



Collaborative Filtering

Ryan Gillard

Hi I'm Ryan, I'm a machine learning scientist at Google, and I love applying math and machine learning to big data to better make sense of the world. **Now that we've** learned the basics of recommender systems and how content-based variants work, in this module we will dive into the powerful recommendation technique of collaborative filtering.

Learn how to...

Build a collaborative filtering recommendation system using WALS.

Understand user-item interaction data and find similarities.

Write an input function for the WALS Matrix Factorization TensorFlow Estimator.

Batch predict recommendations.

Create a productionized version.

Finally, we'll learn how to create a productionized version and go over some of the strengths and weaknesses of our model and ways we can help address some of the issues.

Content-based recommendations
use similarities between items in an
embedding space



We've learned about how content-based recommendation systems can use properties and features of items to make recommendations. If given items that a user likes, we can search an embedding space for similar items—in other words, items in the local neighborhood of the item factor embedding space—using some distance metric. This can be great, because it doesn't need data about other users and can recommend niche items. However, it usually requires domain knowledge, it only makes safe recommendations and stays in our local bubble of embedding space, and it doesn't try things off our limited data manifold.

For instance, the similarities between the properties of fruit smoothies—color, taste, acidity, texture, etc.—can all be represented by points for each item in a multi-dimensional embedding space.

<https://pixabay.com/en/smoothies-juice-fruits-fruit-ripe-2253430/> cc0

What if we don't know the best factors to compare with?

Proprietary + Confidential



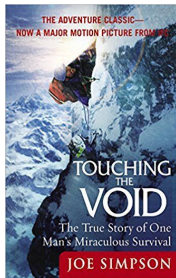
Google Cloud

What if we don't know the best factors to compare items with? We often think we know the factors that influence certain behaviors or lead to certain preferences, but sometimes we can be wrong. Let's see an example.

<https://pixabay.com/en/todo-list-despaired-man-person-sad-297195/> cc0

What if we don't know the best factors to compare with?

Proprietary + Confidential



1988



1999

Google Cloud

“Touching the Void” was published in 1988, and didn’t sell many copies. Then “Into the Air” came out in 1999 and was a bestseller. A bookseller noticed that many users who were buying “Into thin Air” also bought “Touching the Void”, and started recommending it to buyers of “Into thin Air.” Soon sales of Touching the Void started to take off, and now that book outsells Into thin Air by more than two to one.

Note that recommendations were made without knowing anything about the users, other than their buying behavior, and without knowing anything about the content of the two books. This is an example of “collaborative filtering”, and shows the power of that technique for making recommendations.

Collaborative filtering learns latent factors and can explore outside user's personal bubble

Proprietary + Confidential



Google Cloud

Instead of requiring lots of domain knowledge, collaborative filtering can learn the latent factors contained in the user-item interaction data we have. Collaborative filtering can also give recommendations some serendipity, because even though one user's data manifold, or neighborhood, may be small and only in one small bubble of embedding space, other users may have the first user's manifold as a subset of theirs. This means that the users are similar to each other, and perhaps the first user might actually like to branch out and see things that the other users also liked.

<https://pixabay.com/en/bubble-soap-bubbles-colorful-1891638/> cc0

Collaborative filtering recommendations use similarities between **items and user** Proprietary + Confidential
simultaneously in an embedding space



Google Cloud


Collaborative filtering solves two problems at the same time between users and items. It uses similarities between items and users simultaneously. Obviously, a hybrid approach of content-based, collaborative filtering and knowledge-based recommendations usually works best, which we will cover in the next module, but for now let's look at the awesome power of collaborative filtering.









For example, instead of just having items, such as drinks, represented as points in a multi-dimensional embedding space, users themselves too can be represented along each dimension: two embeddings within the same space.

<https://pixabay.com/en/drinks-alcohol-cocktails-alcoholic-2578446/> cc0

Start from a user-interaction matrix where rows are users and items are columns

Proprietary + Confidential



	 Harry Potter	 The Triplets of Belleville	 Shrek	 The Dark Knight Rises	 Memento
					
					
					
			?		

Google Cloud

Collaborative filtering uses the interactions between users and items to find the similarities between users and items. This can be represented as a matrix, although usually a very sparse one, because usually there are large numbers of users and items, and most have never interacted. Therefore, to get around this sparsity problem, collaborative filtering usually uses matrix factorization. Matrix factorization collaborative filtering starts from a ratings or user-interaction matrix, where the rows are users and items are columns. In this example, there are four users and five items, which are movies that the users may have seen on our website. Our goal is to recommend movies to each user that they would like to see. So should user 4 see the movie Shrek or not? Well that depends on the ratings.

Sometimes these ratings are explicit

Proprietary + Confidential










	 Harry Potter	 The Triplets of Belleville	 Shrek	 The Dark Knight Rises	 Memento
	4		3	1	
		5			4
	1	3	4		
			?	5	5

Google Cloud

Sometimes these ratings are explicit. This could be the number of stars, the number of thumbs up, or maybe just a simple like or dislike button click. The main point here is that a user is intentionally, explicitly leaving feedback for that item. You'll notice that there are a lot of blank squares, and we'll learn how to handle that later. In this example, the scores for each movie could be 1 through 5, where 1 is strong dislike, 3 is neutral, and 5 is strong like.

Most often these ratings are implicit

Proprietary + Confidential










	 Harry Potter	 The Triplets of Belleville	 Shrek	 The Dark Knight Rises	 Memento
	✓		✓	✓	
		✓			✓
	✓	✓	✓		
			?	✓	✓

Google Cloud

Explicit feedback is usually difficult to get, but thankfully there is also implicit feedback, which is still more information than if there was no feedback. Implicit feedback is different from explicit feedback because it is not intentionally given as a means of rating the item the user has interacted with. However, there was some type of interaction, and from that we can infer whether the user had a positive or negative experience. This could be whether someone viewed a video, how long they watched a video, if a user spent a lot of time on a page, if they clicked certain areas or buttons on the page, etc. Much of the time we cannot give a real valued score, as in the last table. For instance, in this example we simply assign a checkbox, which will end up numerically as a 1, to indicate that user i watched movie j . Sometimes, there is both explicit and implicit data for user-item interactions that can be leveraged for recommendations.

This matrix ties an interaction between a user and a video

Proprietary + Confidential

	0	Time	User #	Video #	1
					
	Harry Potter	The Triplets of Belleville	Shrek	The Dark Knight Rises	Memento
	✓		✓	✓	
		✓			✓
	✓	✓	✓		
			?	✓	✓

Google Cloud

If we think about how our user-movie data might be stored in a database, there might be an index column, a timestamp column, a user ID column, a movie ID column, and then a column with the rating. In this implicit feedback example, the implicit rating from user 2, for the movie Memento, was a 1, meaning that whatever the interaction was, that interaction was flagged as implicit positive feedback.

Quiz

If we were creating a YouTube video recommender system where we had “like” and “dislike” data and also the duration a video was watched, which feedback would be considered explicit and which would be considered implicit?

	Like/dislike	Watch duration
A	Implicit	Explicit
B	Implicit	Implicit
C	Explicit	Explicit
D	Explicit	Implicit

Now it's time to test your knowledge about the types of feedback. If we were creating a YouTube video recommender system, where we had “like” and “dislike” data, and also the duration a video was watched, which feedback would be considered explicit, and which would be considered implicit? Was the like and dislike data explicit or implicit feedback? Was the watch duration explicit or implicit feedback?

Quiz

If we were creating a YouTube video recommender system where we had “like” and “dislike” data and also the duration a video was watched, which feedback would be considered explicit and which would be considered implicit?

	Like/dislike	Watch duration
A	Implicit	Explicit
B	Implicit	Implicit
C	Explicit	Explicit
D	Explicit	Implicit

The correct answer is D!

The like and dislike data is explicit, because a user is purposefully giving either positive feedback by clicking like, or negative feedback by clicking dislike. They are telling us explicitly. Whereas, watch duration is more like implicit feedback. The user isn't purposefully telling us whether they liked the video, but we can make an inference. Perhaps users that like the video, will watch it longer or even to full completion, whereas users that don't like the video may quickly change to something else, and thus have a short duration.

We can organize items by similarity in one dimension

Proprietary + Confidential



Google Cloud

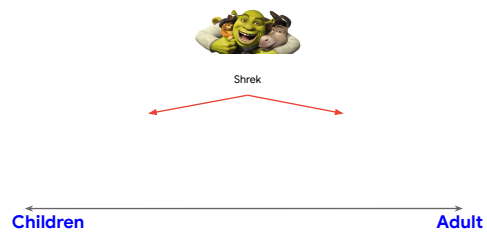
Content-based recommendations used embedding spaces for items only, whereas now for collaborative filtering, we are learning where users and items fit within a common embedding space along dimensions that they have in common. We can choose a number of dimensions to represent them in, either using human-derived features or using latent features that are under the hood of our preferences which we will learn how to find very soon.

Each item has a vector within this embedding space that describes the item's amount of expression of each dimension. Each user also has a vector within this embedding space that describes how strong their preference is for each dimension. For now, let's keep things simple and keep things just one dimensional looking at items, and we'll get back to multi-dimensional embeddings later and how users fit in. We'll start simple and then build ourselves up.

We could organize items, let's say movies, by similarity in one dimension, for example of where they fall on the spectrum of movies for children to movies for adults.

We can organize items by similarity
in one dimension

Proprietary + Confidential



Google Cloud

Let's say we have the movie Shrek. We want to know where it falls on this children to adult movies spectrum.

We can organize items by similarity
in one dimension

Proprietary + Confidential



Shrek

←
Children

Adult →

Google Cloud

Using our own built-up knowledge, we would say that Shrek would probably go to the far left, because it is an animated movie mainly for children.

We can organize items by similarity
in one dimension

Proprietary + Confidential



Google Cloud

What about the movie The Dark Knight Rises? Where do we think it goes along our axis?

We can organize items by similarity
in one dimension

Proprietary + Confidential

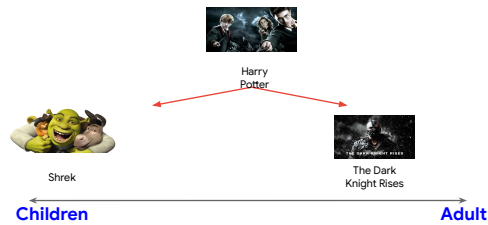


Google Cloud

Well, the Dark Knight Rises we know is definitely a more adult movie than Shrek so it should go to the right of Shrek on the spectrum.

We can organize items by similarity
in one dimension

Proprietary + Confidential

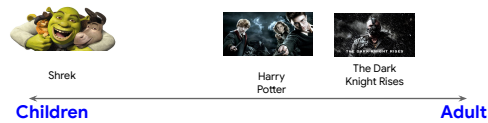


Google Cloud

Now that we have two movies placed, things start to get harder because we have three choices to decide from. Here we have Harry Potter that we want to place along our spectrum. Does it go to the left of Shrek, to the right of The Dark Knight Rises or somewhere in between?

We can organize items by similarity
in one dimension

Proprietary + Confidential

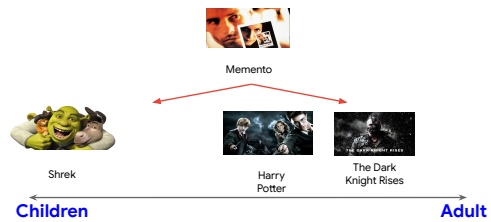


Google Cloud

From our internal representations of these movies, we can guess that Harry Potter goes between Shrek and The Dark Knight Rises. **It definitely has more adult themes than Shrek but still not as much violence or adult themes as the Dark Knight Rises. Exactly where in between it should go we will get to soon.**

We can organize items by similarity
in one dimension

Proprietary + Confidential

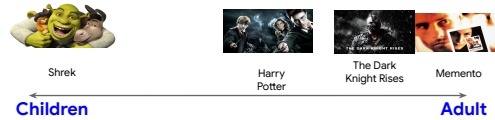


Google Cloud

All right: three movies placed, more to go. We now have four choices of where to place the movie Memento. What to do?

We can organize items by similarity
in one dimension

Proprietary + Confidential

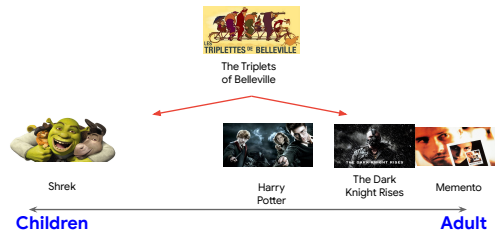


Google Cloud

Memento has some very adult themes, and our knowledge would probably place it even farther right than The Dark Knight Rises.

We can organize items by similarity
in one dimension

Proprietary + Confidential

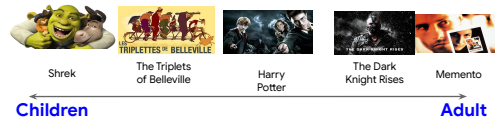


Google Cloud

Lastly, we want to place The Triplets of Belleville. As you can see, every time we add a movie, the number of positions we can place it in relative to the other movies increases by 1. Imagine if we had thousands and thousands of movies.

