```
from fastai.collab import *
from fastai.tabular.all import *
```

Collaborative Filtering from scratch - Movie Reviews

This notebook, based on the work here, is a deep dive into how collaborative filtering works. I will be using collaborative filtering to recommend movies based on the preferences of other users and I will build the model from scratch, to help me gain a deeper understanding.

We will be approaching collaborative filtering with 2 techniques:

1. **Probabilistic matrix factorization** (PMF)
2. **Deep learning**

I will start by using PMF to understand the basis of collaborative filtering models. We will analyse our embeddings to see all the useful information that can be gained from collaborative filtering.

I will conclude the notebook by approaching collaborative filtering with deep learning and discussing boostrapping and feedback loops.

So, what is collaborative filtering? Collaborative filtering is a method whereby a user's preferences are predicted based on the preferences of other users. It is underpinned by the assumption that people who agreed in the past will agree in the future.

A key foundational idea is **latent factors**. For example, Netflix don't need to ask you and keep a database of exactly which genres, film lengths, and time periods you like. Latent factors will be the parameters, unspecified in what they represent, used to compare the preferences of various users. We use the parameters to understand the likelihood that a person will like a product.

However, collaborative filtering isn't entirely for Netflix! There is a general class of problems collaborative filtering can solve and we generally refer to **items** which could be movies, links, diagnoses and many more...

Data Extraction

Let's start by gathering some data representing movie ratings from MovieLens dataset. Since this model is being used to inform my learning, rather than produce useful results, we will use a subset of the data for speed.

```
path = untar_data(URLs.ML_100k)
```

```
100.15% [4931584/4924029 00:00<00:00]
```

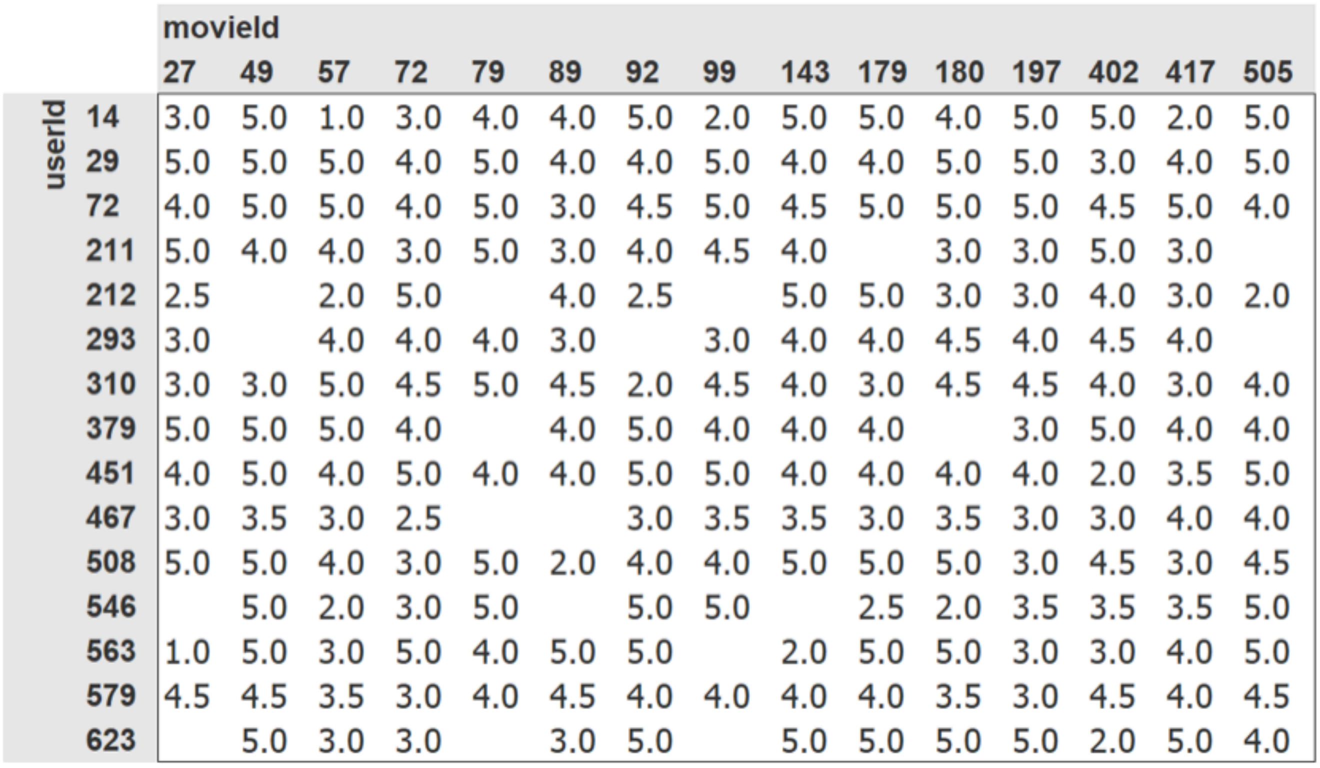
```
df = pd.read_csv(path/'u.data', # README specifies location of data
                 delimiter='\t', # README specifies tab separated file or TSV
                 header=None, # README specifies no labelled columns
                 names=['user', 'movie', 'rating', 'timestamp']) # we can add ou
```

```
df.head()
```

	user	movie	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

It is useful to understand what this dataset looks like cross-tabulated. Essentially, we want to fill the missing spots with predictions. This is called **matrix completion** which is the core of PMF collaborative filtering. These predictions will represent the predicted rating the user will give the movie, based on other users preferences. This could assist in making recommendations and assisting users to discover new content.

