign it a 1. The negative samples are also fairly easy: pick one random link and one random book, make sure they are not in pairs, and assign them a -1 or a 0. (We'll use either a -1 or 0 for the negative labels depending on whether we want to make this a regression or a classification problem. Either approach is valid, and we'll try out both methods.)

The code below creates a generator that yields batches of samples each time it is called. Neural networks are trained incrementally - a batch at a time - which means that a generator is a useful function for returning examples on which to train. Using a generator alleviates the need to store all of the training data in memory which might be an issue if we were working with a larger dataset such as images.

```
In [23]: import numpy as np
import random
random.seed(100)
def generate_batch(pairs, n_positive = 50, negative_ratio = 1.0, classification = False):
    """Generate batches of samples for training"""
    batch_size = n_positive * (1 + negative_ratio)
    batch = np.zeros((batch_size, 3))
    # Adjust label based on task
    if classification:
        neg_label = 0
    else:
        neg_label = -1
    # This creates a generator
```

```
    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)
        # Increment idx by 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]
```

To get a new batch, call next on the generator.

```
In [24]: next(generate_batch(pairs, n_positive = 2, negative_ratio = 2))
```

Out[24]:({'book': array([6895., 7206., 22162., 28410., 25757., 29814.]),
'link': array([260., 34924., 5588., 33217., 22920., 11452.])},
array([1., -1., 1., -1., -1., -1.]))

```
In [25]: x, y = next(generate_batch(pairs, n_positive = 2, negative_ratio = 2))
# Show a few example training pairs
for label, b_idx, l_idx in zip(y, x['book'], x['link']):
    print(f'Book: {index_book[b_idx]:30} Link: {index_link[l_idx]:40} Label: {label}')
```

Book: Deep Six (novel) Link: president of the united states Label: 1.0
Book: The Counterfeit Man Link: gerald gardner (wiccan) Label: -1.0
Book: Soul Music (novel) Link: peter crowther Label: -1.0
Book: The Soul of the Robot Link: category:house of night series Label: -1.0
Book: Des Imagistes Link: august strindberg Label: -1.0
Book: Bag of Bones Link: category:novels by stephen king Label: 1.0

The neural network will take in the book index and the link index and try to embed them in such a way that it can predict the label from the embeddings.

Neural Network Embedding Model

With our dataset and a supervised machine learning task, we're almost there. The next step is the most technically complicated but thankfully fairly simple with Keras. We are going to construct the neural network that learns the entity embeddings. The input to this network is the (book, link) (either positive or negative) as integers, and the output will be a prediction of whether or not the link was present in the book article. However, we're not actually interested in the prediction except as the device used to train the network by comparison to the label. What we are after is at the heart of the network: the embedding layers, one for the book and one for the link each of which maps the input entity to a 50-dimensional vector. The layers of our network are as follows:

1. Input: parallel inputs for the book and link
2. Embedding: parallel embeddings for the book and link
3. Dot: computes the dot product between the embeddings to merge them together
4. Reshape: utility layer needed to correct the shape of the dot product
5. Dense: fully connected layer with sigmoid activation to generate output for classification

After converting the inputs to an embedding, we need a way to combine the embeddings into a single number. For this, we can use the dot product which does element-wise multiplication of numbers in the vectors and then sums the result to a single number. This raw number (after reshaping) is then the output of the model for the case of regression. In regression, our labels are either -1 or 1, and so the model loss function will be a mean squared error in order to minimize the distance between the prediction and the output. Using the dot product with normalization means that the Dot layer is finding the cosine similarity between the embedding for the book and the link. Using this method for combining the embeddings means we are trying to make the network learn similar embeddings for books that link to similar pages.

Classification vs Regression

For classification, we add an extra fully connected Dense layer with a sigmoid activation to squash the outputs between 0 and 1 because the labels are either 0 or 1. The loss function for classification is binary_crossentropy which measures the error of the neural network predictions in a binary classification problem and is a measure of the similarity between two distributions. We can train with either classification or regression, and in practice, I found that both approaches produced similar embeddings. I'm not sure about the technical merits of these methods, and I'd be interested to hear if one is better than the other.