We can organize items by similarity
in one dimension

Proprietary + Confidential

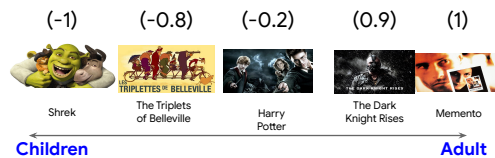


Google Cloud

The Triplets of Belleville is pretty far along the children's movie scale, but not quite as much as Shrek, so this is where we will put it. However, this is all very qualitative with just some unknown relative position along a number line. Can we make this a little more quantitative, maybe with some numbers?

We can organize items by similarity in one dimension

Proprietary + Confidential

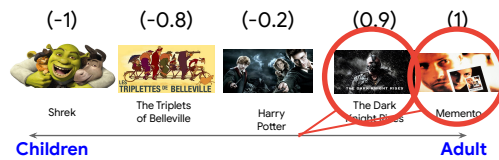


Google Cloud

We've now added values on where we think these movies lie along our number line. High negative values are considered very much children's movies, and high positive values are considered very adult. Movies close to zero are basically halfway between what we'd say is very childish and very adult. We've essentially created a 1D embedding space with our coordinate system, and our 5 points could have these values, indicating where they sit on the children's movie/adult movie spectrum. Now, you can calculate similarities via distance metrics between the points in our coordinate system. Those that are closer, or have a smaller distance, are more similar, at least with respect to this children/adult factor we chose.

We can organize items by similarity
in one dimension

Proprietary + Confidential

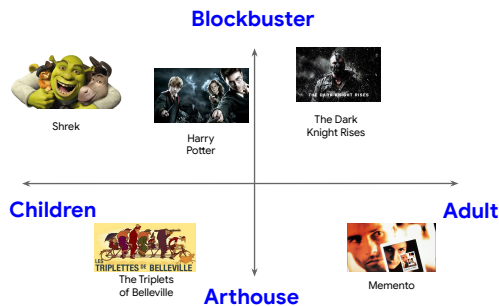


Based on these values along this axis, "The Dark Knight Rises" and "Memento" are the most similar movies. However, even if two movies are very close in this one-dimensional projection using a child/adult spectrum, in other projections, using other features of the movies, they could be extremely far apart.

Google Cloud

We can organize items by similarity
in two dimensions

Proprietary + Confidential

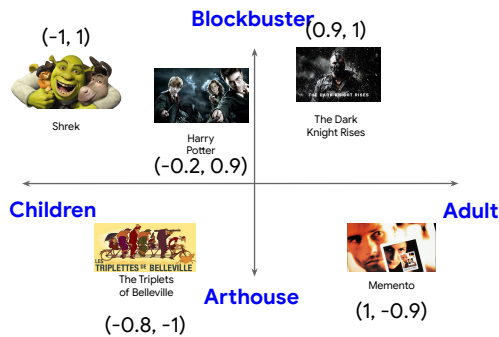


Google Cloud

Let's now add a second dimension to calculate similarities between movies, for example where movies fall on the arthouse/blockbuster spectrum. Not only have we arranged the movies along the children/adult movie spectrum, they are now spread out along the second dimension. As we add dimensions, our embedding spaces grow exponentially larger and become sparser and sparser with large voids between points in the embedding space.

We can organize items by similarity
in two dimensions

Proprietary + Confidential

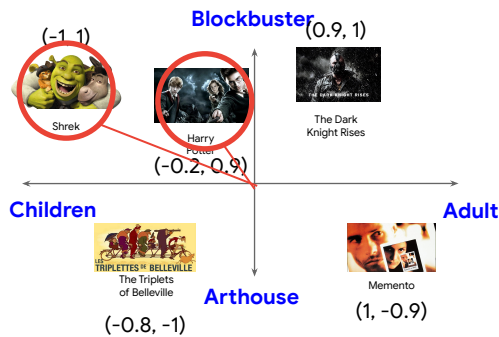


Google Cloud

In this 2D embedding space coordinate system we created, our 5 points could have these values indicating where they sit on both the children's movie/adult movie and the arthouse/blockbuster spectrums. As we can see, the movies' points in our two-dimensional embedding space have the same coordinate values as along our original children/adult spectrum; however they have now been spread out along our new arthouse/blockbuster spectrum. The points haven't changed at all, but our coordinate system has! Coordinate systems aren't important; they are just a way to describe points within space.

We can organize items by similarity
in two dimensions

Proprietary + Confidential



Google Cloud

Based on these values in our new two-dimensional embedding space, "The Dark Knight Rises" and "Memento" are no longer the closest, or most similar movies, at least not using the dot product distance metric. The two closest are now "Harry Potter" and "Shrek." In fact, our previous closest aren't now even the second closest, which goes to "Harry Potter" and "The Dark Knight Rises." This is due to how close these three movies are along the blockbuster axis, which is dominating in the dot product calculation.

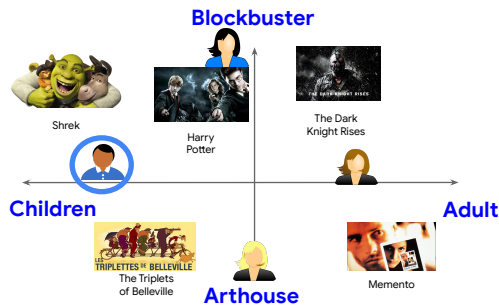
Proprietary + Confidential

Google Cloud

Let's see how this looks in our embedding space.

Item retrieval in two dimensions based on user

Proprietary + Confidential

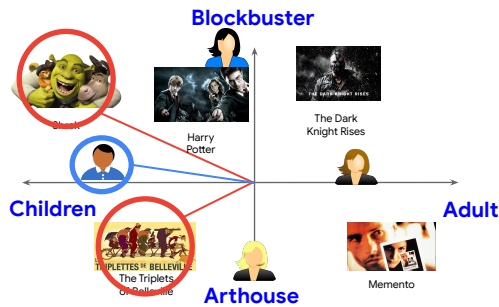


Google Cloud

Going back to our coordinate system plot from before, we have our movies plotted in the two-dimensional embedding space at their coordinate values based on where they fall within the two factors we chose. We've also now added the users at their respective coordinates. So, what should we recommend to each user? Let's look at the leftmost user.

Item retrieval in two dimensions based on user

Proprietary + Confidential

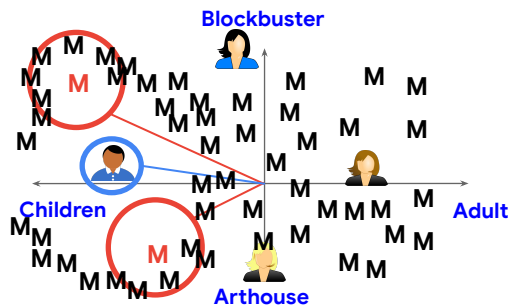


Google Cloud

To find the answer to what movie the leftmost user should watch, all we need to do is take the dot product between the user and each movie. In this example, we chose to return the top 2 highest movies, which end up being Shrek and The Triplets of Belleville, because they had the two highest dot products for that user across all of the other movies. Now, this might seem obvious from looking at the graph; however it is not as easy for a computer to "see" this, because our trained eyes and the neural networks in our brains have had decades of experience to "see."

Item retrieval in two dimensions based on user

Proprietary + Confidential

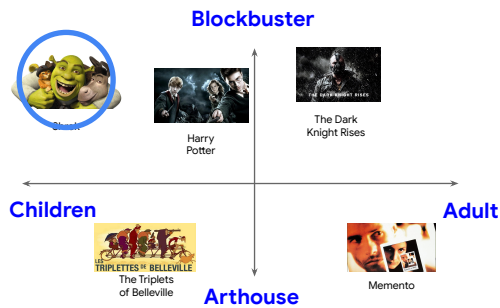


Google Cloud

Yet, this is only with two dimensions and with few items. If we were to increase the dimensionality, we humans would have a very difficult time discerning which movies are the closest; we would probably stop inspecting and just take the dot product.

Item retrieval in two dimensions based on item

Proprietary + Confidential

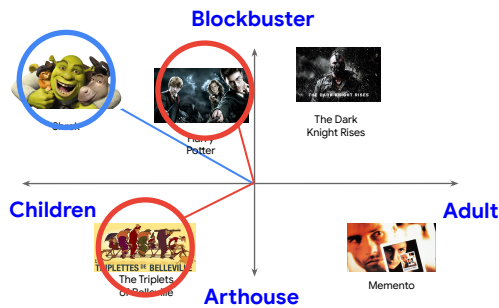


Google Cloud

The same can be done for items! Let's say we want to see what the movies closest to Shrek are.

Item retrieval in two dimensions based on item

Proprietary + Confidential



Google Cloud

Instead of taking the dot product between a user and all of the items, we take our item of interest, and take the dot product with all of the other items. In this example, we once again are returning the top 2, which turns out to be Harry Potter and The Triplets of Belleville. You can do the exact same to find users similar to a user of interest by taking the dot product between users.

Each user and item is a d -dimensional point within an embedding space

So as we can see, users and items can be represented as d -dimensional points within an embedding space. In our example, we chose some human experience-based factors for each of our two dimensions, and humans manually scored each movie, giving it a value for each dimension along its axis. We did the same for evaluating each user's coordinates in our embedding space. Well, this works great for toy examples with 4 users and 5 items, but it quickly becomes unscalable as more users and items are added to the system. Furthermore, as we saw in the last course when psychologists used what they thought would be good features for embeddings turned out not to be that great, here our human-derived features may not be the best choice. So how can we choose then?

Each user and item is a d -dimensional point within an embedding space

Embeddings can be **learned from data**

Fortunately for us, the embeddings can be learned from data! Instead of defining the factors that we will assign values along in our coordinate system, we use the user-item interaction data to learn the latent factors that best factorize the user-item interaction matrix into a user-factor embedding and item-factor embedding.

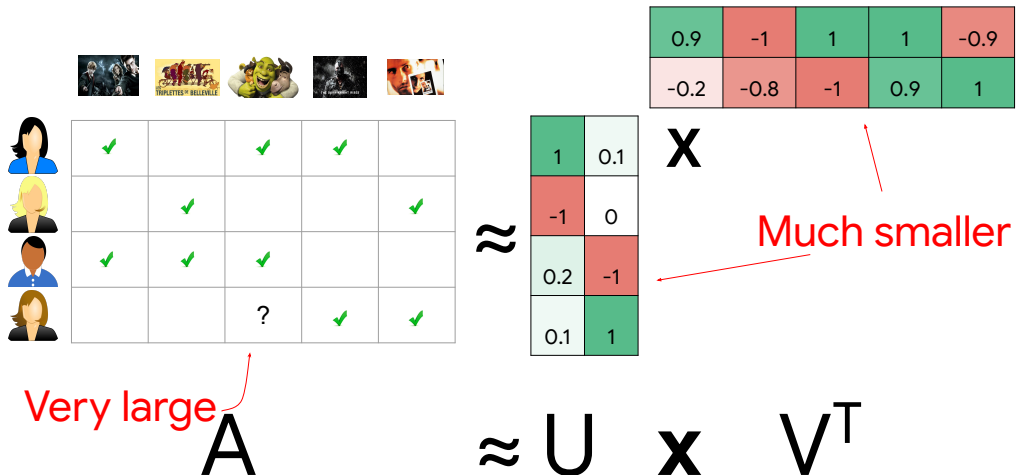
Each user and item is a d -dimensional point within an embedding space.

Embeddings can be **learned from data**.

We're compressing the data to find the best generalities to rely on, called ***latent factors***.

We are essentially compressing the data, our very sparse interaction matrix, and relying on the generalities within, which are the latent factors. You might be reminded of PCA, SVD, or other dimensionality reduction techniques like we covered in the last course for reusable embeddings.

The factorization splits this matrix into row factors and column factors that are essentially user and item embeddings



The user-interaction matrix factorization splits this very large user-by-item matrix into two smaller matrices of row factors, which are users in this case, and column factors, which are movies in this case. You can think of these two factor matrices as essentially user and item embeddings. Now you might be wondering why matrix U has two columns and matrix V has 2 rows? Well this is simply the dimensionality of our learned embedding space of our latent features. Remember, a latent feature is a feature that we are not directly observing or defining, but are instead inferring through our model from the other variables that are directly observed, namely the user-item interaction pairs. Just like for other dimensionality reduction techniques like PCA and SVD, the number of latent features is a hyperparameter that we can use as a knob for the tradeoff between more information compression and more reconstruction error from our approximated matrices.

 10 thousand movies

50 million users



500
billion!

Just how much space are we saving? Let's say there are u users and i items, which gives our user item interaction matrix, A , a size of u by i . If our website has 50 million users and 10 thousand movies, that is already 500 billion interaction pairs!



10 thousand movies

50 million users




$k(\text{users} + \text{movies})$

On the other hand, if we can we have k latent features, then U will have u by k elements, and V will have k by i elements. The total space is just the number of users plus the number of movies times the number of latent features. Even with 10 latent features in this case, it is still 1000 time less space than before.