It's worth noting that the values we try to predict don't have to be ratings, it could even just be 1s and 0s, representing whether a user purchased a product.



	movieId														
userId	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
14	3.0	5.0	1.0	3.0	4.0	4.0	5.0	2.0	5.0	5.0	4.0	5.0	5.0	2.0	5.0
29	5.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	4.0	4.0	5.0	5.0	3.0	4.0	5.0
72	4.0	5.0	5.0	4.0	5.0	3.0	4.5	5.0	4.5	5.0	5.0	5.0	4.5	5.0	4.0
211	5.0	4.0	4.0	3.0	5.0	3.0	4.0	4.5	4.0		3.0	3.0	5.0	3.0	
212	2.5		2.0	5.0		4.0	2.5		5.0	5.0	3.0	3.0	4.0	3.0	2.0
293	3.0		4.0	4.0	4.0	3.0		3.0	4.0	4.0	4.5	4.0	4.5	4.0	
310	3.0	3.0	5.0	4.5	5.0	4.5	2.0	4.5	4.0	3.0	4.5	4.5	4.0	3.0	4.0
379	5.0	5.0	5.0	4.0		4.0	5.0	4.0	4.0	4.0		3.0	5.0	4.0	4.0
451	4.0	5.0	4.0	5.0	4.0	4.0	5.0	5.0	4.0	4.0	4.0	4.0	2.0	3.5	5.0
467	3.0	3.5	3.0	2.5			3.0	3.5	3.5	3.0	3.5	3.0	3.0	4.0	4.0
508	5.0	5.0	4.0	3.0	5.0	2.0	4.0	4.0	5.0	5.0	5.0	3.0	4.5	3.0	4.5
546		5.0	2.0	3.0	5.0		5.0	5.0		2.5	2.0	3.5	3.5	3.5	5.0
563	1.0	5.0	3.0	5.0	4.0	5.0	5.0		2.0	5.0	5.0	3.0	3.0	4.0	5.0
579	4.5	4.5	3.5	3.0	4.0	4.5	4.0	4.0	4.0	4.0	3.5	3.0	4.5	4.0	4.5
623		5.0	3.0	3.0		3.0	5.0		5.0	5.0	5.0	5.0	2.0	5.0	4.0

Latent Factors

The next step is to define our latent factors, which will be a set of parameters for each movie and user.

We will take the **dot product** of the movie parameters and the user parameters. A useful way to understand these parameters is to consider the movie latent factors to 'describe' the movie, while the user latent factors indicate the users 'preferences' towards the 5 parameters describing the movie.

A lower score indicates a lower preference or a lower quantity of this certain factor in a movie. A higher score indicates the converse.

When we take the dot product of the two sets of latent factors, we get an indication of how suitable the movie is for their tastes, in other words - a predicted rating.

How do we do this in PyTorch? The user latent factors and the movie latent factors will be stored in an **embedding matrix** (AKA factor matrix) and the latent factors are described as "movie embeddings" and "user embeddings". PyTorch stores the embeddings in a matrix, and every time it needs to calculate a dot product, it will "search up" the parameters in the embedding matrix.

Now that we have a way of making a prediction, we also have a way of evaluating loss; for example, we could take the RMSE of the predictions. With loss and a way of making predictions, we have all we need for a machine learning model - we simply need to use stochastic gradient descent to update our parameters!

So, let's **learn** our latent factors using FastAI, which wraps PyTorch and contains a convinient `CollabDataLoaders` object.

We can also combine the names of the movies to make it a bit more fun to interpret!

```
movies = pd.read_csv(path/'u.item', delimiter='|', encoding='latin-1', usecols=
```

```
df = df.merge(movies)
```

```
dls = CollabDataLoaders.from_df(df,
                                item_name='title', # we pass the name of the mo
                                bs=64)
dls.show_batch()
```

	user	title	rating
0	650	Mrs. Doubtfire (1993)	3
1	332	Relic, The (1997)	4
2	270	Welcome to the Dollhouse (1995)	5
3	747	Some Like It Hot (1959)	5
4	918	Paris, Texas (1984)	4
5	22	Grease (1978)	4
6	397	Pulp Fiction (1994)	5
7	223	Desperate Measures (1998)	2
8	733	Unhook the Stars (1996)	3
9	642	Snow White and the Seven Dwarfs (1937)	2

Let's choose an arbitrary number of latent factors for both users and movies and create embedding matrices.

```
n_users = len(dls.classes['user'])
n_movies = len(dls.classes['title'])
n_factors = 5

# we won't actually use this, just for illustrative purposes
user_factors = torch.randn(n_users, n_factors) # create an embedding matrix, ir
movie_factors = torch.randn(n_movies, n_factors) # create an embedding matrix,
```

At this point, you might be wondering how we "*look up in the embedding matrix*", as deep learning models can only do matrix products and activation functions. To get a prediction for a particular user-movie pair we need to find the user's vector from the user embedding matrix and find the movie's vector from the vector embedding matrix.

We can't do indexing in deep learning architectures. A one-hot-encoded (OHE) approach would work, but would be computationally inefficient. So, most deep learning architectures "*search*" by introducing an **embedding layer**.

The embedding layer:

- maps an integer index (i.e., user ID, movie ID) to a vector.
- works like a matric look up, but has gradients (for backpropagation).

To use **embedding** we can index the array like we normally would. But it is critical to understand what is actually happening here is not the same as indexing.

Note to self: "Stochastic" means random, referring to the fact that the algorithm uses a randomly selected subset (called a mini-batch) to update the model's parameters, rather than the entire dataset.

Creating the Model

When we write a model in PyTorch, we use object-oriented programming. We need to use **inheritance** by defining `Module` as our superclass, whenever designing a model in PyTorch. By inserting a function called `forward()` we can use `DotProduct` like a function, as PyTorch will automatically call `forward()` when the model is used.

This is important to understand when creating models with PyTorch. Firstly, every model inherits from `nn.Module`, the superclass. We put the calculation of our model in a function called `forward()` which PyTorch will automatically call when the model is used, meaning we can use our **object** like it is a **function**. PyTorch will handle backpropagation for us.

Note to self: "Backpropagation" is the algorithm used to compute gradients of the loss function with respect to the model's parameters. It uses the **chain rule** of calculus to compute how much each parameter contributed to the error (or loss).

- `__init__()`: We will use a similar kind of approach as above to define our **embedding matrices** using the built-in `Embedding` class.
- `forward()`: `x` will represent the user and movie for a batch. Each row will be one user and one movie.
    - We will extract `users`, by extracting every row in the 0th column, then searching the users embedding matrix. `users` will represent the factors for each user in the batch.
    - We will extract `movies`, by extracting every row in the 1st column, then searching the movies embedding matrix. `movies` will represent the factors for each movie in the batch.