The optimizer - the algorithm used to update the parameters (also called weights) of the neural network after calculating the gradients through backpropagation - is Adam in both cases (Adam is a modification to Stochastic Gradient Descent). We use the default parameters for this optimizer. The nice thing about modern neural network frameworks is we don't have to worry about backpropagation or updating the model parameters because that is done for us. It's nice to have an idea of what is occurring behind the scenes, but it's not entirely necessary to use a neural network effectively.

In [26]:
```
from keras.layers import Input, Embedding, Dot, Reshape, Dense
from keras.models import Model
```

In [27]:
```
def book_embedding_model(embedding_size = 50, classification = False):
    """Model to embed books and wikilinks using the functional API.
       Trained to discern if a link is present in a article"""
    # 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),
                    output_dim = embedding_size)(book)
    # Embedding the link (shape will be (None, 1, 50))
    link_embedding = Embedding(name = 'link_embedding',
                    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)
    # If classifcation, add extra layer and loss function is binary cross entropy
    if classification:
        merged = Dense(1, activation = 'sigmoid')(merged)
        model = Model(inputs = [book, link], outputs = merged)
        model.compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
    # Otherwise loss function is mean squared error
    else:
        model = Model(inputs = [book, link], outputs = merged)
        model.compile(optimizer = 'Adam', loss = 'mse')
    return model
# Instantiate model and show parameters
model = book_embedding_model()
model.summary()
```

```
__________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
book (InputLayer)            (None, 1)                 0
__________________________________________________________________________________________________
link (InputLayer)            (None, 1)                 0
__________________________________________________________________________________________________
book_embedding (Embedding)   (None, 1, 50)             1851000     book[0][0]
__________________________________________________________________________________________________
link_embedding (Embedding)   (None, 1, 50)             2087900     link[0][0]
__________________________________________________________________________________________________
dot_product (Dot)            (None, 1, 1)              0           book_embedding[0][0]
                                                                   link_embedding[0][0]
__________________________________________________________________________________________________
reshape_1 (Reshape)          (None, 1)                 0           dot_product[0][0]
==================================================================================================
Total params: 3,938,900
Trainable params: 3,938,900
Non-trainable params: 0
__________________________________________________________________________________________________
```

There are nearly 4.0 million weights (parameters) that need to be learned by the neural network. Each of these represents one number in an embedding for one entity. During training, the neural network adjusts these parameters in order to minimize the loss function on the training data.

ANALYSIS

We have the training data - in a generator - and a model. The next step is to train the model to learn the entity embeddings. During this process, the model will update the embeddings (change the model parameters) to accomplish the task of predicting whether a certain link is on a book page or not. The resulting embeddings can then be used as a representation of books and links.

There are a few parameters to adjust for training. The batch size should generally be as large as possible given the memory constraints of your machine. The negative ratio can be adjusted based on results. We tried two and it seemed to work well. The number of steps per epoch is chosen such that the model sees a number of examples equal to the number of pairs on each epoch. This is repeated for 15 epochs (which might be more than necessary).

```
In [28]: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,verbose = 2)
```

```
----------------------------------------------------------------------------------------------------Epoch 1/15
 - 16s - loss: 0.9625
Epoch 2/15
 - 15s - loss: 0.7629
Epoch 3/15
 - 15s - loss: 0.5419
Epoch 4/15
 - 15s - loss: 0.5022
Epoch 5/15
 - 15s - loss: 0.4791
Epoch 6/15
 - 15s - loss: 0.4745
Epoch 7/15
 - 15s - loss: 0.4633
Epoch 8/15
 - 15s - loss: 0.4680
Epoch 9/15
 - 15s - loss: 0.4640
Epoch 10/15
 - 15s - loss: 0.4590
Epoch 11/15
 - 15s - loss: 0.4511
Epoch 12/15
 - 15s - loss: 0.4478
Epoch 13/15
 - 15s - loss: 0.4489
Epoch 14/15
 - 15s - loss: 0.4477
Epoch 15/15
 - 15s - loss: 0.4506
```

The loss decreases as training progress which should give us confidence the model is learning something!