 10 thousand movies

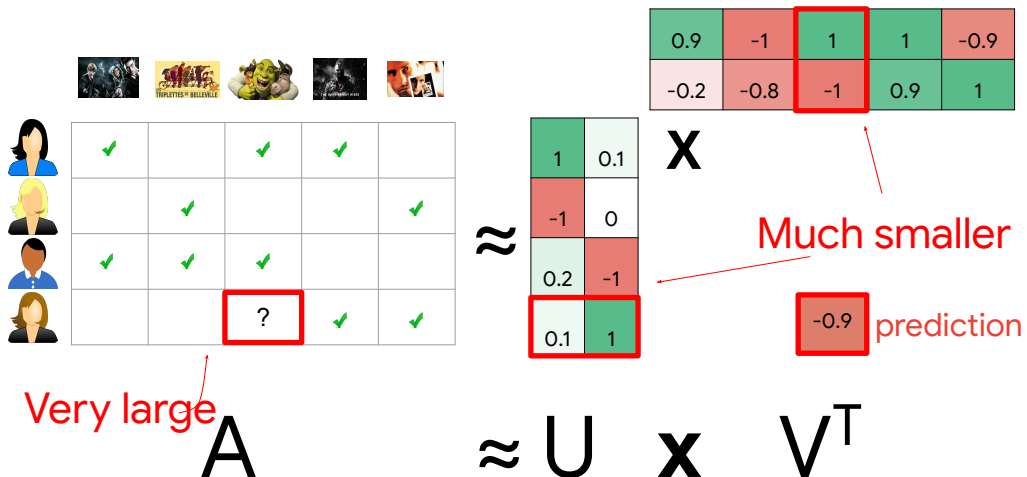
50 million users

$$k < \frac{U * V}{2(U + V)}$$


As long as the number of latent features is less than half the harmonic mean of the number of users and the number of items, this will save space. For this hypothetical website, that would be almost 10 thousand latent features, 9998 to be precise, which is almost as if we hadn't embedded the movies at all, since there were 10 thousand movies to begin with. Each movie is essentially its own feature!

$$k < \frac{U * V}{2(U + V)}$$










The factorization splits this matrix into row factors and column factors that are essentially user and item embeddings



To make a prediction, we simply take the dot product of a user with the item factors, or an item with the user factors, and we will get a prediction value for that particular rating. In this example, we want to predict the rating for user 4 for the movie Shrek, which is the third column, so we multiply 0.1 with 1 and 1 with -1 to get a value of -0.9. Harry Potter gets a prediction of -0.11, and The Triplets of Belleville also gets a prediction of -0.9. If we are recommending the highest predicted unwatched movie to user 4 out of the movies shown, then it would be Harry Potter, because it has the highest predicted score.

Quiz

Based on this user-item interaction matrix, which movie should user 4 watch?










						
		0.9	-1	1	1	-0.9
		-0.2	-0.8	-1	0.9	1
1	0.1		✓		✓	✓
-1	0			✓		✓
0.2	-1		✓	✓	✓	
0.1	1				✓	✓

- A. Harry Potter
- B. Triplets of Belleville
- C. Shrek
- D. The Dark Knight Rises

Now it's time to test your knowledge about finding the best recommendations based on user and item similarities. Based on this user-item interaction matrix of users and movies, which movie should user 4 watch?

Quiz

Based on this user-item interaction matrix, which movie should user 4 watch?

						
		0.9	-1	1	1	-0.9
		-0.2	-0.8	-1	0.9	1
1	0.1		✓		✓	✓
-1	0			✓		✓
0.2	-1		✓	✓	✓	
0.1	1		-0.11		✓	✓

- A. Harry Potter **-0.11**
- B. Triplets of Belleville **-0.9**
- C. Shrek **-0.9**
- D. The Dark Knight Rises **1.0**

The correct answer is A!

To find this answer, we should have calculated all of the dot products between user 4's user-factor vector and all of the movie-factor vectors. User 4's predicted rating for Harry Potter via the dot product is -0.11.











Quiz

Based on this user-item interaction matrix, which movie should user 4 watch?

			0.9	-1	1	1	-0.9
			-0.2	-0.8	-1	0.9	1
			✓		✓	✓	
				✓			✓
			✓	✓	✓		
			-0.11	-0.9	-0.9	1	0.91

- A. Harry Potter **-0.11**
- B. Triplets of Belleville **-0.9**
- C. Shrek **-0.9**
- D. The Dark Knight Rises **1.0**

The Dark Knight Rises is 1, and Memento is 0.91.

					
	0.9	-1	1	1	-0.9
	-0.2	-0.8	-1	0.9	1
	1	0.1	0	0.2	-1
	0.1	1	-0.11	-0.9	-0.9
	0.9	1	0.91	1	0.91

- Google Cloud

Remember, we are just recommending the top 1 movies in this question, so we should return the max dot product between user 4's user-factor vector of (0.1, 1) with all of the movie-factor vectors. So why is the answer Harry Potter, when The Dark Knight Rises has the highest dot product out of all the movies?

Quiz

Based on this user-item interaction matrix, which movie should user 4 watch?

		0.9	-1	1	1	-0.9
		-0.2	-0.8	-1	0.9	1
1	0.1		✓		✓	✓
-1	0			✓		✓
0.2	-1		✓	✓	✓	
0.1	1		-0.11	-0.9	-0.9	1

- A. Harry Potter **-0.11**
- B. Triplets of Belleville **-0.9**
- C. Shrek **-0.9**
- D. The Dark Knight Rises **1.0**

The reason is because the user has already rated The Dark Knight implicitly in this example, and thus we don't want to recommend that the user watch a movie they've already seen, because most users know whether they liked something in the past and instead want to watch something new.

Quiz

Based on this user-item interaction matrix, which movie should user 4 watch?

		0.9	-1	1	1	-0.9
		-0.2	-0.8	-1	0.9	1
1	0.1		✓		✓	✓
-1	0			✓		✓
0.2	-1		✓	✓	✓	
0.1	1		-0.11	-0.9	-0.9	✓

- A. Harry Potter **-0.11**
- B. Triplets of Belleville **-0.9**
- C. Shrek **-0.9**
- D. The Dark Knight Rises **1.0**

Therefore, we must filter out all of the movies that have already been rated, and then, from the resulting subset, choose the top 1.

Quiz

Based on this user-item interaction matrix, which movie should user 4 watch?





			0.9	-1	1	1	-0.9
			-0.2	-0.8	-1	0.9	1
			✓		✓	✓	
				✓			✓
			✓	✓	✓		
			-0.11	-0.9	-0.9	✓	✓






- A. Harry Potter **-0.11**
- B. Triplets of Belleville **-0.9**
- C. Shrek **-0.9**
- D. The Dark Knight Rises **1.0**

This results in Harry Potter, although with a negative dot product, which means that the movie is predicted to be below neutral sentiment but has the highest dot product value out of the unwatched movies.

User and item matrix and embeddings

Proprietary + Confidential

0.9	-1	1	1	-0.9
-0.2	-0.8	-1	0.9	1
				

				
✓		✓	✓	
	✓			✓
✓	✓	✓		
		?	✓	✓

Google Cloud

So we've now learned how the user-item interaction works and how we can create user embeddings and item embeddings within the same latent space which can be learned from data. So what machine learning algorithms can help us with this?

Collaborative filtering is usually carried out using matrix factorization

Proprietary + Confidential

- Factorize user-interactions matrix into user-factors and item-factors.
- Given user ID, multiply by item-factors to get predicted ratings for all items.
- Return top k rated items for this user.

Google Cloud

Collaborative filtering uses a user-item interaction matrix that is a 2D table that provides ratings for user-item pairs. These matrices are usually very sparse because there is a large number of users AND a large number of items, which creates a MASSIVE cartesian product. These gargantuan matrices normally need to be shrunk down to a more tractable size, for example through matrix factorization.

With this method, we first factorize the user-interactions matrix into two smaller matrices: user-factors and item-factors. If given a user ID, we can multiply by item-factors to get the predicted ratings for all items. Also, vice versa, if given a movie ID, we can multiply by user-factors to get the predicted rating for all users. Now that we have all of the predicted ratings, we can return the top k rated items back to the user, who can now possibly make a better decision.

$$A \approx U \times V^T$$

$$\min_{U,V} \sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2$$

We want to factorize our user-interaction matrix A into the two much smaller matrices, user-factor matrix U and item-factor matrix V . Because this is an approximation, we want to minimize the squared error between the original user-interaction matrix, and the product of the two factor matrices.

$$\min_{U,V} \sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2$$

$$A \approx U \times V^T$$

$$\min_{U,V} \sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2$$



This resembles a least squares problem, and there are several ways to solve it, but which one should we choose?

$$\min_{U,V} \sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2$$

Scale <https://pixabay.com/en/silhouette-scales-justice-scale-3267887/> (cc0)

Stochastic gradient descent (SGD)

One method is to use the classic learning algorithm, stochastic gradient descent, or SGD. Let's go through the **strengths and weaknesses** of using SGD.

Stochastic gradient descent (SGD)

Flexible

SGD is very flexible. It can be used for many different types of problems and loss functions, and we probably already have a lot of experience using it, so plus one to the strengths.

Stochastic gradient descent (SGD)

Flexible
Parallel

SGD is also parallel, which is awesome, because then we can scale out our matrix factorization and quickly tackle much larger problems. Another plus one to the strengths.

Stochastic gradient descent (SGD)

Flexible
Parallel
Slower

SGD, however, is one of the slower algorithms we could use, because it possibly has to go through many iterations until it learns a good fit. Plus one to the weaknesses now.

Stochastic gradient descent (SGD)

Flexible

Parallel

Slower

Hard to handle unobserved interaction pairs

Remember all of the unobserved user-item interaction pairs we had in the prior tables? Well unfortunately, user-item interaction data tends to be very sparse, especially for problems with a large number of users and columns, so we want an algorithm that can handle this lack observations. Unfortunately, SGD has a hard time handling these, so that is another con for SGD, bringing us to a total of two strengths and two weaknesses.

Alternating least squares (ALS)

Another algorithm we can use is alternating least squares, or ALS, which alternately solves for U holding V constant, and then solves for V holding U constant. Let's look at the strengths and weaknesses of this method now.

Alternating least squares (ALS)

Least Squares Only

Well unfortunately, as the name suggests, this algorithm only works for least squares problems, so that is plus one for weaknesses. However, luckily for this particular problem we have a least squares loss that we are trying to minimize, so this is still a viable option.

Alternating least squares (ALS)

Least Squares Only
Parallel

Just like SGD, ALS is a parallelizable algorithm, therefore we can also scale out our problem to handle much larger data in a shorter amount of time. Plus one to the strengths.

Alternating least squares (ALS)

Least Squares Only
Parallel
Faster

ALS is usually much faster at convergence than SGD. This makes sense, because SGD is more of a generalist algorithm, which gives it its flexibility; however ALS is a specialist algorithm for least squares problems, which this is, so it is somewhat optimized to solve these types of problems faster. Another plus one to the strengths.

Alternating least squares (ALS)

Least Squares Only

Parallel

Faster

Easy to handle unobserved interaction pairs

Lastly, unlike SGD, ALS or a close variant can easily handle unobserved interaction pairs, which is great news because those are usually extremely abundant in our interaction matrix. That's another strength for ALS, bringing us to a total of three strengths and only one weakness.

Unobserved pairs



	Harry Potter	The Triplets of Belleville	Shrek	The Dark Knight Rises	Memento
User 1 (Black hair, blue shirt)	1		1	1	
User 2 (Blonde hair, black shirt)		1			1
User 3 (Dark skin, blue shirt)	1	1	1		
User 4 (Brown hair, black shirt)				1	1

Now exactly what happens to the unobserved interaction pairs? Let's look at our earlier interaction matrix, specifically the implicit feedback version. As you can see, we've replaced the check marks with ones instead. So how do the different algorithms treat the missing interaction pairs data?

Singular value decomposition (SVD)

$$|A - UV^T|^2$$

	 Harry Potter	 The Triplets of Belleville	 Shrek	 The Dark Knight Rises	 Memento
	1	0	1	1	0
	0	1	0	0	1
	1	1	1	0	0
	0	0	0	1	1

One simple way to solve this is to perform the singular value decomposition as we did in a previous course for latent semantic analysis. This matrix factorization method, however, requires all real numbers in the matrix it is decomposing; therefore, the unobserved pairs are given values of zero. You might think that this is not significant, but it greatly changes what the data means, because we are essentially making a large assumption and imputing zeroes wherever data was missing. This can lead to poor performance in making recommendations. What else can we try instead?

$$|A - UV^T|^2$$

Matrix factorization ALS

$$\sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2$$



	Harry Potter	The Triplets of Belleville	Shrek	The Dark Knight Rises	Memento
	1		1	1	
		1			1
	1	1	1		
				1	1

Using matrix factorization for collaborative filtering, such as ALS, we just ignore the missing values. Notice that in the loss formula we are minimizing, we are only summing over the observed instances. This still isn't a great fix, but with a minor tweak, we can obtain something that usually works very well.

$$\sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2$$

Weighted ALS (WALS)

$$\sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2 + w_0 * \sum_{(i,j) \notin \text{obs}} (0 - U_i V_j)^2$$



	Harry Potter	The Triplets of Belleville	Shrek	The Dark Knight Rises	Memento
	1	0	1	1	0
	0	1	0	0	1
	1	1	1	0	0
	0	0	0	1	1

What if, instead of merely setting all unobserved pairs to zero or ignoring them completely, we assigned a weight for those interaction pairs that are missing. We can think of this as not giving it an absolutely certain negative score, but instead as a low confidence score. This is called weighted alternating least squares, or WALS for short. Notice that in the modified equation, we have the ALS term on the left and a new term on the right that weights the unobserved entries. This usually provides much better recommendations and is the algorithm we will now focus on.

$$\sum_{(i,j) \in \text{obs}} (A_{ij} - U_i V_j)^2 + w_0 * \sum_{(i,j) \notin \text{obs}} (0 - U_i V_j)^2$$

Quiz

There are many ways to handle unobserved user-interaction matrix pairs. ___ explicitly sets all missing values to zero. ___ simply ignores missing values. ___ uses weights instead of zeros that can be thought of as representing _____.