Then, we can do our dot product, using `dim=1` because we are summing across the columns for each row. The sum across each row will be the prediction for a certain movie and user.

We use the sigmoid function to squish the predictions between 0 and 5, since a rating cannot be more than 5. We define the upper bound as 5.5, as a sigmoid function is asymptotic, but some people do give 5 star reviews.

```
class DotProduct(Module):
  def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
    self.user_factors = Embedding(n_users, n_factors)
    self.movie_factors = Embedding(n_movies, n_factors)
    self.y_range = y_range

  def forward(self, x):
    users = self.user_factors(x[:,0])
    movies = self.movie_factors(x[:,1])
    return sigmoid_range((users * movies).sum(dim=1), *self.y_range)
```

Now that we have our architecture and parameters defined, we create a `Learner` to train (or optimise) our model. While we could use a special function set everything up for us, since we are doing it from scratch we use the plain old `Learner` class.

We can define our hyperparameters and fit the model.

```
model = DotProduct(n_users, n_movies, 50)
learn = Learner(
    dls,
    model,
    loss_func=MSELossFlat()
)
```

```
learn.fit_one_cycle(5, 5e-3)
```

```
epoch  train_loss  valid_loss  time
    0    0.932399    0.994371  00:14
    1    0.645935    0.948996  00:10
    2    0.456119    0.953378  00:09
    3    0.356813    0.953273  00:08
    4    0.357834    0.951467  00:06
```

We can even run this on the CPU, as it is a very efficient model! But, we can make the model better, as it does not currently contain a bias term.

We can imagine the effect of the bias by considering a row of movies that a particular user has liked. What if they're a user that has the same preferences as another user, but they just rate their movies higher?

This can be accounted for using a **bias term**, which we will adjust in our model architecture.

We use `keepdim` to ensure that the output retains its original dimensionality instead of collapsing.

```
class DotProductBias(Module):
  def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
    self.user_factors = Embedding(n_users, n_factors)
    self.user_bias = Embedding(n_users, 1)
    self.movie_factors = Embedding(n_movies, n_factors)
    self.movie_bias = Embedding(n_movies, 1)
    self.y_range = y_range

  def forward(self, x):
    users = self.user_factors(x[:,0])
    movies = self.movie_factors(x[:,1])
    res = (users * movies).sum(dim=1, keepdim=True)
    res += self.user_bias(x[:,0]) + self.movie_bias(x[:,1])
    return sigmoid_range(res, *self.y_range)
```

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3)
```

epoch	train_loss	valid_loss	time
0	0.850025	0.943141	00:08
1	0.569409	0.913761	00:07
2	0.397444	0.939662	00:07
3	0.318175	0.950044	00:08
4	0.291180	0.950547	00:07

Weight Decay

Now, we can see that the model is overfitting, making this a good time to introduce **weight decay** also known as **L2 regularization**.

Weight decay consists of adding to your loss function the sum of all the weight squared. By adding a small contriution to the gradients we can encourage the weights to be as small as possible.

Consider the difference between the parabola `50x**2` and `0.5x**2`.

If we consider `a` in `ax**2` to be a single parameter (in reality our model has many more), we can visualise how letting the model learn very large parameters will lead to an overcomplex function with very sharp/steep changes.

Thus, weight decay will prevent the weights from growing to much, hence hindering the training of the model. But, it will also yield a state where the model trains better. So, it is a fine balance. A larger weight decay decreases the complexity of the model, increasing generalisation, and the converse is true for a smaller weight decay.

The weight decay will set unnecessary parameters (when we have too many parameters for the model) to 0 and it will control the growth of the other parameters.

While we could use weight decay like this:

```
loss_with_wd = loss + wd * (parameters**2).sum()
```

This would be very inefficient.

The whole purpose of loss is to take its gradient. So, instead, we add the `wd` to the parameters instead by calculating the derivative of `wd`.

```
parameters.grad += wd * 2 * parameters
```

In reality, since we set `wd` ourselves, the `2` is just absorbed into it.

When we are using FastAI, especially with computer vision, it will generally do a good job of calculating a good weight decay. But when using tabular data or collaborative filtering, it is hard for the defaults to work well, as there is not enough information about the data. So, it is necessary to manually set the weight decay, starting at 0.1 and dividing by 10 a few times, choosing the model that produces the best results.

Weight decay is for **regularization**, which is making your model no more complex then it has to be. It reduces the capacity of your model to prevent overfitting and improve generalisation.

Now that we understand **what weight decay actually is**, we can use it in our function to help the model generalise better.

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.945960	0.948768	00:08
1	0.691562	0.894969	00:09
2	0.529700	0.876837	00:07
3	0.447543	0.866099	00:08
4	0.430891	0.860853	00:08

Now the loss on the validation set is consistently going down and the model is no longer overfitting!

Note to self: "Noise" is random or irrelevant variations in the data that can distort patterns, especially when the model fits to the variation instead of generalisation (also known as **overfitting**).

Creating our Own Embedding Module

Let's go a layer deeper with our understanding by manually creating an **embedding module**.

When we are defining parameters in our `__init__` function we need to use `nn.Parameter` to tell PyTorch that this variable is storing a **parameter** or **weight**. When we use `Embedding` and other pre-defined modules, this is done for us.

We will define a function called `create_params()` that creates a matrix of random values (specified by the `size` argument).

We use `.normal_()` to modify `torch.zeros(*size)` in-place.

> *In PyTorch, functions ending with an underscore change the object in place.*

```
def create_params(size):
  return nn.Parameter(torch.zeros(*size).normal_(0, 0.01))
```

Now, we can create our model as above. Except, instead of using `Embedding` we will use the function we defined above, `create_params()` . Otherwise, the rest of the model is identical.