Extract Embeddings and Analyze

The trained model has learned - hopefully - representations of books and wikilinks that place similar entities next to one another in the embedding space. To find out if this is the case, we extract the embeddings and use them to find similar books and links.

```
In [30]: # Extract embeddings
book_layer = model.get_layer('book_embedding')
book_weights = book_layer.get_weights()[0]
book_weights.shape
```

Out[30]: (37020, 50)

Each book is now represented as a 50-dimensional vector.

We need to normalize the embeddings so that the dot product between two embeddings becomes the cosine similarity.

```
In [31]: book_weights = book_weights / np.linalg.norm(book_weights, axis = 1).reshape((-1, 1))
book_weights[0][:10]
np.sum(np.square(book_weights[0]))
```

Out[31]: array([0.10427548, -0.23591727, 0.14965919, -0.07087466, 0.09660177, -0.20169103, 0.05245946, -0.08891259, -0.12968971, -0.03209771], dtype=float32)

Out[31]: 1.0

Normalize just means divide each vector by the square root of the sum of squared components.

DEMO

We've trained the model and extracted the embeddings - great - but where is the book recommendation system? Now that we have the embeddings, we can use them to recommend books that our model has learned are most similar to a given book.

Function to Find Most Similar Entities

The function below takes in either a book or a link, a set of embeddings, and returns the *n* most similar items to the query. It does this by computing the dot product between the query and embeddings. Because we normalized the embeddings, the dot product represents the cosine similarity between two vectors. This is a measure of similarity that does not depend on the magnitude of the vector in contrast to the Euclidean distance. (The Euclidean distance would be another valid metric of similarity to use to compare the embeddings.)

Once we have the dot products, we can sort the results to find the closest entities in the embedding space. With cosine similarity, higher numbers indicate entities that are closer together, with -1 the furthest apart and +1 closest together.

```
In [32]: import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('fivethirtyeight')
plt.rcParams['font.size'] = 15
def find_similar(name, weights, index_name = 'book', n = 10, least = False, return_dist = False, plot = False):
    """Find n most similar items (or least) to name based on embeddings. Option to also plot the results"""
    # Select index and reverse index
    if index_name ==
        'book': index =
        book_index rindex =
        index_book
    elif index_name == 'page':
        index = link_index
        rindex = index_link
    # Check to make sure `name` is in index
    try:
        # Calculate dot product between book and all others
        dists = np.dot(weights, weights[index[name]])
    except KeyError:
        print(f'{name} Not Found.')
        return
    # Sort distance indexes from smallest to largest
    sorted_dists = np.argsort(dists)
    # Plot results if specified
    if plot:
        # Find furthest and closest items
        furthest = sorted_dists[:(n // 2)]
        closest = sorted_dists[-n-1: len(dists) - 1]
        items = [rindex[c] for c in furthest]
        items.extend(rindex[c] for c in closest)
        # Find furthest and closets distances
        distances = [dists[c] for c in furthest]
        distances.extend(dists[c] for c in closest)
        colors = ['r' for _ in range(n //2)]
        colors.extend('g' for _ in range(n))
        data = pd.DataFrame({'distance': distances}, index = items)
        # Horizontal bar chart
        data['distance'].plot.barh(color = colors, figsize = (10, 8),
```