- A. WALS, SVD, ALS, low confidence
- B. SVD, ALS, WALS, low confidence
- C. SVD, ALS, WALS, high confidence
- D. SVD, ALS, WSVD, low confidence

Now it's time to test your knowledge about the ways unobserved pairs are handled by different algorithms. There are many ways to handle unobserved user-interaction matrix pairs. BLANK explicitly sets all missing values to zero. BLANK simply ignores missing values. BLANK uses weights instead of zeros that can be thought of as representing BLANK. Choose the answers that best fill in the blanks.

Quiz

There are many ways to handle unobserved user-interaction matrix pairs. ___ explicitly sets all missing values to zero. ___ simply ignores missing values. ___ uses weights instead of zeros that can be thought of as representing ____.

- A. WALS, SVD, ALS, low confidence
- B. SVD, ALS, WALS, low confidence**
- C. SVD, ALS, WALS, high confidence
- D. SVD, ALS, WSVD, low confidence

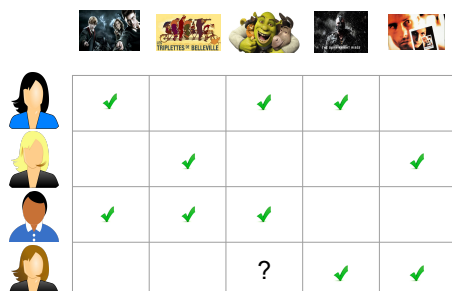
The correct answer is B!

In singular value decomposition, or SVD, we explicitly set all missing values to zero. This can be problematic because we are making a large assumption and imputing data that could have meaning, when in fact there was no data there, which can throw off our predictions.

The matrix factorization for collaborative filtering algorithm, alternating least squares, or ALS, simply ignores missing values.

Weighted alternating least squares, or WALS, uses weights instead of zeros, which can be thought of as representing low confidence. WALS provides some of the best recommendation performance compared to these other methods and is what we will focus on using for collaborative filtering.

The WALS Estimator in TensorFlow does not need any labels; it just needs the ratings matrix organized into rows and columns

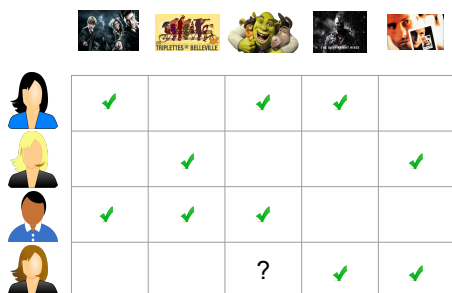


	Star Wars	Toy Story	Yoda	The Godfather	The Godfather Part II
User 1	✓		✓	✓	
User 2		✓			✓
User 3	✓	✓	✓		
User 4			?	✓	✓

Sparse and
large

So how do we create a TensorFlow model to use WALS? Well fortunately, there is a WALS Estimator! Using this, we simply just need to structure our inputs correctly to be fed into the estimator, and it takes care of most of the rest.

The WALS Estimator in TensorFlow does not need any labels; it just needs the ratings matrix organized into rows and columns



	✓		✓	✓	
		✓			✓
	✓	✓	✓		
			?	✓	✓

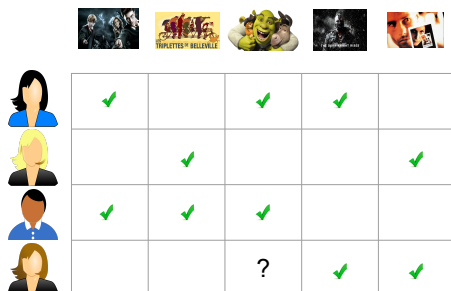
Sparse and
large





```
def training_input_fn():
    features = {
        INPUT_ROWS: tf.SparseTensor(...),
        INPUT_COLS: tf.SparseTensor(...)
    }

    return features, None
```

This begins with the training input function, where we simply define our features.

The WALS Estimator in TensorFlow does not need any labels; it just needs the ratings matrix organized into rows and columns



	✓		✓	✓	
		✓			✓
	✓	✓	✓		
			?	✓	✓

Sparse and
large

```
def training_input_fn():
    features = {
        INPUT_ROWS: tf.SparseTensor(...),
        INPUT_COLS: tf.SparseTensor(...)
    }

    return features, None
```

Shouldn't feeding the rows be
enough? Why also columns?

But wait, aren't there usually labels? In this case we are alternating some of our features as labels. That is where the alternating comes from, in alternating least squares. We fix the rows and solve for the columns, then we fix the columns and solve for the rows. So the rows might begin as our typical "features," where we are predicting the columns, and comparing with the actual column values as our "labels." from the ratings matrix, and vice versa when alternated.

The W in WALS is for Weighted

Proprietary + Confidential

```
__init__(
    num_rows,
    num_cols,
    embedding_dimension,
    unobserved_weight=0.1,
    regularization_coeff=None,
    row_init='random',
    col_init='random',
    row_weights,
    col_weights,
    ...)
```

Google Cloud

Also, remember the W in WALS is for weighted. That means we can customize our problem; it's not just for managing missing observations. You can add weights for specific entries if you want. One reason might be to encode our profit margin on items and use that as a weight. This way, more profitable items will be recommended more.

We're ignoring this here, but if you wanted to, it's just another key in the features dictionary that you could add. Note here that INPUT_ROWS and INPUT_COLS are both from a batch, not all of the rows. We'll dive deeper into this later, but now let's take a look at the ALS algorithm.

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

Google Cloud

Here is the pseudocode for the alternating least squares algorithm. Remember, with matrix factorization, we are iteratively learning U and V, which multiplied together hopefully reconstructs a close approximation to our original user-item interaction matrix, however it won't be perfect. However, with alternating least squares we can get very close. Let's walk through it to build a deeper intuition of what is happening under the hood.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State  $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 

```

```

}r_{ij}v_j$
\EndFor
\State \textbf{end for}
\For {$j=1$ to $m$}
\State $v_j = (\sum_{r_{ij}\in r_{*j}}\{ }u_{iu_i}^T+\lambda I_k)^{-1}\sum_{r_{ij}\in r_{*j}}\{
}r_{ij}u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

Google Cloud

First we initialize the U and V factor matrices. These are our learned row and column factors, which based on our previous examples, could be users for U and movies for V. Just as how we would normally treat an embedding, these start off typically as random normal noise. Our goal is to calculate these two embeddings simultaneously.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State $\mathbf{u}_i = (\sum_{r_{ij} \in r_{i*}} \mathbf{v}_j \mathbf{v}_j^T + \lambda \mathbf{I}_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} \mathbf{v}_j$
\EndFor
\EndRepeat

```

```

\State \textbf{end for}
\For {$j=1$ to $m$}
\State $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} \{
\} r_{ij} u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

The loop makes it possible to batch the rows or columns

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

Google Cloud

Next, we are going to enter a loop that will run until we reach the loss converges usually within some tolerance. This is our alternation loop. Let's enter and see just exactly what is alternating back and forth.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State $\$u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$
\EndFor
\State \textbf{end for}

```



```

\For {$j=1$ to $m$}
\State $v_j = (\sum_{r_{ij} \in r_{*j}} u_{iu_i}^T + \lambda l_k)^{-1} \sum_{r_{ij} \in r_{*j}} \{
\} r_{ij} u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

At this stage, we need column factors and the ratings entries for this row

In the first phase of the alternation, we are solving for the row factors U by looping through all of the rows, which in our example would be the users. Does the equation look familiar? Well of course, it is the ordinary least squares normal equation, with L2 regularization added, with regularization constant lambda. We're used to seeing this with Xs and Ys, but it is essentially the same. This means that the u_i 's we are solving for are analogous to the coefficients or weight vector learned from linear regression.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State $\mathbf{u}_i = (\sum_{r_{ij} \in r_{i*}} \mathbf{v}_j \mathbf{v}_j^T + \lambda \mathbf{I}_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} \mathbf{v}_j$

```

```

}r_{ij}v_j$
\EndFor
\State \textbf{end for}
\For {$j=1$ to $m$}
\State $v_j = (\sum_{r_{ij}\in r_{*j}}\{ }u_{iu_i}^T+\lambda I_k)^{-1}\sum_{r_{ij}\in r_{*j}}\{
}r_{ij}u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

At this stage, we need column factors and the ratings entries for this row

The normal equation's Gram matrix is using V, our column factors, instead of X.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State $\mathbf{u}_i = (\sum_{r_{ij} \in r_{i*}} \mathbf{v}_j \mathbf{v}_j^T + \lambda \mathbf{I}_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} \mathbf{v}_j$
\EndFor
\State \textbf{end for}
\For {$j=1$ to $m$}
\State $\mathbf{v}_j = (\sum_{r_{ij} \in r_{*j}} \mathbf{u}_i \mathbf{u}_i^T + \lambda \mathbf{I}_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} \mathbf{u}_i$
\EndFor
\EndRepeat
\end{algorithmic}
\end{document}

```

```
}r_{ij}u_i$  
\EndFor  
\State \textbf{end for}  
\Until convergence  
\end{algorithmic}  
\end{algorithm}  
\end{document}
```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence
  
```

At this stage, we need column factors and the ratings entries for this row

Adding in the L2 regularization, we then take the inverse as usual, which gives us the inverse regularized Gram matrix.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}
  
```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother
  
```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State $\$u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$
\EndFor
\State \textbf{end for}
\For {$j=1$ to $m$}
  
```

```

\State $v_j = (\sum_{r_{ij} \in r_{*j}} u_{iu_i}^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} \{
\} r_{ij} u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

At this stage, we need column factors and the ratings entries for this row

Last but not least, we multiply by the moment matrix to find our weight vectors. As before, our usual X has been replaced with the column factors V , and instead of our labels Y , we have the i th row of the ratings matrix, R , as our labels.

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For  $\{i=1 \text{ to } n\}$ 
\State  $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
\EndFor
\State \textbf{end for}

```



```

\For {$j=1$ to $m$}
\State $v_j = (\sum_{r_{ij} \in r_{*j}} u_{iu_i}^T + \lambda l_k)^{-1} \sum_{r_{ij} \in r_{*j}} \{
\} r_{ij} u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

The ALS algorithm works by alternating between rows and columns to factorize the matrix

Algorithm 1 ALS for Matrix Completion

```

1: Initialize U, V
2: repeat
3:   for  $i = 1$  to  $n$  do
4:      $u_i = (\sum_{r_{ij} \in r_{i*}} v_j v_j^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} v_j$ 
5:   end for
6:   for  $j = 1$  to  $m$  do
7:      $v_j = (\sum_{r_{ij} \in r_{*j}} u_i u_i^T + \lambda I_k)^{-1} \sum_{r_{ij} \in r_{*j}} r_{ij} u_i$ 
8:   end for
9: until convergence

```

At this stage, we need row factors and the ratings entries for this column

Now that we have solved for all of the row factors, using the column factors as our "features" and the rating matrix rows as our "labels," we are going to alternate gears and do the same, but now we will iterate over all of the columns and solve for the column factors, using the row factors that we just solved for as our "features" and the rating matrix columns as our "labels."

```

\documentclass{article}
\usepackage{amsmath}
\usepackage{algorithm}
\usepackage[noend]{algpseudocode}

```

```

\makeatletter
\def\BState{\State\hskip-\ALG@thistlm}
\makeatother

```

```

\begin{document}
\begin{algorithm}
\caption{ALS for Matrix Completion}\label{euclid}
\begin{algorithmic}[1]
\State Initialize U, V
\Repeat
\For {$i=1$ to $n$}
\State $\mathbf{u}_i = (\sum_{r_{ij} \in r_{i*}} \mathbf{v}_j \mathbf{v}_j^T + \lambda \mathbf{I}_k)^{-1} \sum_{r_{ij} \in r_{i*}} r_{ij} \mathbf{v}_j$
\EndFor
\EndRepeat
\EndAlgorithmic
\end{algorithm}

```

```

\EndFor
\State \textbf{end for}
\For {$j=1$ to $m$}
\State $v_j = (\sum_{r_{ij} \in r_{*j}} u_{iu_i}^T + \lambda l_k)^{-1} \sum_{r_{ij} \in r_{*j}} \{
\} r_{ij} u_i$
\EndFor
\State \textbf{end for}
\Until convergence
\end{algorithmic}
\end{algorithm}
\end{document}

```

Quiz

ALS is an alternating least squares algorithm. In ordinary least squares we have the analytic solution of the normal equation:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

In ALS, during the row-factor solving phase, where we fix column-factors:

Row-factors U are analogous to __,

Column-factors V are analogous to __,

Ratings matrix R is analogous to __.