```
class DotProductBias(Module):
  def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
    self.user_factors = create_params([n_users, n_factors])
    self.user_bias = create_params([n_users])
    self.movie_factors = create_params([n_movies, n_factors])
    self.movie_bias = create_params([n_movies])
    self.y_range = y_range

  def forward(self, x):
    users = self.user_factors[x[:,0]]
    movies = self.movie_factors[x[:,1]]
    res = (users * movies).sum(dim=1)
    res += self.user_bias[x[:,0]] + self.movie_bias[x[:,1]]
    return sigmoid_range(res, *self.y_range)
```

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.849849	0.945003	00:08
1	0.686200	0.903113	00:08
2	0.534848	0.871842	00:08
3	0.458874	0.858972	00:08
4	0.430648	0.853904	00:09

Collaborative Filtering Analysis

Now that we've defined our parameters, what else can we do with them? Obviously, our model can make useful recommendations for our users, but we can also interpret the embeddings and biases manually.

If we look at the `movie_bias` parameters, this will give us an indication of which movies were generally well liked and which movies weren't.

```
movie_bias = learn.model.movie_bias.squeeze()
idxs = movie_bias.argsort()[:5]
[dls.classes['title'][i] for i in idxs]
```

```
['Mortal Kombat: Annihilation (1997)',
 'Children of the Corn: The Gathering (1996)',
 'Free Willy 3: The Rescue (1997)',
 'Vampire in Brooklyn (1995)',
 'Grease 2 (1982)']
```

More specifically, this list is telling us which movies were generally **unliked, even by people who would ordinarily like this type of movie.**

```
idxs = movie_bias.argsort(descending=True)[:5]
[dls.classes['title'][i] for i in idxs]
```

```
['Good Will Hunting (1997)',
 'Shawshank Redemption, The (1994)',
 'Close Shave, A (1995)',
 'To Kill a Mockingbird (1962)',
 "Schindler's List (1993)"]
```

On the other hand, this list is tellins us which movies were generally **liked, even by people who ordinarily not like this type of movie.**

The bias term shifts the score of a movie for everyone either up or down depending on how "*good*" the movie is.

We could also do this with the bias terms for the users, and this would give us an indication if the user **generally likes movies more than others or not**.

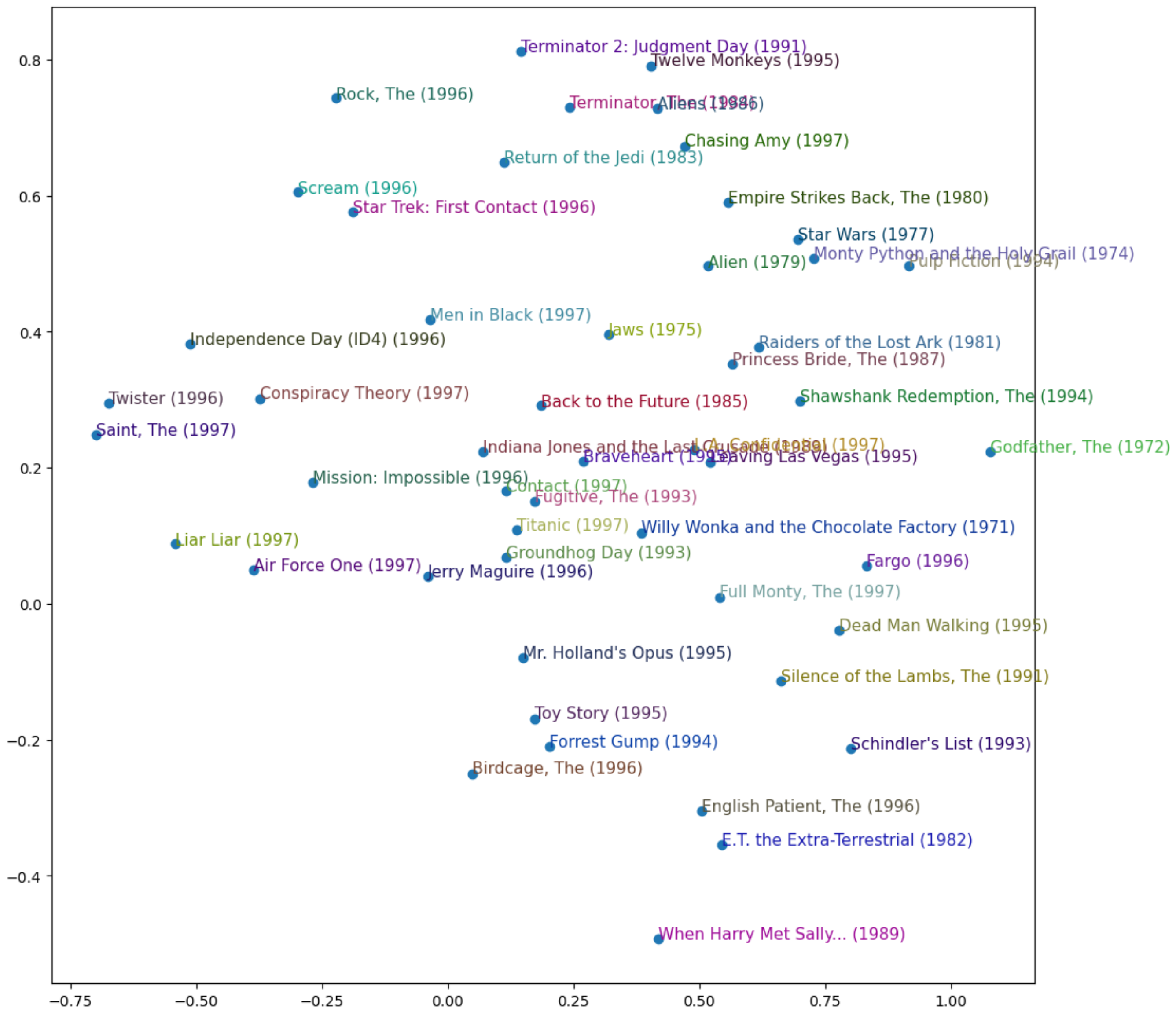
We can also analyse our embedding matrices, but this is not so easy to do, as they are a complex interplay of many, many factors.

However, we can use **principle component analysis** (PCA) to analyse the most important underlying **directions**. Essentially, we can produce a graph that **groups together like movies**.

The detail behind principle component analysis is something we will touch on at a later date, but the graph below shows the interesting and useful information contained in our embedding matrices.

> We don't have to tell our model *anything* about movies and it is able to find this information with a few thousand ratings and simple mathematical functions!