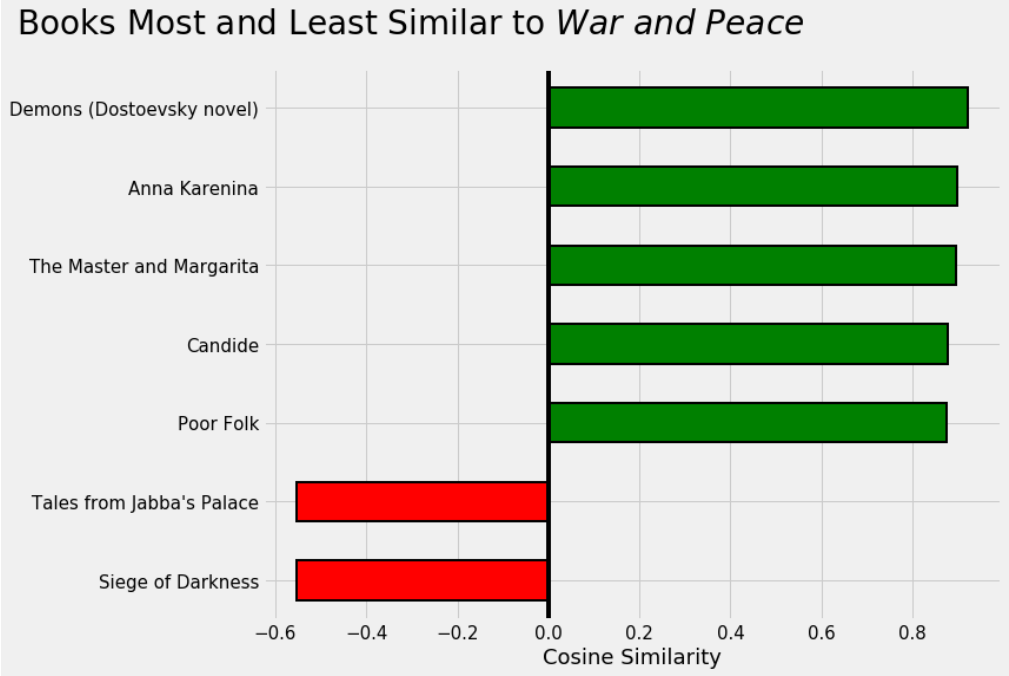
```
        edgecolor = 'k', linewidth = 2)
    plt.xlabel('Cosine Similarity');
    plt.axvline(x = 0,  color = 'k');
    # Formatting for italicized title
    name_str = f'{index_name.capitalize()}s Most and Least Similar to'
    for word in name.split():
        # Title uses latex for italize
        name_str += ' $\it{' + word + '}$'
    plt.title(name_str, x = 0.2, size = 28, y = 1.05)
    return None
# If specified, find the least similar
if least:
    # Take the first n from sorted distances
    closest = sorted_dists[:n]
    print(f'{index_name.capitalize()}s furthest from {name}.\n')
# Otherwise find the most similar
else:
    # Take the last n sorted distances
    closest = sorted_dists[-n:]
    # Need distances later on
    if return_dist:
        return dists, closest
    print(f'{index_name.capitalize()}s closest to {name}.\n')
# Need distances later on
if return_dist:
    return dists, closest
# Print formatting
max_width = max([len(rindex[c]) for c in closest])
# Print the most similar and distances
for c in reversed(closest):
    print(f'{index_name.capitalize()}: {rindex[c]:{max_width + 2}} Similarity: {dists[c]:.{2}}')
```

(We know that this function works if the most similar book is the book itself. Because we multiply the item vector times all the other embeddings, the most similar should be the item itself with a similarity of 1.0.)

```
In [33]: find_similar('War and Peace', book_weights)
------------------------------------------------------------------------------------------
 Books closest to War and Peace.

 Book: War and Peace                       Similarity: 1.0
 Book: Emperor of America                  Similarity: 0.48
 Book: The Centurions (Hunter novel)       Similarity: 0.46
 Book: Debt of Honor                       Similarity: 0.45
 Book: Life: A User's Manual               Similarity: 0.45
 Book: Keeper (Appelt novel)               Similarity: 0.44
 Book: En présence de Schopenhauer         Similarity: 0.44
 Book: Elite (video game)                  Similarity: 0.44
 Book: Piece of Cake (novel)               Similarity: 0.43
 Book: The Light That Failed               Similarity: 0.42
```