- A. X, y, β
- B. y, X, β
- C. β, X, y
- D. β, y, X

Google Cloud

ALS is an alternating least squares algorithm. In ordinary least squares, we have the analytic solution of the normal equation: Beta equals the inverse of X transpose X, plus lambda times the identity matrix, all multiplied by X transpose y.

In ALS, during the row-factor solving phase, where we fix column-factors: Row-factors U are analogous to BLANK, column-factors V are analogous to BLANK, and the ratings matrix R is analogous to BLANK. Choose the answer that best fills in the blanks.

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

\linebreak

X,y,\beta\linebreak

y,X,\beta\linebreak

\beta,X,y\linebreak

\beta,y,X\linebreak

Quiz

ALS is an alternating least squares algorithm.
In ordinary least squares we have the analytic solution of the normal equation:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

In ALS, during the row-factor solving phase, where we fix column-factors:

Row-factors U are analogous to __,

Column-factors V are analogous to __,

Ratings matrix R is analogous to __.

A. X, y, β

B. y, X, β

C. β, X, y

D. β, y, X

The correct answer is C! When solving for the row-factors U, we fix the column-factors V.

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

\\linebreak

X,y,\\beta\\linebreak

y,X,\\beta\\linebreak

\\beta,X,y\\linebreak

\\beta,y,X\\linebreak

Quiz

ALS is an alternating least squares algorithm.
In ordinary least squares we have the analytic solution of the normal equation:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

In ALS, during the row-factor solving phase, where we fix column-factors:

Row-factors U are analogous to __,

Column-factors V are analogous to __,

Ratings matrix R is analogous to __.

A. X, y, β

B. y, X, β

C. β, X, y

D. β, y, X

We are solving for the latent variable embedding weights which is analogous to what beta represents in linear regression, the coefficients or weights of predictor variables.

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

\linebreak

X,y,\beta\linebreak

y,X,\beta\linebreak

\beta,X,y\linebreak

\beta,y,X\linebreak

Quiz

ALS is an alternating least squares algorithm.
In ordinary least squares we have the analytic solution of the normal equation:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

In ALS, during the row-factor solving phase, where we fix column-factors:

Row-factors U are analogous to __,

Column-factors V are analogous to __,

Ratings matrix R is analogous to __.

A. X, y, β

B. y, X, β

C. β, X, y

D. β, y, X

In linear regression, our predictors are often denoted by the matrix X, which in this case are our column-factors V, which we are holding fixed for this half-cycle.

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

\\linebreak

X,y,\\beta\\linebreak

y,X,\\beta\\linebreak

\\beta,X,y\\linebreak

\\beta,y,X\\linebreak

Quiz

ALS is an alternating least squares algorithm.
In ordinary least squares we have the analytic solution of the normal equation:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

In ALS, during the row-factor solving phase, where we fix column-factors:

Row-factors U are analogous to __,

Column-factors V are analogous to __,

Ratings matrix R is analogous to __.

A. X, y, β

B. y, X, β

C. β, X, y

D. β, y, X

Lastly, whether solving for row-factors or column-factors, our ratings matrix R is analogous to the labels of linear regression usually denoted y. Now, since we are solving for vectors, we are not using the entire ratings matrix R. We are only using the ith row of the ratings matrix, when solving for the ith row-factor.

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

\linebreak

X,y,\beta\linebreak

y,X,\beta\linebreak

\beta,X,y\linebreak

\beta,y,X\linebreak

Quiz

ALS is an alternating least squares algorithm.
In ordinary least squares we have the analytic solution of the normal equation:

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

In ALS, during the row-factor solving phase, where we fix column-factors:

Row-factors U are analogous to __,

Column-factors V are analogous to __,

Ratings matrix R is analogous to __.

A. X, y, β

B. y, X, β

☒ C. $\beta, X, y \implies X, \beta, y$

D. β, y, X

Remember, when solving for the column-factors in the next half-cycle of the ALS algorithm, the variable analogies flip, with V now being solved for like Beta, U is now fixed like the predictors X, and y is still the ratings matrix, but now the jth column of it is being used to solve for the jth column-factor.

All of this happening with k latent factors that represent the user and item embeddings into k dimensional spaces for both.

$$\beta = (X^T X + \lambda I_k)^{-1} X^T y$$

\linebreak

X,y,\beta\linebreak

y,X,\beta\linebreak

\beta,X,y\linebreak

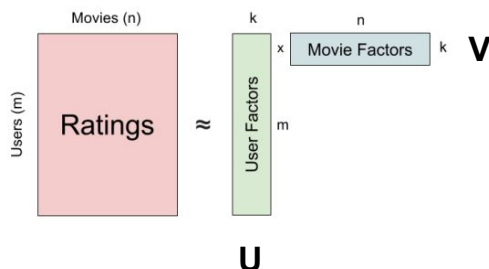
\beta,y,X\linebreak

Getting data to the ALS algorithm

Feed the ALS algorithm whole rows (or columns) at a time, but because knowing which stage it's in is difficult, feed both!

Normally, during each half of the alternating loop, we'd feed the ALS algorithm whole rows or columns at a time, but since it's hard to know which stage it's in, we'll just feed both instead, so that it always has the right data, regardless of what stage it is in. Remember, this will be fed in batches of rows and columns from the ratings matrix.

Hopefully, we will go through the matrix multiple times so things will even out, and you'll end up processing all the rows and columns. It might be important to make sure that `batch_size` doesn't cleanly divide `dataset_length` so that the rollover offset causes different groupings in the batch, so that the same batches don't continually repeat themselves.



Iterative algorithm:

- Fix V , compute U
- Fix U , compute V

Graphically, going off our user-movie example, we have our user by movie matrix R that we are hoping to factorize into our user factors matrix U and our movie factors matrix V , made up of k latent factors. We iterate alternately until convergence, fixing V , computing U , and then fixing U , computing V , and so on.

```
def training_input_fn():  
    features = {  
        INPUT_ROWS: tf.SparseTensor(...)  
        INPUT_COLS: tf.SparseTensor(...)  
    }  
  
    return features, None
```

Coming back to the training input function, remember: we read in batches of rows and columns at the same time, and they are stored in a sparse tensor, which we will discuss the details of in the next section. Remember, we don't need labels because, due to the alternation, the labels come from the feature we are not currently solving for.

WALS requires the ratings matrix to be really matrix entries, so you will have to map id-strings to be 0,1,2,...


Ok, so we need rows and columns of a ratings matrix, and then we can continue? Well, most data warehouses for systems of millions of users, with millions of items, don't store the complete cartesian product. That would be a huge waste of space because that matrix is extremely sparse. Instead, ratings data is usually stored when there is an interaction that becomes a record or row. There may be an interaction timestamp, and most definitely there will be a column that contains an identifier of the user of that interaction, another column of the item that user interacted with, and then a column for the actual interaction data, whether that is a number of stars, like or dislike, or the duration of interaction. However, these are usually represented as tables and not as actually matrices that have row indices and column indices. Let's look at an example.

WALS requires the ratings matrix to be really matrix entries, so you will have to map id-strings to be 0,1,2,...

	visitorid	contentid	session_duration
0	7337153711992174438	100074831	44652
1	5190801220865459604	100170790	1214205
2	5874973374932455844	100510126	32109
3	2293633612703952721	100510126	47744
4	1173698801255170595	100676857	10512

In this example, visitorid contains the unique user identifier, and contentid contains the unique item id, which are the video ids. The actual interaction data in this example is session duration, which is probably going to be used as implicit feedback, where we infer that, if a user has a longer session, they like that content better than content with shorter sessions.

Notice anything unusual about this table? Now unless these are the row indices of the users of the entire galaxy, those are some really large numbers for visitorid, many magnitudes more than all of the people on Earth. Same goes for contentid, even though 100 million is much more believable than the 73 quintillion visitors. Remember, these two columns need to map to a contiguous matrix, so these should be the indices of those rows and columns. Also, it would help if we scale session duration to be a small number.



	userId	itemId	rating
0	0	0	0.231208
1	1	1	1.000000
2	2	2	0.166260
3	3	2	0.247218
4	4	3	0.054431

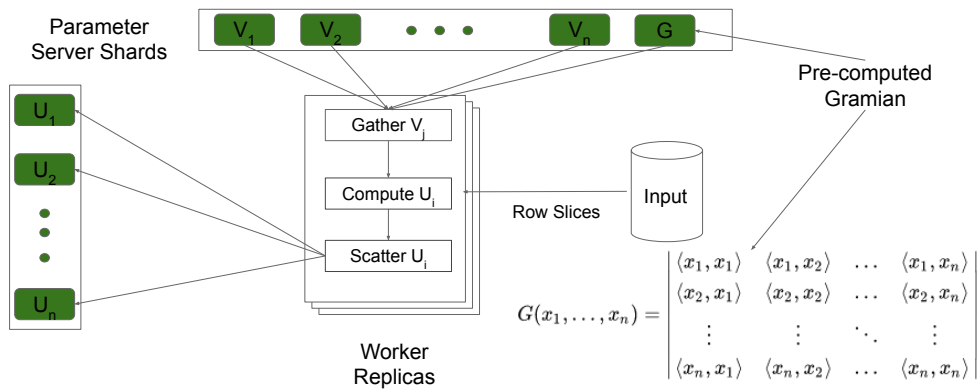
Save the mapping to
persistent storage because
you'll need to map input values
to the mapped values!

What we need to create is a mapping! We'll map visitor ID to user ID, content ID to item ID, and session_duration to rating. This mapping needs to be saved to persistent storage, because you'll need to map input values to the mapped values, not just during training, but also during inference! This way you can quickly take any visitor ID, content ID, and session_duration and get the corresponding mapped values used and output by the model.

Speaking of inference, when making predictions, you need access at time of prediction not just to these mappings, but to the entire input dataset, because you may want to filter out any previously interacted-with items, such as a previous purchase, view, or rating, to provide the top K recommendations of new items.

Should you recommend an already rated item to a user? For some problems, users don't want to be recommended things that they already bought or saw, like a movie; however, for a restaurant they liked, they may want to return.

Even though the update depends on the full column factor, it is possible to distribute WALS by precomputing a the Gramian G



Also, we can distribute our data instead of sending whole rows and columns in our input function to one worker. Each worker's mini-batch consists of a subset of rows of the matrix. The training step computes the new values for the corresponding row factors. However, the update depends on the full column factor, which would be costly to fetch each step. So we use a trick where we precompute a Gramian, G , which is just the determinant of the matrix inner product, $X^T X$. Given G , the worker now only needs to look at a subset of rows of V (those corresponding to non-zero entries in the input) to compute the update. Now, we can use Gather and Scatter to perform fetches and updates and use custom C++ kernels for the compute. This is much easier to distribute and scale.

Quiz

When using the WALS Estimator, it is important to have the inputs in the correct format. What should we do with the table below in the input function before it is used by the Estimator?

	clientID	productID	sentiment
0	s98h2oknsa8	87198471098	Like
1	xnl891nbjk21	95218970198	Love
2	kn0iokjs0i2n	12719850987	Neutral
3	mna19n2oino	89713598759	Dislike

- A. Map clientID to int in [0, num_clients)
- B. Map productID to int in [0, num_products)
- C. Map sentiment from string to numeric
- D. A & B
- E. A & C
- F. All of the above

When using the WALS Estimator, it is important to have the inputs in the correct format. What should we do with the table here, in the input function, before it is used by the Estimator? Do we want to map clientID to integers in the range 0 inclusive to the number of clients exclusive? Do we want to map productID to integers in the range 0 inclusive to the number of products exclusive? Do we want to map sentiment from a string representation to numeric representation? Or maybe some combination?

Quiz

When using the WALS Estimator, it is important to have the inputs in the correct format. What should we do with the table below in the input function before it is used by the Estimator?

	clientID	productID	sentiment
0	s98h2oknsa8	87198471098	Like
1	xnl891nbjk21	95218970198	Love
2	kn0iokjs0i2n	12719850987	Neutral
3	mna19n2oino	89713598759	Dislike

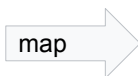
- A. Map clientID to int in [0, num_clients)
- B. Map productID to int in [0, num_products)
- C. Map sentiment from string to numeric
- D. A & B
- E. A & C
- F. All of the above

The correct answer is F! The clientID is represented as an alphanumeric string in this data, so we need to map each string to an integer representing that client's user index. The productID at least isn't a string, but it is a long integer that is not representative of the actual index, so we need to map each to an integer representing that product's item index. As for the rating, we probably have an example of explicit feedback; however, it is a string, so we will need an ordinal mapping, perhaps from lowest sentiment to highest sentiment by integers, and we can even scale that to be between 0 or 1 or some other range. The main goal is to eventually get the ratings into a numeric format. Therefore, we have to create all three mappings and save them in persistent storage, to be used for future training and inference.