```
g = df.groupby('title')['rating'].count()
top_movies = g.sort_values(ascending=False).index.values[:1000]
top_idxs = tensor([learn.dls.classes['title'].o2i[m] for m in top_movies])
movie_w = learn.model.movie_factors[top_idxs].cpu().detach()
movie_pca = movie_w.pca(3)
fac0,fac1,fac2 = movie_pca.t()
idxs = list(range(50))
X = fac0[idxs]
Y = fac2[idxs]
plt.figure(figsize=(12,12))
plt.scatter(X, Y)
for i, x, y in zip(top_movies[idxs], X, Y):
    plt.text(x,y,i, color=np.random.rand(3)*0.7, fontsize=11)
plt.show()
```



Using Framework

Now that we've built a collaborative filtering model from scratch, it's time to make use of FastAI's framework, which is similar to what we ordinarily use as a machine learning practitioner.

```
learn = collab_learner(dls, n_factors=50, y_range=(0,5.5))
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.861928	0.945396	00:08
1	0.663045	0.890177	00:07
2	0.534758	0.875460	00:08
3	0.443639	0.861173	00:07
4	0.437523	0.854919	00:07

We can easily look at the underlying structure of our model...

```
learn.model
```

```
EmbeddingDotBias(
  (u_weight): Embedding(944, 50)
  (i_weight): Embedding(1665, 50)
  (u_bias): Embedding(944, 1)
  (i_bias): Embedding(1665, 1)
)
```

And we can access these layers using dot notation to generate the same information about the bias terms as before!

```
movie_bias = learn.model.i_bias.weight.squeeze()
idxs = movie_bias.argsort(descending=True)[:5]
[dls.classes['title'][i] for i in idxs]
```

```
['Good Will Hunting (1997)',
 'Titanic (1997)',
 'Apt Pupil (1998)',
 'To Kill a Mockingbird (1962)',
 'Star Wars (1977)']
```

Embedding Distance

Embedding distance is another useful technique if we want to find out how similar two things are - like 2 movies or 2 users.

We can use compare the embedding vectors between 2 movies or 2 users to determine how similar they are. We will use **cosine similarity**, which measures how close 2 vectors are in terms of their direction (rather than their magnitude).

1. If two vectors point in the same direction, cosine similarity will be close to 1.
2. If they are perpendicular, cosine similarity is 0.
3. If they point in opposite directions, cosine similarity is -1.

This is the formula for cosine similarity, purely for reference - the code will do this for us:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

- .i_weight.weight` retrieves the embedding vectors for all movies from the model.
- `dls.classes['title']` contains a mapping of movie titles to indicies. `.o2i()` means *object-to-index* and it converts a movie title to its corresponding index in the dataset.
- `nn.CosineSimilarity()` computes the cosine similarity between The Silence of the Lambs and **every other movie**. We use `[None]` to add a batch dimension, so it can be compared with all other embeddings at once.
- `argsort()` sorts the movies based on similarity, most similar to least similar.
- since `dls.classes['title']` contains a mapping of movie titles to indicies, we can look up `idx` s and return the top 5, which are the 5 movies most similar to The Silence of the Lambs.

```
movie_factors = learn.model.i_weight.weight
idx = dls.classes['title'].o2i['Silence of the Lambs, The (1991)']
distances = nn.CosineSimilarity(dim=1)(movie_factors, movie_factors[idx][None])
idx = distances.argsort(descending=True)
dls.classes['title'][idx][:5]
```

```
(#5) ['Silence of the Lambs, The (1991)','Innocents, The (1961)','Little
Princess, A (1995)','Apt Pupil (1998)','Jean de Florette (1986)']
```

Deep Learning

When using PMF, to generate the prediction for a single movie-user combo, we take the dot product of the two corresponding embedding vectors. With deep learning, we instead concatenate the two embedding vectors and pass it through the NN. This means the embedding vector for movies and the embedding vector for users can be different lengths.

FastAI has a function `get_emb_sz()` that returns recommended sizes for embedding matrices.

```
embs = get_emb_sz(dls)
embs
```

```
[(944, 74), (1665, 102)]
```

Now we can implement our model in PyTorch, using the `forward()` function like we previously learnt.

```
class CollabNN(Module):
  def __init__(self, user_sz, item_sz, y_range=(0,5.5), n_act=100):
    self.user_factors = Embedding(*user_sz)
    self.item_factors = Embedding(*item_sz)
    self.layers = nn.Sequential(
        nn.Linear(user_sz[1] + item_sz[1], n_act),
        nn.ReLU(),
        nn.Linear(n_act, 1)
    )
    self.y_range = y_range

  def forward(self, x):
    embs = self.user_factors(x[:,0]), self.item_factors(x[:,1])
    x = self.layers(torch.cat(embs, dim=1))
    return sigmoid_range(x, *self.y_range)
```

A lot of this code should look pretty familiar. We define our 2 embedding matrices based on the sizes passed into our class. We define our `layers` using `nn.Sequential()` which are exactly the same as our MNIST manual NN but with different sizes.

The `forward()` method is where the magic happens. `x` is our input tensor, where each row is one training example. So `x` is a 2D tensor of shape **(batch_size, 2)**. The first column contains user indices and the second column contains item indices.

We can extract the relevant embedding vectors using `user_factors` and `item_factors` respectively. Then, we pass it through our `self.layers` after concatenating the vectors with `torch.cat()` along the 1st dimension.