Finding the most furthest book-

```
In [34]:find_similar('War and Peace', book_weights, least = True, n = 5)
-------------------------------------------------------------------------------------
  Books furthest from War and Peace.

  Book: Ozma of Oz             Similarity: -0.4
  Book: The Titan (novel)      Similarity: -0.4
  Book: Death in the Clouds    Similarity: -0.41
  Book: The Summer Tree        Similarity: -0.42
  Book: Caravan to Vaccarès    Similarity: -0.46
-------------------------------------------------------------------------------------
```



We have successfully built a book recommendation system using neural network embeddings.

Visualizations

One of the most interesting parts about embeddings is that we can use them to visualize concepts such as *War and Peace* or *biography*. First, we have to take the embeddings from 50 dimensions down to either 3 or 2. We can do this using pca, tsne, or umap. We'll try both tsne and umap for comparison. TSNE takes much longer and is designed to retain local structure within the data. UMAP is generally quicker and is designed for a balance between the local and global structure in the embedding.

Manifold Embeddings

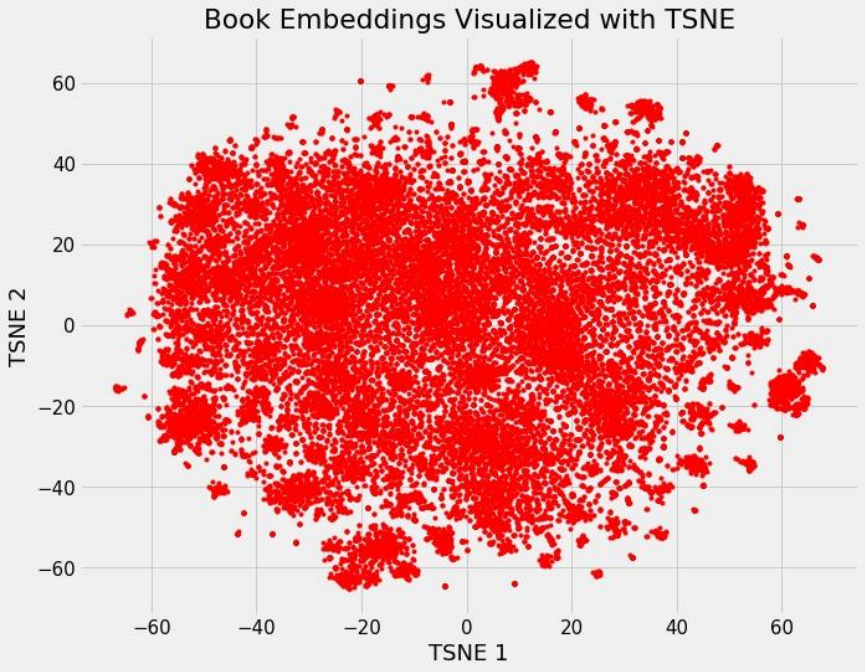
TSNE: t-Stochastic Distributed Neighbors Embedding, and UMAP: Uniform Manifold Approximation and Projection, are both methods that use the idea of a manifold to map vectors to a lower dimensional embedded space. Therefore, we are taking the 37,000 dimensions in the case of books, embedding them to 50 dimensions with the neural network, and then embedding them down to 2 dimensions with a manifold. The primary idea behind dimension reduction with a manifold is that there is a lower dimensional representation of the vectors that can still capture the variation between different groups. We want the embeddings to represent similar entities close to one another but in fewer dimensions that allow us to visualize the entities.

```
In [59]: from sklearn.manifold import TSNE
from umap import UMAP

In [60]: def reduce_dim(weights, components = 3, method = 'tsne'):
    """Reduce dimensions of embeddings"""
    if method == 'tsne':
        return TSNE(components, metric = 'cosine').fit_transform(weights)
    elif method == 'umap':
        # Might want to try different parameters for UMAP
        return UMAP(n_components=components, metric = 'cosine',
                    init = 'random', n_neighbors = 5).fit_transform(weights)

In [61]: book_r = reduce_dim(book_weights_class, components = 2, method = 'tsne')
Book_r.shape
----------------------------------------------------------------------------------------
```