Data from table to matrix

Proprietary + Confidential

	visitorId	contentId	session_duration
0	7337153711992174438	100074831	44652
1	5190801220865459604	100170790	1214205
2	5874973374932455844	100510126	32109
3	2293633612703952721	100510126	47744
4	1173698801255170595	100676857	10512



	userId	itemId	rating
0	0	0	0.231208
1	1	1	1.000000
2	2	2	0.166260
3	3	2	0.247218
4	4	3	0.054431

Google Cloud

Now that we've learned how to get the data into the right form from a table in our data warehouse to our user-item interaction matrix through mapping business-centric values to row and column indices, do we need to implement the ALS algorithm from scratch to be able to use it and learn the user and item embeddings?

TensorFlow Estimator

Proprietary + Confidential



Google Cloud

Thankfully for us, TensorFlow has the WALSMatrixFactorization estimator to take care of most of the work, to create our recommender system. As with all canned estimators, we just need to connect some of the piping, such as input functions, serving input functions, and the train and evaluate loop, and the estimator will take care of everything else.

Because WALS requires whole rows or columns, the data has to be preprocessed to provide SparseTensors of rows/columns

Because the WALSMatrixFactorization estimator requires whole rows and columns, we have to preprocess the data from our data warehouse to be of the right form, namely into a structure that we can store into each SparseTensor for rows and columns.

Because WALS requires whole rows or columns, the data has to be preprocessed to provide SparseTensors of rows/columns

```
import tensorflow as tf
grouped_by_items = mapped_df.groupby('itemId')
with tf.python_io.TFRecordWriter('data/users_for_item') as ofp:
    for item, grouped in grouped_by_items:
        example = tf.train.Example(features=tf.train.Features(feature={
            'key': tf.train.Feature(int64_list=tf.train.Int64List(value=[item])),
            'indices': tf.train.Feature(int64_list=tf.train.Int64List(value=grouped['userId'].values)),
            'values': tf.train.Feature(float_list=tf.train.FloatList(value=grouped['rating'].values))
        }))
        ofp.write(example.SerializeToString())
```

tf.sparse_merge

```
sparse_merge(
    sp_ids,
    sp_values,
    vocab_size,
    name=None,
    already_sorted=False
)
```

Here is an example of our preprocessing step for the columns, since we are grouping by itemId. You will have similar code for rows, where you group by userId instead.

Because WALS requires whole rows or columns, the data has to be preprocessed to provide SparseTensors of rows/columns

```
import tensorflow as tf
grouped_by_items = mapped_df.groupby('itemId')
with tf.python_io.TFRecordWriter('data/users_for_item') as ofp:
    for item, grouped in grouped_by_items:
        example = tf.train.Example(features=tf.train.Features(feature={
            'key': tf.train.Feature(int64_list=tf.train.Int64List(value=[item])),
            'indices': tf.train.Feature(int64_list=tf.train.Int64List(value=grouped['userId'].values)),
            'values': tf.train.Feature(float_list=tf.train.FloatList(value=grouped['rating'].values))
        }))
        ofp.write(example.SerializeToString())
```

TensorFlow records are more efficient and support hierarchical structures

tf.sparse_merge

```
sparse_merge(
    sp_ids,
    sp_values,
    vocab_size,
    name=None,
    already_sorted=False
)
```

So we want SparseTensors, a hierarchical data structure. What file type should we write our preprocessed data to? Here we are storing two arrays. It's kind of painful to do this in CSV, and inefficient to do it in JSON. Therefore, our best option will be to use TensorFlow records. And look, it's fairly easy to do in Python!

Here, we create a TF Record Writer with our specified path “users for item” in this example.

Because WALS requires whole rows or columns, the data has to be preprocessed to provide SparseTensors of rows/columns

```
import tensorflow as tf
grouped_by_items = mapped_df.groupby('itemId')
with tf.python_io.TFRecordWriter('data/users_for_item') as ofp:
    for item, grouped in grouped_by_items:
        example = tf.train.Example(features=tf.train.Features(feature={
            'key': tf.train.Feature(int64_list=tf.train.Int64List(value=[item])),
            'indices': tf.train.Feature(int64_list=tf.train.Int64List(value=grouped['userId'].values)),
            'values': tf.train.Feature(float_list=tf.train.FloatList(value=grouped['rating'].values))
        }))
        ofp.write(example.SerializeToString())
```

tf.sparse_merge

```
sparse_merge(
    sp_ids,
    sp_values,
    vocab_size,
    name=None,
    already_sorted=False
)
```

For each item, we want to create an example that stores our features, which we can then write out as serialized examples to our output file.

Because WALS requires whole rows or columns, the data has to be preprocessed to provide SparseTensors of rows/columns

```
import tensorflow as tf
grouped_by_items = mapped_df.groupby('itemId')
with tf.python_io.TFRecordWriter('data/users_for_item') as ofp:
    for item, grouped in grouped_by_items:
        example = tf.train.Example(features=tf.train.Features(feature={
            'item': tf.train.Feature(int64_list=tf.train.Int64List(value=[item])),
            'indices': tf.train.Feature(int64_list=tf.train.Int64List(value=grouped['userId'].values)),
            'values': tf.train.Feature(float_list=tf.train.FloatList(value=grouped['rating'].values))
        }))
        ofp.write(example.SerializeToString())
```

Save indices, values -- they can
be quickly converted into a
SparseTensor using
`tf.sparse_merge`

`tf.sparse_merge`

```
sparse_merge(
    sp_ids,
    sp_values,
    vocab_size,
    name=None,
    already_sorted=False
)
```

SparseTensors are hierarchical data structures where indices and values are stored, which in this case will be the userIds, or the index of the users from the user-item interaction matrix and the ratings, respectively. We will also store the key, which is the item index, from the user-item interaction matrix. This value is important to save, which we'll see later, because during batching the first dimension of the SparseTensor indices tensor, with rank 2, becomes the batch indices instead, so we'll need the key data to replace that incorrect overwrite when we're done with the batching phase of the input function.

The indices and values can be quickly converted into a SparseTensor using `tf.sparse_merge`, where you need the ids, values, and the vocab_size, such as the number of items in this example.

Quiz

If we want to recommend items for a user, when we are writing out to the TF Record file:

Our **key** train feature should be _____.
Our **indices** train feature should be _____.
Our **values** train feature should be _____.

- A. item, userId, rating
- B. item, itemId, rating
- C. userID, user, rating
- D. rating, user, userId
- E. user, itemId, rating
- F. rating, user, item

Now that we've learned how we should preprocess our data for WALS, let's test your knowledge! If we want to recommend items for a user, when we are writing out to the TF Record file: Our key train feature should be BLANK. Our indices train feature should be BLANK. Our values train feature should be BLANK. Choose the answer that best fills in the blanks.

Quiz

If we want to recommend items for a user, when we are writing out to the TF Record file:

Our **key** train feature should be _____.
Our **indices** train feature should be _____.
Our **values** train feature should be _____.

- A. item, userId, rating
- B. item, itemId, rating
- C. userID, user, rating
- D. rating, user, userId
- E. user, itemId, rating
- F. rating, user, item

The correct answer is E! Because we are recommending **items**, note the plural, for a **user**, note the singular; we are in a situation where each example will be a unique user per row. There will be a variable number of items per each user which will be the columns, based on what that user interacted with; some users interacting with only a few items and some users interacting with many items. The indices tensor will be the item indices, or itemIds, they have interacted with, and the values tensor will be the corresponding ratings at those user-item interaction points within the user-item interaction matrix. Answer A, however, should look familiar because that is the example we just went through, which is for recommending users for an item.

Data preprocessed into TF Records

Proprietary + Confidential

```
'key': tf.train.Feature(int64_list=
tf.train.Int64List(value=[user])),

'indices': tf.train.Feature(int64_list=
tf.train.Int64List(value=grouped['itemId'].values)),

'values': tf.train.Feature(float_list=
tf.train.FloatList(value=grouped['rating'].values
```

Google Cloud

So now that we have preprocessed our data and saved that out to TF Records, we have the data in the correct format to be read into the needed SparseTensor format.

Data preprocessed into TF Records

Proprietary + Confidential

```
'key': tf.train.Feature(int64_list=
tf.train.Int64List(value=[user])),

'indices': tf.train.Feature(int64_list=
tf.train.Int64List(value=grouped['itemId'].values)),

'values': tf.train.Feature(float_list=
tf.train.FloatList(value=grouped['rating'].values
```

Google Cloud

If we want to be able to eventually be able to predict items for users we will need a key which will be the user index.

Data preprocessed into TF Records

Proprietary + Confidential

```
'key': tf.train.Feature(int64_list=
tf.train.Int64List(value=[user])),

'indices': tf.train.Feature(int64_list=
tf.train.Int64List(value=grouped['itemId'].values)),

'values': tf.train.Feature(float_list=
tf.train.FloatList(value=grouped['rating'].values
```

Google Cloud

We'll also need the items that each user interacted with which will be our indices.

Data preprocessed into TF Records

Proprietary + Confidential

```
'key': tf.train.Feature(int64_list=
tf.train.Int64List(value=[user])),

'indices': tf.train.Feature(int64_list=
tf.train.Int64List(value=grouped['itemId'].values)),

'values': tf.train.Feature(float_list=
tf.train.FloatList(value=grouped['rating'].values
```

Google Cloud

And we'll also need the ratings of these interactions which we'll set as our values. But just how does this get into our estimator? What even are the pieces needed to get the WALS Matrix Factorization estimator to work? Let's look!

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

Here's the code to run the WALS Matrix Factorization estimator.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

At the moment, WALS is a **contrib**.Estimator so I'm using Experiment and will be using learn_runner.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['num_users'], num_cols=args['num_items'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

We need to obviously tell WALS how big our matrix is, how many users and how many items in our interaction matrix.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embedding'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

We need to give the number of dimensions or latent factors we want to compress our interaction matrix down into for our embeddings.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

Of course we also need to tell the estimator where to write out the model files.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

We'll need a train input function that takes our preprocessed TF Records

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

and likewise an eval input function for those TF Records.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

Also, we'll need the number of steps, or batches, we are going to train for.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

And 1 evaluation step.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_to_eval,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

We'll want to set the minimum evaluation frequency so that we don't get our training bogged down by evaluating too often.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategy=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args))
```

Google Cloud

Lastly, we'll set an export strategy with our serving input function for inference serving.

WALS Matrix Factorization Estimator

Proprietary + Confidential

```
tf.contrib.learn.Experiment(  
    tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir']),  
    train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),  
    eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),  
    train_steps=train_steps,  
    eval_steps=1,  
    min_eval_frequency=steps_in_epoch,  
  
    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(  
        serving_input_fn=create_serving_input_fn(args)))
```

Google Cloud

Let's first take a deeper dive into our input functions so that we can see how to use our newly created preprocessed TF Records.

The input function has to read the files and create `SparseTensors` for the rows and for the columns.

So now that we have preprocessed our data and saved that out to TF Records, our input function is going to need to read them. Therefore, let's now define a function that will parse our TF records.

The input function has to read the files and create sparse tensors for the rows and for the columns


```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user', filename, vocab_size,
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

First, our function `parse_tfrecords` receives the `filename` and the `vocab_size`, which is required as we saw in the `sparse_merge` signature earlier. For `items_for_user`, the vocabulary is the items, so `vocab_size` equals `nitems`, and for `users_for_item`, the vocabulary is the users, so `vocab_size` equals `nusers`.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):  
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))  
    dataset = tf.data.TFRecordDataset(files)  
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))  
    dataset = dataset.repeat(num_epochs)  
    dataset = dataset.batch(args['batch_size'])  
    dataset = dataset.map(lambda x: remap_keys(x))  
    return dataset.make_one_shot_iterator().get_next()  
  
def _input_fn():  
    features = {  
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',  
args['nitems']),  
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])  
    }  
    return features, None
```



Now that we are in the function, we first create a list of all of the files at the input path with the passed filename signature, including wildcards, using `tf.gfile.Glob`.

The input function has to read the files and create sparse tensors for the rows and for the columns

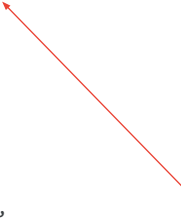
```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

When we have our file list, we will create a TF Record Dataset.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):  
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))  
    dataset = tf.data.TFRecordDataset(files)  
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))  
    dataset = dataset.repeat(num_epochs)  
    dataset = dataset.batch(args['batch_size'])  
    dataset = dataset.map(lambda x: remap_keys(x))  
    return dataset.make_one_shot_iterator().get_next()  
  
def _input_fn():  
    features = {  
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',  
args['nitems']),  
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])  
    }  
    return features, None
```



Next, we take our TF Record Dataset and apply a map to it, where we will decode each serialized example using a custom function we made and the corresponding vocab_size. In this case the vocab size for input rows is nitems and input columns is nusers.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

Of course, we don't want to only go through the files one time; therefore we will apply a repeat on our dataset, a num_epochs number of times.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

We also know that batching is important in machine learning, so instead of using only one example at a time, we want to use a `batch_size` number of examples.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

After batching is complete, we need to remap the keys that we talked about earlier in order to fix the first dimension of the rank 2 indices tensor of our SparseTensor, because batching overwrote it, with the index within the batch.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

Lastly, now that our SparseTensors are fixed, we will return the next batch of SparseTensors from our dataset using a one-shot iterator.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        'WALSMatrixFactorization:INPUT_ROWS: parse_tfrecords: items for user': filename, vocab_size
        args['items']],
        'WALSMatrixFactorization:INPUT_COLS: parse_tfrecords: users for item': args['users']]
    }
    return features, None
```

This is all wrapped by the input function, which will be called by the WALSMatrixFactorization estimator. We call `parse_tfrecords` for our rows, which will be items for user, and for our columns, which will be users per item. This is saved in a features dictionary. We could have other features in the dictionary, such as priority weights, etc.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

We return features only from the input function, because our labels are within our features as we alternate back and forth between solving rows and columns while keeping the other fixed.