As before, we scale the output with `sigmoid_range()`.

```
model = CollabNN(*embs)
```

```
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3, wd=0.01)
```

epoch	train_loss	valid_loss	time
0	0.933298	0.953081	00:09
1	0.847710	0.935981	00:07
2	0.826919	0.887819	00:10
3	0.773110	0.876167	00:10
4	0.746893	0.869014	00:10

Now, we've created a deep learning collaborative model in PyTorch. But, if we want to make our lives a lot easier, we can use the `collab_learner()` from FasrAI with the argument `use_nn=True` to define 2 hidden layers of size 100 and 50 respectively.

```
learn = collab_learner(dls, use_nn=True, y_range=(0, 5.5), layers=[100,50])
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.952636	0.984127	00:11
1	0.870174	0.910506	00:09
2	0.841028	0.890496	00:09
3	0.774100	0.872670	00:09
4	0.752286	0.870227	00:09

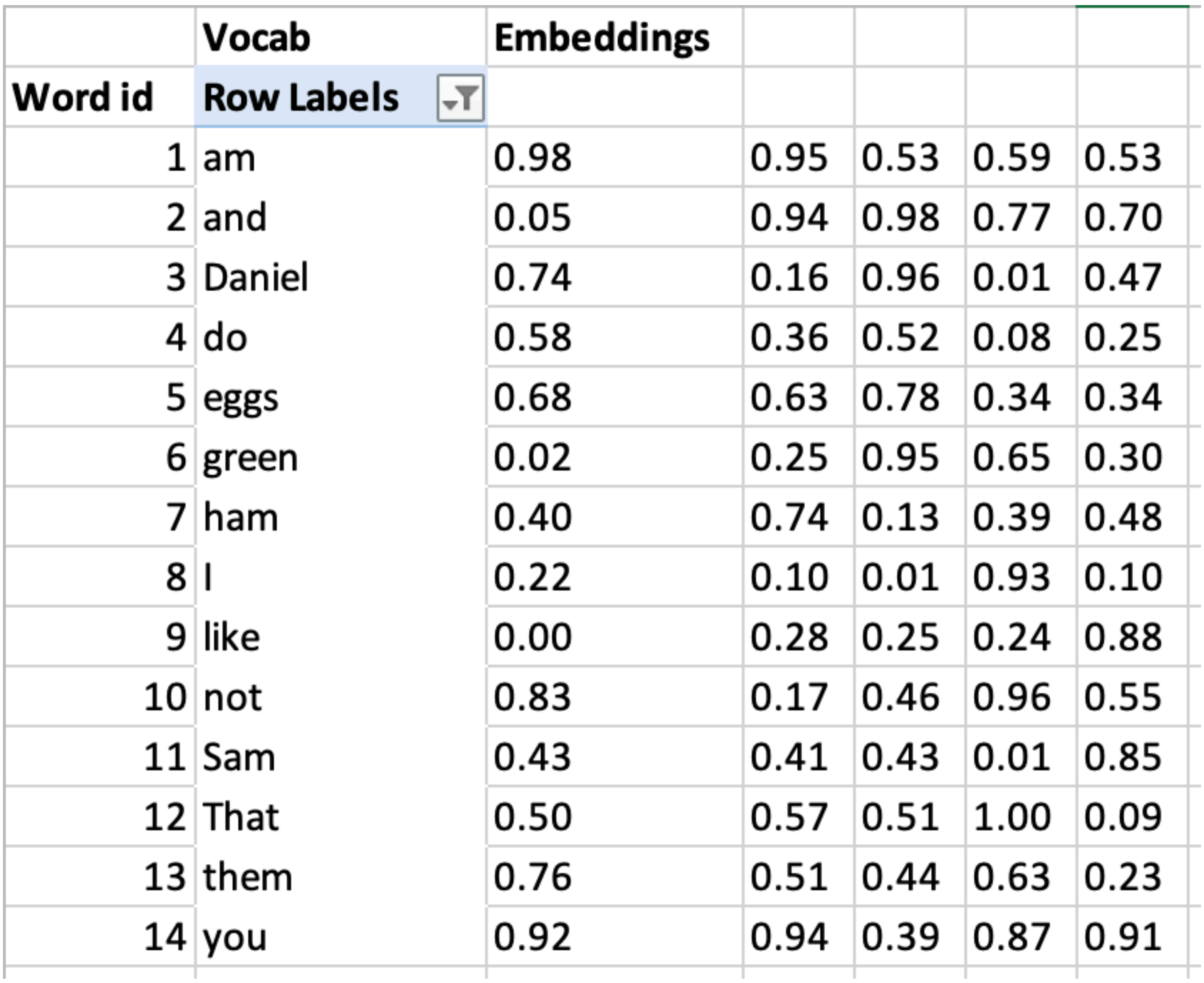
Our `learn.model` is a type of `EmbeddingNN` defined by FastAI. It turns out that `EmbeddingNN` is actually just a `TabularModel`, as it inherits from this class. It has output size of 1 and 0 continuous variables.

Because our model is a `TabularModel` we can easily improve it by adding other information, like user info, movie info, data, time and more... And, we've already seen how to optimise a `TabularModel` too.

In real life, companies generally use a combination of **dot product** and **neural net** in collaborative systems. **Dot product** better takes advantage of the problem domain, generally producing better results (as we can see when comparing our 2 models - the dot product version has a better validation loss). **But**, neural nets allow us to add in useful metadata which can improve recommendations.

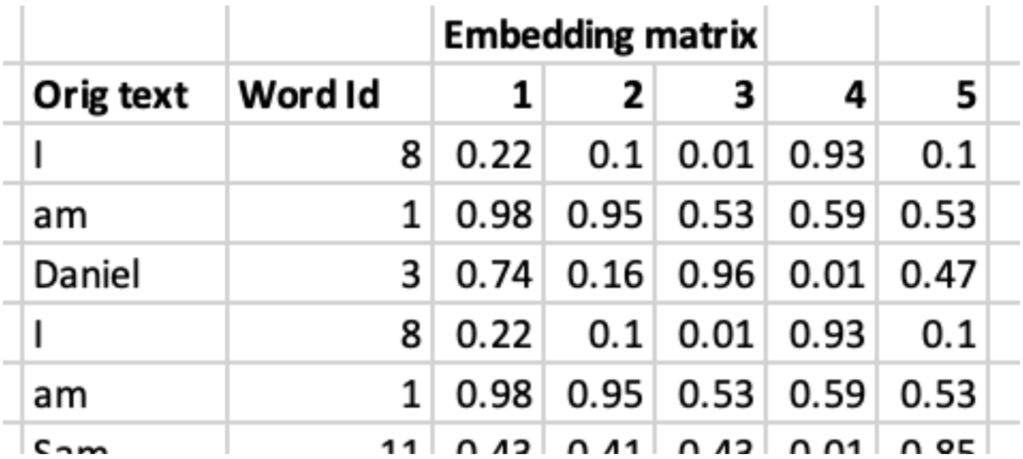
Deep Learning and Embeddings

It turns out that embedding matrices are critical to much of the machine learning we have learnt previously, including with tabular models and NLP.

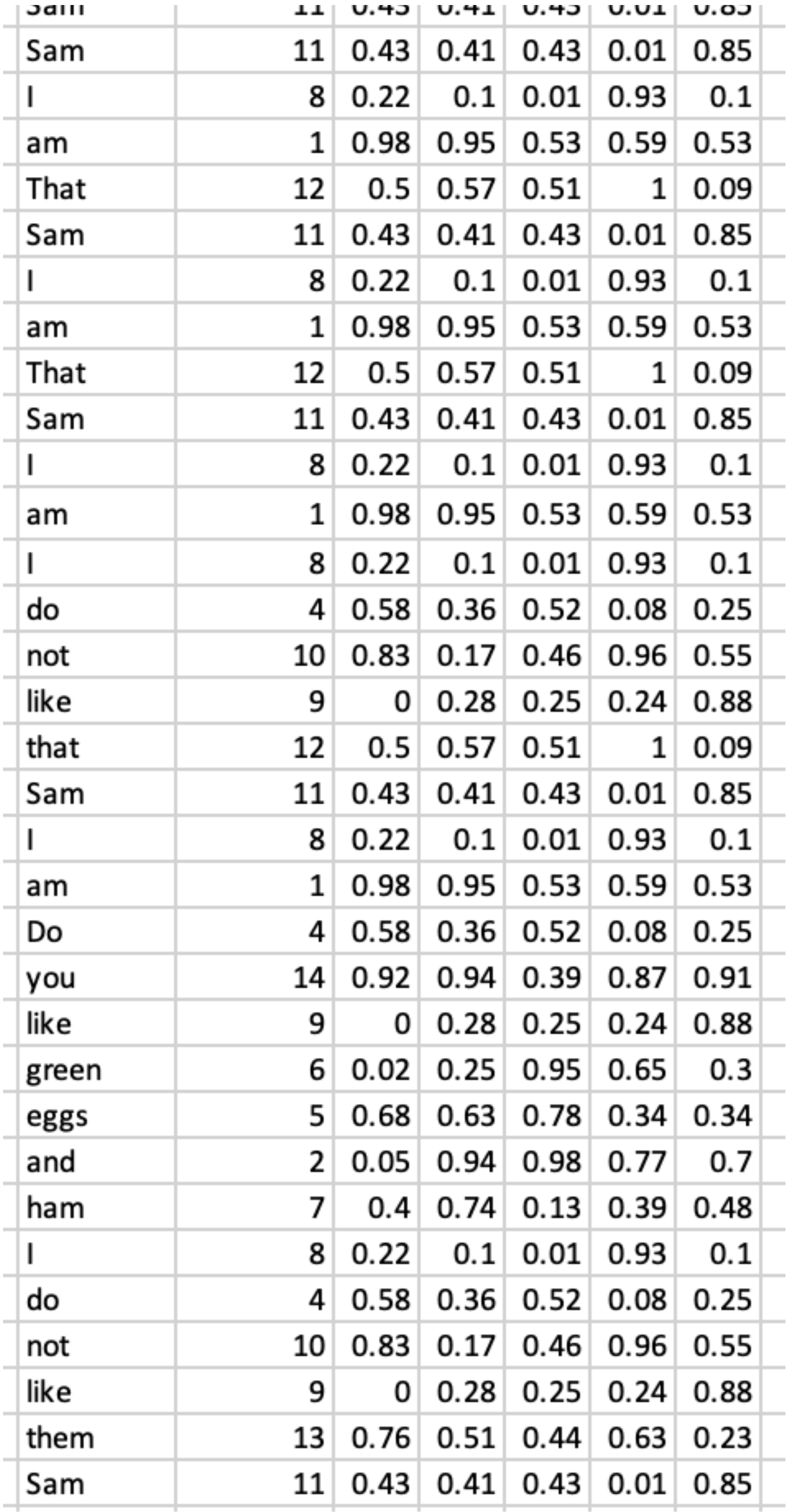


	Vocab	Embeddings				
Word id	Row Labels					
1	am	0.98	0.95	0.53	0.59	0.53
2	and	0.05	0.94	0.98	0.77	0.70
3	Daniel	0.74	0.16	0.96	0.01	0.47
4	do	0.58	0.36	0.52	0.08	0.25
5	eggs	0.68	0.63	0.78	0.34	0.34
6	green	0.02	0.25	0.95	0.65	0.30
7	ham	0.40	0.74	0.13	0.39	0.48
8	I	0.22	0.10	0.01	0.93	0.10
9	like	0.00	0.28	0.25	0.24	0.88
10	not	0.83	0.17	0.46	0.96	0.55
11	Sam	0.43	0.41	0.43	0.01	0.85
12	That	0.50	0.57	0.51	1.00	0.09
13	them	0.76	0.51	0.44	0.63	0.23
14	you	0.92	0.94	0.39	0.87	0.91

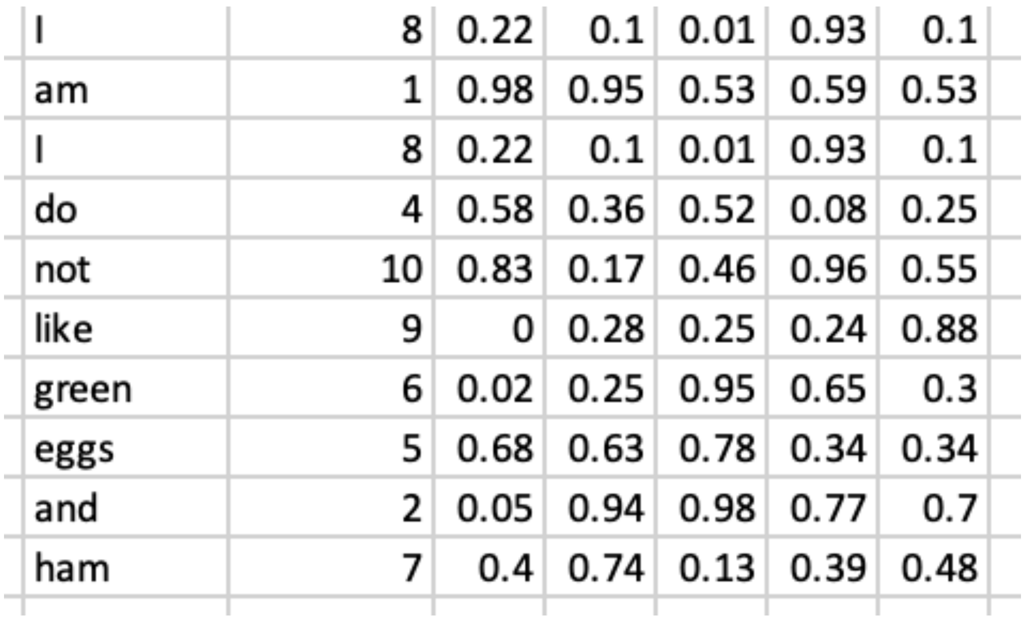