Out[61]: (37020, 2)

We've now taken the initial 37,000 dimension book vector and reduced it to just 2 dimensions.

```
In [62]: InteractiveShell.ast_node_interactivity = 'last'
plt.figure(figsize = (10, 8))
plt.plot(book_r[:, 0], book_r[:, 1], 'r.')
plt.xlabel('TSNE 1'); plt.ylabel('TSNE 2'); plt.title('Book Embeddings Visualized with TSNE');
```



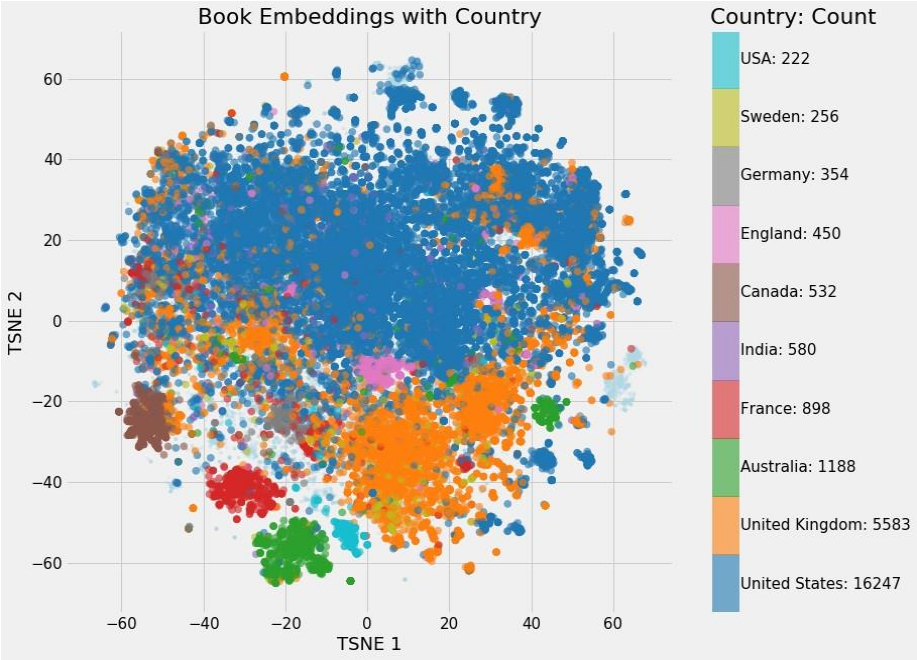
There do appear to be a few noticeable clumps. However, it's difficult to derive any meaning from this plot since we aren't distinguishing books in any way.

Let's make a function that can color the plot by any attribute in the book infobox.