Quiz

The WALS MF estimator takes the INPUT_ROWS and INPUT_COLS as features. If we have our items_for_user and users_for_item TF Records, which filename and vocab_size should we use for each of these features, respectively?

- A. items_for_user, nusers, users_for_item, nitems
- B. users_for_item, nusers, items_for_user, nitems
- C. items_for_user, nusers, users_for_item, nitems
- D. users_for_item, nitems, items_for_user, nusers

Now that we've written our input function and TF records helper function, let's test your knowledge. The WALS Matrix Factorization estimator takes the input rows and input columns as features. If we have our items for user and users for item TF Records, which filename and vocab_size should we use for each of these features, respectively?

Quiz

The WALS MF estimator takes the INPUT_ROWS and INPUT_COLS as features. If we have our items_for_user and users_for_item TF Records, which filename and vocab_size should we use for each of these features, respectively?

- A. items_for_user, nusers, users_for_item, nitems
- B. users_for_item, nusers, items_for_user, nitems
- C. items_for_user, nitems, users_for_item, nusers
- D. users_for_item, nitems, items_for_user, nusers

The correct answer is C! Remember, our user-item interaction matrix has shape number of users by number of items. Therefore, there is a row for each unique user and a column for each unique item.

Quiz

The WALS MF estimator takes the **INPUT_ROWS** and INPUT_COLS as features. If we have our items_for_user and users_for_item TF Records, which filename and vocab_size should we use for each of these features, respectively?

- A. items_for_user, nusers, users_for_item, nitems
- B. users_for_item, nusers, items_for_user, nitems
- C. items_for_user, nitems, users_for_item, nusers
- D. users_for_item, nitems, items_for_user, nusers

Thus, when looking at the input rows features it will be a batch of rows from our dataset. This means we should be using the items_for_user TF Record since there will be multiple items for each unique user. The vocab_size, which is used for the indices of our SparseTensor for the input rows feature will be nitems, because each unique user will have interacted with a subset of items from the entire item inventory, or vocabulary.

Quiz

The WALS MF estimator takes the INPUT_ROWS and INPUT_COLS as features. If we have our items_for_user and users_for_item TF Records, which filename and vocab_size should we use for each of these features, respectively?

- A. items_for_user, nusers, users_for_item, nitems
- B. users_for_item, nusers, items_for_user, nitems
- C. items_for_user, nitems, users_for_item, nusers
- D. users_for_item, nitems, items_for_user, nusers

However, when assigning the input columns feature, we should use the users_for_item TF Record because now we are looking at a unique item that has had multiple users that interacted with it. The vocab size should be nusers since when making our input columns SparseTensor we will use this vocab_size to for our indices tensor since each item has interacted with a subset of the entire user base, or vocabulary.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

Great, we've now had an overview of our WALS Matrix Factorization estimator input function starting from our TF Record files creating our SparseTensor features of rows for users and columns for items.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

But now let's dive even deeper into the input function, starting with an examination of our custom `decode_example` function.

Decode the TF Record files and invoke `sparse_merge` to create the necessary `SparseTensor`.

We've seen the overview of our TF record parsing function. Let's dive deeper into some of its components, namely the custom functions, starting with our decode example function.

Decode the TF Record files and invoke sparse_merge to create the necessary SparseTensor

```
def decode_example(proto, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
                'indices': tf.VarLenFeature(dtype=tf.int64),
                'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(proto, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                            vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
                                             values = tf.concat([values.values, [0.0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

We call our decode example, using a dataset map, where x is our example protos, and our vocab size is either the number of items or users, depending on which phase of WALs we're in.

Decode the TF Record files and invoke `sparse_merge` to create the necessary SparseTensor

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([], tf.int64),
               'indices': tf.VarLenFeature(dtype=tf.int64),
               'values': tf.VarLenFeature(dtype=tf.float32)}

    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                           vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
                                           values = tf.concat([values.values, [0.0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor
```

Specify the features that
were saved in the TF Record
file

```
def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

We first want to specify the schema of the TF Record files, using a features dictionary, into either fixed length or variable length features. Let's say we were decoding the item TF Record. Then the key will be the item index, which is just one value, therefore it is fixed length. The indices will be the user indices, and there are a variable number of these, because some items might have few ratings, while other items may have many ratings. Lastly, the values will be the ratings, which will be the same variable length as the indices.

Decode the TF Record files and invoke sparse_merge to create the necessary SparseTensor

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
                'indices': tf.VarLenFeature(dtype=tf.int64),
                'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                             vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
                                             values = tf.concat([values.values, [0.0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

Parse from the file

Using the feature map dictionary, we will now parse the features from the example protos, one example at a time.

Decode the TF Record files and invoke `sparse_merge` to create the necessary `SparseTensor`

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
                'indices': tf.VarLenFeature(dtype=tf.int64),
                'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'], Create SparseTensor
    vocab_size=vocab_size)

    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
        values = tf.concat([values.values, [0.0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

Now we want to use `tf.sparse_merge` to easily convert our indices and values into a `SparseTensor`, using the `vocab_size` of the number of items or users, depending on which phase WALS is in. Having the `SparseTensor` format representation of our data is a great choice since our matrix is extremely sparse due to a very high number of users and items. This way using the vocab size we only have to keep track of the interaction pairs that happened and not all of the interaction pairs that didn't through the indices tensor that corresponds with the values tensor that we setup when writing our TF Records during preprocessing.

Decode the TF Record files and invoke sparse_merge to create the necessary SparseTensor

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
               'indices': tf.VarLenFeature(dtype=tf.int64),
               'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                           vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
                                           values = tf.concat([values.values, [0.0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

Save key (itemId or userId)

Remember how we mentioned earlier that the batching process will replace, for instance, the itemId, in the first dimension of the SparseTensor's indices, with the index within the batch? Well, here we want to save the key into a tensor so that we can use it to remap the indices after batching is complete.

Decode the TF Record files and invoke sparse_merge to create the necessary SparseTensor

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
                'indices': tf.VarLenFeature(dtype=tf.int64),
                'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                             vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis=-1),
                                             values=tf.concat([values.values, [0.0]], axis=-1), dense_shape=values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

Splice key into SparseTensor with
tf.concat

Because this is a map, we are mapping the example protos in our dataset to sparse tensors, but we need some way to not lose our key so that we can fix the indices later; some place to store the key. A solution is to take the SparseTensor created by the sparse_merge operation and create a new SparseTensor, but concatenate the key to the end of the indices tensor within. Because the indices and values tensors need to be of the same length, we also concatenate zero to the end of the values tensor as a dummy placeholder. This way the key is spliced into our mapping, which we can then extract later to fix the indexing.

Decode the TF Record files and invoke sparse_merge to create the necessary SparseTensor

```
def decode_example(protos, vocab_size):
    features = {'key': tf.FixedLenFeature([1], tf.int64),
                'indices': tf.VarLenFeature(dtype=tf.int64),
                'values': tf.VarLenFeature(dtype=tf.float32)}
    parsed_features = tf.parse_single_example(protos, features)
    values = tf.sparse_merge(parsed_features['indices'], parsed_features['values'],
                             vocab_size=vocab_size)
    # Save key to remap after batching
    key = parsed_features['key']
    decoded_sparse_tensor = tf.SparseTensor(indices=tf.concat([values.indices, [key]], axis = 0),
                                             values = tf.concat([values.values, [0]], axis = 0), dense_shape = values.dense_shape)
    return decoded_sparse_tensor

def parse_tfrecords(filename, vocab_size):
    ...
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    ...
```

We are finally done designing our SparseTensor and return it to complete the dataset mapping operation.

Quiz

In our `decode_example` function, we use `VarLenFeature` indices and values because of _____, we perform a `sparse_merge` because of _____, and we concatenate the key because of _____.

- A. having many ratings per row/col, needing `SparseTensors`, batching
- B. batching, needing `SparseTensors`, having many ratings per row/col
- C. having many ratings per row/col, batching, needing `SparseTensors`
- D. needing `SparseTensors`, having many ratings per row/col, batching

Now that we've gone through the `decode_example` function, let's see what we have learned. In our `decode_example` function, we use `VarLenFeature` indices and values of `BLANK`, we perform a `sparse_merge` because of `BLANK`, and we concatenate the key because of `BLANK`. Choose the answer that best fills in the blanks.

Quiz

In our `decode_example` function, we use `VarLenFeature` indices and values because of _____, we perform a `sparse_merge` because of _____, and we concatenate the key because of _____.

- A. having many ratings per row/col, needing `SparseTensors`, batching
- B. batching, needing `SparseTensors`, having many ratings per row/col
- C. having many ratings per row/col, batching, needing `SparseTensors`
- D. needing `SparseTensors`, having many ratings per row/col, batching

The correct answer is A! In our `decode_example` function, we use `VarLenFeature` indices and values because of having many ratings per row/col. For instance, when predicting items for users, each user will probably have a different number of items they have interacted with, the indices, and corresponding ratings, the values.

We perform a sparse merge because the `WALSMatrixFactorization`'s input function needs the inputs to be `SparseTensors`.

We concatenate the key to the indices tensor because batching overwrites the first dimension of the indices with the batch index, so we use this trick and the `remap_keys` function later to correct it.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

We now know how to decode our TF Record example protos into SparseTensors, but we're not done yet.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

After we repeat and batch our decoded examples we then have to remap the stored keys to undo the batch reindexing. Let's take a look at how that works.

Remap keys to SparseTensor to fix re-indexing after batching.

Proprietary + Confidential

Google Cloud

Remember several steps ago, when we concatenated the keys to the indices tensor within our SparseTensors? Well, now we finally get to use them to fix the re-indexing that happens during batching, where the itemid or userid gets replaced by the example index within the batch.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):  
    ...  
    return remapped_sparse_tensor  
  
def parse_tfrecords(filename, vocab_size):  
    ...  
    dataset = dataset.map(lambda x: remap_keys(x))  
    ...
```

We first want to map our current dataset, that just finished batching to a new dataset that has the fixed indices, so we call `dataset.map` and pass our batch of SparseTensors to our custom `remap_keys` function.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    # Current indices of our SparseTensor that we need to fix
    bad_indices = sparse_tensor.indices
    # Current values of our SparseTensor that we need to fix
    bad_values = sparse_tensor.values

    # Group by the batch_indices and get the count for each
    size = tf.segment_sum(data = tf.ones_like(bad_indices[:,0]), dtype = tf.int64),
        segment_ids = bad_indices[:,0]) - 1
    # The number of batch_indices (this should be batch_size unless it is a partially full batch)
    length = tf.shape(size, out_type = tf.int64)[0]
    # Finds the cumulative sum which we can use for indexing later
    cum = tf.cumsum(size)
    # The offsets between each example in the batch due to concatenation of the keys in
    decode_example
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)
    # Indices of the SparseTensor of the rows added by concatenation of keys in decode_example
    cum_range = cum + length_range
    ...
```

Google Cloud

Once in the function, we want to store the incorrect indices and values from the batch of SparseTensors into their own tensors. There is a lot of tensor manipulation in this function, and we don't want to overwrite the wrong thing, and/or we might need some buffer space, so we will use these two tensors later in the function.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    # Current indices of our SparseTensor that we need to fix
    bad_indices = sparse_tensor.indices
    # Current values of our SparseTensor that we need to fix
    bad_values = sparse_tensor.values

    # Group by the batch indices and get the count for each
    size = tf.segment_sum(data = tf.ones_like(bad_indices), 0, dtype = tf.int64)
    segment_ids = bad_indices[:, 0]

    # The number of batch_indices (this should be batch_size unless it is a partially full batch)
    length = tf.shape(size, out_type = tf.int64)[0]
    # Finds the cumulative sum which we can use for indexing later
    cum = tf.cumsum(size)
    # The offsets between each example in the batch due to concatenation of the keys in
    decode_example
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)
    # Indices of the SparseTensor of the rows added by concatenation of keys in decode_example
    cum_range = cum + length_range
    ...
```

Google Cloud

Next, we want to group by the batch indices and get the count for each. This way, we will know how many instances there were per batch example. We do this by creating a tensor of all ones for each batch example, as the segment data in the segment sum, and using the same slice of the bad indices tensor, as the segment ids, to group into, which are just the batch indices. Effectively, it's a group-by operation of batch index, with the aggregate count of number of users or items for each. Don't forget to subtract one from the segment sum, because we don't want to count our concatenated keys in the totals.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    # Current indices of our SparseTensor that we need to fix
    bad_indices = sparse_tensor.indices
    # Current values of our SparseTensor that we need to fix
    bad_values = sparse_tensor.values