In an NLP model we define a vocabulary (or with a pretrained model, this is predetermined). And for each token in the vocabulary, it will be assigned an **embedding vector**. The list of embedding vectors for our vocabulary will define our **embeddings**. The embeddings will initially be made of random numbers, unless we are using a pretrained model.



		Embedding matrix				
Orig text	Word Id	1	2	3	4	5
I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
Daniel	3	0.74	0.16	0.96	0.01	0.47
I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
Sam	11	0.43	0.41	0.43	0.01	0.85



Sam	11	0.43	0.41	0.43	0.01	0.85
Sam	11	0.43	0.41	0.43	0.01	0.85
I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
That	12	0.5	0.57	0.51	1	0.09
Sam	11	0.43	0.41	0.43	0.01	0.85
I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
That	12	0.5	0.57	0.51	1	0.09
Sam	11	0.43	0.41	0.43	0.01	0.85
I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
I	8	0.22	0.1	0.01	0.93	0.1
do	4	0.58	0.36	0.52	0.08	0.25
not	10	0.83	0.17	0.46	0.96	0.55
like	9	0	0.28	0.25	0.24	0.88
that	12	0.5	0.57	0.51	1	0.09
Sam	11	0.43	0.41	0.43	0.01	0.85
I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
Do	4	0.58	0.36	0.52	0.08	0.25
you	14	0.92	0.94	0.39	0.87	0.91
like	9	0	0.28	0.25	0.24	0.88
green	6	0.02	0.25	0.95	0.65	0.3
eggs	5	0.68	0.63	0.78	0.34	0.34
and	2	0.05	0.94	0.98	0.77	0.7
ham	7	0.4	0.74	0.13	0.39	0.48
I	8	0.22	0.1	0.01	0.93	0.1
do	4	0.58	0.36	0.52	0.08	0.25
not	10	0.83	0.17	0.46	0.96	0.55
like	9	0	0.28	0.25	0.24	0.88
them	13	0.76	0.51	0.44	0.63	0.23
Sam	11	0.43	0.41	0.43	0.01	0.85



I	8	0.22	0.1	0.01	0.93	0.1
am	1	0.98	0.95	0.53	0.59	0.53
I	8	0.22	0.1	0.01	0.93	0.1
do	4	0.58	0.36	0.52	0.08	0.25
not	10	0.83	0.17	0.46	0.96	0.55
like	9	0	0.28	0.25	0.24	0.88
green	6	0.02	0.25	0.95	0.65	0.3
eggs	5	0.68	0.63	0.78	0.34	0.34
and	2	0.05	0.94	0.98	0.77	0.7
ham	7	0.4	0.74	0.13	0.39	0.48

For our input we will split it into tokens and assign each token its **embedding vector** by "searching it up". The resultant matrix will be our **embedding matrix**.

Our embedding matrix will be the input to our neural network. This embedding matrix will be run through matrix multiplication and activation functions. The very parameters used in the embedding matrix will be updated using SGD.

And in this way, we can make predictions using **embeddings**, which can be fine-tuned using our input **embedding matrices**.