In [74]:
```
def plot_by_attribute(attribute):

    """Color book embedding by `attribute`"""
    # Find all the attribute values
    attrs = [book[1].get(attribute, 0) for book in books]
    # Remove attributes not found
    attr_counts = count_items(attrs)
    del attr_counts[0]
    # Include 10 most popular attributes
    attr_to_include, counts = list(attr_counts.keys())[:10], list(attr_counts.values())[:10]
    idx_include = []
    attributes = []
    # Iterate through books searching for the attribute
    for i, book in enumerate(books):
        # Limit to books with the attribute
        if attribute in book[1].keys():
            # Limit to attribute in the 10 most popular
            if book[1][attribute] in attr_to_include:
                idx_include.append(i)
                attributes.append(book[1][attribute])
    # Map to integers
    ints, attrs = pd.factorize(attributes)
    plt.figure(figsize = (12, 10))
    plt.scatter(book_r[:, 0], book_r[:, 1], marker = '.', color = 'lightblue', alpha = 0.2)
    # Plot embedding with only specific attribute highlighted
    plt.scatter(book_r[idx_include, 0], book_r[idx_include, 1], alpha = 0.6,
                c = ints, cmap = plt.cm.tab10, marker = 'o', s = 50)
    # Add colorbar and appropriate labels
    cbar = plt.colorbar()
    cbar.set_ticks([])
```

```
    tick_labels = [f'{attr}: {count}' for attr, count in zip(attr_to_include, counts)]
    # Labeling
    for j, lab in enumerate(tick_labels):
        cbar.ax.text(1, (2 * j + 1) / ((10) * 2), lab, ha='left', va='center')
    cbar.ax.set_title(f'{attribute.capitalize()}: Count', loc = 'left')
    plt.xlabel('TSNE 1'); plt.ylabel('TSNE 2'); plt.title(f'Book Embeddings with {attribute.capitalize()}');
```

In [77]: plot_by_attribute('country')



Conclusions

We built an effective book recommendation system using the principle that books with similar outgoing links are similar and all of the book articles on Wikipedia.

We embedded both the wikilinks and the books using a neural network. To train the neural network, we developed a supervised machine learning problem of classifying if a given link was present on the page for a book.

More than just training the neural network, we saw how to thoroughly inspect the embeddings in order to find the closest books to a given book in embedding space. We also saw how to visualize the embeddings which sometimes can show us interesting clusterings

In the process, we took the original 37,000 dimensions of the books and reduced it first to 50 using the neural network and then to 2 using a manifold learning method. The neural network embedding was useful for making recommendations based on closest entities while the TSNE embedding is useful primarily for visualization.

Retraining the model

Initially we had set the hyperparameters to :

- Embedding Vector size = 100
- Epoch = 4
- Positive and negative ratio = 150 & 10
- Number of trainable parameters = 7,877,800

The screenshots below give the hyperparameter values:

Layer (type)	Output Shape	Param #	Connected to
book (InputLayer)	(None, 1)	0	
link (InputLayer)	(None, 1)	0	
book_embedding (Embedding)	(None, 1, 100)	3702000	book[0][0]
link_embedding (Embedding)	(None, 1, 100)	4175800	link[0][0]
dot_product (Dot)	(None, 1, 1)	0	book_embedding[0][0] link_embedding[0][0]
reshape_1 (Reshape)	(None, 1)	0	dot_product[0][0]

Total params: 7,877,800
Trainable params: 7,877,800
Non-trainable params: 0

```
Epoch 1/4
 - 109s - loss: 0.9937
Epoch 2/4
 - 102s - loss: 0.9658
Epoch 3/4
 - 99s - loss: 0.9047
Epoch 4/4
 - 100s - loss: 0.8631
```

We further went to modify our parameters to achieve a better recommendation

- Embedding Vector size = 50
- Epoch = 15
- Positive and negative ratio = 50 and 1

Model Improvement

As seen from above(Retrain) when we trained with the initial hyperparameters the similarity obtained was:

```
Books closest to Artificial Intelligence: A Modern Approach.

Book: Artificial Intelligence: A Modern Approach          Similarity: 1.0
Book: Journeys of Frodo                                   Similarity: 0.55
Book: The Fourth Protocol                                 Similarity: 0.53
Book: Blood Noir                                          Similarity: 0.53
Book: Gun, with Occasional Music                          Similarity: 0.53
Book: Professor Risley and the Imperial Japanese Troupe   Similarity: 0.52
Book: Les Illusions de la Psychanalyse                    Similarity: 0.52
Book: Goldsborough (novel)                                Similarity: 0.52
Book: Mattimeo                                            Similarity: 0.51
Book: The Burning Mountain                                Similarity: 0.51
```