And the `tabular_learner()` with FastAI operates in a very similar way. When we pass categorical variables, they are assigned embeddings and an embedding matrix is made. In this way, we have parameters to train - as otherwise how can we use "male" and "female" for example, in SGD?

Bootstrapping and Feedback Loops

The only other discussion to briefly have is about **bootstrapping**, which is a statistical method that involves repeatedly sampling for a dataset with replacement to estimate properties.

With recommendation systems, making initial recommendations for a new user or a new movie is a complex area of statistics. What products do you recommend to your very first user? This is called the **bootstrapping problem**.

One approach would be to take the mean of all the embedding vectors of other users, but the scale of the latent factors is essentially meaningless - if 1 represents science-fiction and 3 represents romance what does the mean of 2 represent?

In fact, in general it is better to design a particular user who represents **average taste**.

You could also create a tabular model to construct your initial embedding vector where you ask the user questions about their taste (like Netflix).

But one thing we have to be very careful of is that a small number of extremely enthusiastic users can overwhelm the preferences of everyone else. The example given is anime watchers - who are a small group who tend to watch and review **a lot** of anime.

This problem can be self-perpetuating. If a small group of users sets the direction of the recommendation system, they will attract more people like themselves to your system, which will amplify the bias. This is a kind of feedback loop that will end up driving away the **real average** user (where the money is), and by the time it is noticed, it is often too late.

It is important to have a gradual and thoughtful rollout that is continually monitored by humans. This will avoid bias and feedback loops, that exponentially amplify and deteriorate the system.

Conclusion

With this, we've seen the progression of collaborative filtering models. From probabilistic matrix factorisation to deep learning, it is evident that collaboration systems are just another type of matrix multiplication with activation functions - like all machine learning.

Collaborative filtering models are a fascinating use of mathematics but also have the potential to amplify bias, create feedback loops and destroy systems. It is critical to be thoughtful when employing them.

Thanks for reading my notebook, and again, a big thanks to FastAI from whom the code is adapted.