On improving the hyperparameters the following similarity was obtained:

```
Books closest to Artificial Intelligence: A Modern Approach.

Book: Artificial Intelligence: A Modern Approach   Similarity: 1.0
Book: Designing Virtual Worlds                     Similarity: 0.71
Book: The Wrecker (Stevenson novel)                Similarity: 0.7
Book: Who Will Remember the People...              Similarity: 0.7
Book: Partisans (novel)                            Similarity: 0.69
Book: Charlotte Gray (novel)                       Similarity: 0.68
Book: Pilgrims (short story collection)            Similarity: 0.67
Book: We Were Eight Years in Power                 Similarity: 0.67
Book: Defending Jacob                              Similarity: 0.67
Book: Collector of Names                           Similarity: 0.67
```

Hence, as seen as from the screenshots above there is a drastic improvement in the recommendation made.

Extending the model

The screenshot below shows the recommendation system on our current project where we used book and wikilinks to get the similarity.

```
find_similar('Artificial Intelligence: A Modern Approach', book_weights, n = 5)
```

```
Books closest to Artificial Intelligence: A Modern Approach.

Book: Artificial Intelligence: A Modern Approach          Similarity: 1.0
Book: Computer Graphics: Principles and Practice          Similarity: 0.96
Book: Structure and Interpretation of Computer Programs   Similarity: 0.95
Book: Perl Cookbook                                       Similarity: 0.95
Book: Algorithms + Data Structures = Programs             Similarity: 0.95
```

In this, we used a new data set that consists of movies and their ratings. A new function was created in which the movie name is given as input and the similarity is found. The following screenshot shows the movie similarity based on its ratings.

```
get_recommendations('The Dark Knight').head(10)
```

```
8031         The Dark Knight Rises
6218                Batman Begins
6623                  The Prestige
2085                     Following
7648                     Inception
4145                      Insomnia
3381                        Memento
8613                   Interstellar
7659    Batman: Under the Red Hood
1134                Batman Returns
Name: title, dtype: object
```

Finally, we implemented a similar recommender system using a new dataset.

Innovation

We built our recommendation system for books and this model was extended for movies recommendation.

Summary

1. Our goal was to build a recommender system using a feed-forward neural network, with our input data in the form of JSON files (book titles and wikilinks for each book) which we downloaded and imported into the jupyter notebook. Our output data mainly focussed on providing recommendation (similar) of books that were inputted.

2. Our neural network consisted of several layers Input layers, Embedding layers and a layer with the help of a sigmoid activation function.

3. We used several hyperparameters which are: Batch size, Epoch and we used the binary cross entropy loss function to determine the losses and also an Adam optimizer to optimize the weights through backpropagation.

4. The number of trainable parameters used in our model was close to about 4 million weights and it had a time complexity of O(n).

5. The demo requirements have been displayed in the model above.

6. We re-trained our model several times by altering the hyperparameters to obtain the best-optimized hyperparameter for the model to give the best recommendation.

7. We extended are recommendation system to a movie data set which consisted of movie ratings and their titles.

8. After finding the best hyperparameters we found the best recommendations made for the books which enhanced our overall performance.

9. We tried our best to come up with our own recommendation system which can be used for recommending books.

References

1. As on April 28th: https://arxiv.org/pdf/1707.07435.pdf
2. As on April 29th: https://github.com/xiaoouzhang/Collaborative-Deep-Learning-for-Recommender-Systems
3. As on April 28th: https://towardsdatascience.com/recommender-systems-with-deep-learning-architectures-1adf4eb0f7a6
4. As on April 28th: https://www.sciencedirect.com/science/article/pii/S2405959518302029
5. As on April 29th: https://pdfs.semanticscholar.org/8bdb/cce242fd63013dc82ad6341a58474119a225.pdf