    # Group by the batch_indices and get the count for each
    size = tf.segment_sum(data = tf.ones_like(bad_indices[:,0]), dtype = tf.int64),
        segment_ids = bad_indices[:,0]) - 1
    # The number of batch indices (this should be batch size unless it is a partially full batch)
    length = tf.shape(size, out_type = tf.int64)[0]
    # Finds the cumulative sum which we can use for indexing later
    cum = tf.cumsum(size)
    # The offsets between each example in the batch due to concatenation of the keys in
    decode_example
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)
    # Indices of the SparseTensor of the rows added by concatenation of keys in decode_example
    cum_range = cum + length_range
    ...
```

Google Cloud

We also want to get the number of batch indices. Normally this would just be the batch size, but in the event of a partially filled batch, it will be smaller, and we need to know this number. This length is essentially just the first dimension of the shape of our grouped-by size tensor.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    # Current indices of our SparseTensor that we need to fix
    bad_indices = sparse_tensor.indices
    # Current values of our SparseTensor that we need to fix
    bad_values = sparse_tensor.values

    # Group by the batch_indices and get the count for each
    size = tf.segment_sum(data = tf.ones_like(bad_indices[:,0]), dtype = tf.int64),
        segment_ids = bad_indices[:,0]) - 1
    # The number of batch_indices (this should be batch_size unless it is a partially full batch)
    length = tf.shape(size, out_type = tf.int64)[0]
    # Find the cumulative sum which we can use for indexing later
    cum = tf.cumsum(size)
    # The offsets between each example in the batch due to concatenation of the keys in
    decode_example
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)
    # Indices of the SparseTensor of the rows added by concatenation of keys in decode_example
    cum_range = cum + length_range
    ...
```

Google Cloud

We need to know the offsets of our actual data from the keys, with a key inserted after every chunk of real data, which if you remember are variable length, so there is no simple way *a priori* know where to slice or skip. To aid us in this, we can find the cumulative sum of the size tensor, which can help us with the global re-indexing of the batch.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    # Current indices of our SparseTensor that we need to fix
    bad_indices = sparse_tensor.indices
    # Current values of our SparseTensor that we need to fix
    bad_values = sparse_tensor.values

    # Group by the batch_indices and get the count for each
    size = tf.segment_sum(data = tf.ones_like(bad_indices[:,0]), dtype = tf.int64),
        segment_ids = bad_indices[:,0]) - 1
    # The number of batch_indices (this should be batch_size unless it is a partially full batch)
    length = tf.shape(size, out_type = tf.int64)[0]
    # Finds the cumulative sum which we can use for indexing later
    cum = tf.cumsum(size)

    # The offsets between each example in the batch due to concatenation of the keys in
    decode_example =
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)
    # Indices of the SparseTensor of the rows added by concatenation of keys in decode_example
    cum_range = cum + length_range
    ...
```

Additionally, to know the offsets between each example in the batch, due to our storage solution of concatenating the keys in `decode_example`, we will need to create a tensor that has a range from 0 inclusive, to the number of batch examples exclusive. `tf.range` makes this easy, and we can use our `length` tensor, which is just the number of batch indices, to create this range.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    # Current indices of our SparseTensor that we need to fix
    bad_indices = sparse_tensor.indices
    # Current values of our SparseTensor that we need to fix
    bad_values = sparse_tensor.values

    # Group by the batch_indices and get the count for each
    size = tf.segment_sum(data = tf.ones_like(bad_indices[:,0]), dtype = tf.int64),
        segment_ids = bad_indices[:,0]) - 1
    # The number of batch_indices (this should be batch_size unless it is a partially full batch)
    length = tf.shape(size, out_type = tf.int64)[0]
    # Finds the cumulative sum which we can use for indexing later
    cum = tf.cumsum(size)
    # The offsets between each example in the batch due to concatenation of the keys in
    decode_example
    length_range = tf.range(start = 0, limit = length, delta = 1, dtype = tf.int64)
    # indices of the SparseTensor of the rows added by concatenation of keys in decode_example
    cum_range = cum + length_range
    ...
```

Google Cloud

Lastly, in this segment of the function, adding the length range to the cumulative sum will give us the cumulative range, which are the indices of the fake data we added to the SparseTensor, by concatenating the keys in decode_example.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # The keys that we have extracted back out of our enumerated SparseTensor
    gathered_indices = tf.squeeze(tf.gather(bad_indices, cum_range)[0,1])

    # The enumerated row indices of the SparseTensor's indices member
    sparse_indices_range = tf.range(tf.shape(bad_indices)[0], dtype = tf.int64)
    ...
```

We want to get the key values we stored within the indices tensor. To do that, we first use a `tf.gather`, with the bad indices tensor we made at the beginning of the function, from the input `SparseTensor`, as our params, and using the cumulative range we calculated earlier, as the gather's indices. We slice the tensor only to keep the second dimension, because the first dimension is the batch index, which we definitely don't want—and in fact are trying to get rid of—and the second dimension has our key. We then squeeze the tensor to get it into the correct shape of a vector.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # The keys that we have extracted back out of our concatenated SparseTensor
    gathered_indices = tf.squeeze(tf.gather(bad_indices, cum_range)[:,:1])

    # The enumerated row indices of the sparse_tensor's indices tensor
    sparse_indices_range = tf.range(tf.shape(bad_indices)[0], dtype=tf.int64)
    ...
```

For us to be able to remove our concatenated rows, we first need to know how many rows we have in the batch total, and then enumerate the range of row indices. We do this first by finding the shape of our indices tensor and taking the first dimension. We then use `tf.range`, with that scalar we just found, as our limit to get the enumeration.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Want to find the row indices of the SparseTensor that are actual data & not the concatenation
    # So we want to find the intersection of the two sets and then take the opposite of that
    s = sparse_indices_range
    s = cum_range

    # Number of multiples we are going to tile x, which is our sparse_indices_range
    tile_multiples = tf.concat([tf.ones(tf.shape(tf.shape(x))), dtype=tf.int64),
                               tf.shape(s, out_type = tf.int64)], axis = 0)
    # Expands x, our sparse_indices_range, into a rank 2 tensor
    # Then multiplies the rows by 1 (no copying) and the columns by the number of examples in the
    batch
    x_tile = tf.tile(tf.expand_dims(x, -1), tile_multiples)
    # Essentially a vectorized logical or, that we then negate
    x_not_in_s = ~tf.reduce_any(tf.equal(x_tile, s), -1)
    ...
```

Google Cloud

After all of that work, we now have all of the pieces to find the row indices of the actual data, and can drop the key rows from the indices tensor. We have created two sets of indices, and in our venn diagram, we want the opposite of the intersection. First, we are going to copy some older tensors into new ones that we can play with called *x*, which will be the sparse indices enumerated range we just made in the last step, and *s*, which is the cumulative sum range, which are the indices of the keys. We want the indices of the **NOT** keys!

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Want to find the row indices of the SparseTensor that are actual data & not the concatenated
rows
    # So we want to find the intersection of the two sets and then take the opposite of that
    x = sparse_indices_range
    s = cum_range

    # Number of multiples we are going to tile x, which is our sparse indices range
    tile_multiples = tf.concat([tf.ones(tf.shape(tf.shape(x)), dtype=tf.int64),
                                tf.shape(s, out_type = tf.int64)], axis = 0)

    # Expands x, our sparse_indices_range, into a rank 2 tensor
    # Then multiplies the rows by 1 (no copying) and the columns by the number of examples in the
batch
    x_tile = tf.tile(tf.expand_dims(x, -1), tile_multiples)
    # Essentially a vectorized logical or, that we then negate
    x_not_in_s = ~tf.reduce_any(tf.equal(x_tile, s), -1)
    ...
```

We will be doing some tiling. First, we need to calculate the number of multiples we are going to tile x, our sparse indices range. We do this by first finding the shape of x, and finding the shape of that which will essentially be the number of dimensions of x, and creating a tensor of 1s of that shape. This gets concatenated with the shape of s, our cumulative sum range, which is just the number of keys.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Want to find the row indices of the SparseTensor that are actual data & not the concatenated
rows
    # So we want to find the intersection of the two sets and then take the opposite of that
    x = sparse_indices_range
    s = cum_range

    # Number of multiples we are going to tile x, which is our sparse_indices_range
    tile_multiples = tf.concat([tf.ones(tf.shape(tf.shape(x))), dtype=tf.int64),
                               tf.shape(s, out_type = tf.int64)], axis = 0)

    # Expand x, our sparse_indices_range, into a rank 3 tensor
    # Then multiplies the rows by 1 (no copying) and the columns by the number of examples in the
dataset
    x_tile = tf.tile(tf.expand_dims(x, -1), tile_multiples)
    # Essentially a vectorized logical or, that we then negate
    x_not_in_s = ~tf.reduce_any(tf.equal(x_tile, s), -1)
    ...
```

Now that we know how many times we want to tile x, let's tile it! First we are going to expand dims x into a rank 2 tensor by adding a dimension to the end. We then are going to tile that tensor a tile_multiples number of times, calculated from above. This then gives us a rank 2 tensor, with the enumerated indices tiled multiple times, one for each key.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Want to find the row indices of the SparseTensor that are actual data & not the concatenated
rows
    # So we want to find the intersection of the two sets and then take the opposite of that
    x = sparse_indices_range
    s = cum_range

    # Number of multiples we are going to tile x, which is our sparse_indices_range
    tile_multiples = tf.concat([tf.ones(tf.shape(tf.shape(x))), dtype=tf.int64),
                               tf.shape(s, out_type = tf.int64)], axis = 0)

    # Expands x, our sparse_indices_range, into a rank 2 tensor
    # Then multiplies the rows by 1 (no copying) and the columns by the number of examples in the
batch
    x_tile = tf.tile(tf.expand_dims(x, -1), tile_multiples)
    # Essentially a vectorized logical or, that we then negate
    # not in s = -tf.reduce_any(tf.equal(x_tile, s)) -1
    ...
```

Google Cloud

TensorFlow is great for doing vectorized math! Now that we have this tiling, we are going to perform on each tile slice for each key simultaneously. First, we create a boolean tensor, where each vector gets a True at the index of the key, and False elsewhere. We then use a `tf.reduce_any` operation on the last axis, which essentially acts as a logical or, merging all of the boolean vectors into one, where any element that is True corresponds to a key index. Remember, we want the **NOT** key indices, so we need to negate the following tensor so that the True elements will be the elements that are not the keys. We can perform this negation by adding the tilde unary complement operator which will flip all of the 0's to 1's and vice versa.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # The SparseTensor's indices that are our actual data by using the boolean mask we just had
    # above
    # Applied to the entire indices member of our SparseTensor
    selected_indices = tf.boolean_mask(tensor = bad_indices, mask = x_not_in_s, axis = 0)
    # Apply the same boolean_mask to the entire values member of our SparseTensor
    # Gets the actual values data
    selected_values = tf.boolean_mask(tensor = bad_values, mask = x_not_in_s, axis = 0)
    ...
```

What to do with these boolean masks that give the NOT key indices? We can apply them to one of the original tensors we created, the bad indices tensor, which has the original indices tensor that got messed up during batching. We can simply use `tf.boolean_mask` to apply it to our tensor, which will return the indices from our actual data, and drop the keys that we concatenated in. Did we do all of this work to just throw the keys out anyway? Not at all; we'll be using them in just a moment, but we first had to extract the actual indices data.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # The SparseTensor's indices that are our actual data by using the boolean_mask we just made
    above
    # Applied to the entire indices member of our SparseTensor
    selected_indices = tf.boolean_mask(tensor = bad_indices, mask = x_not_in_s, axis = 0)
    # Apply the same boolean mask to the entire values member of our SparseTensor
    # Gets the actual values data
    selected_values = tf.boolean_mask(tensor = bad_values, mask = x_not_in_s, axis = 0)
    ...
```

Of course, this is all being put back into a SparseTensor, where the lengths of the indices and values tensors have to match because they correspond with each other. So we will apply the same boolean mask, except this time to the original values tensor instead of to the indices. This will return our actual value data, and drop the dummy zeros we added to the values tensor, back in the `decode_example` function.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Need to replace the first column of selected indices with key
    # So we first need to tile gathered indices
    tiling = tf.tile(input = tf.expand_dims(gathered_indices[0], -1),
                    multiples = tf.expand_dims(size[0], -1))

    # We have to repeatedly apply the tiling to each example in the batch
    # Since it is jagged we cannot use tf.map_fn due to the stacking of the TensorArray
    # So we have to create our own custom version
    def loop_body(i, tensor_grow):
        return i + 1, tf.concat(values = [tensor_grow,
            tf.tile(input = tf.expand_dims(gathered_indices[i], -1),
                multiples = tf.expand_dims(size[i], -1))], axis = 0)

    _, result = tf.nn.nested_loop(lambda i, tensor_grow: i < length, loop_body,
        [tf.constant(1, dtype = tf.int64), tiling])
    ...
```

Google Cloud

We're getting close to the end now, but we essentially have only returned to the state we would have been in after batching if we never concatenated the keys. So now it's time to make all of this effort worth it, by inserting the keys into the right spot and dropping the batch indices that overwrote them. We will need to replace the first column of the `selected_indices` tensor we just created from our boolean mask with the keys over the batch indices. This will be done through tiling; however it is not as simple as the last time we used tiling. This is due to the variable length of the data. Some items might have 1 user interaction, some might have 5, others 10. The same is true for users who have item interactions. This jagged nature doesn't quite work well with TensorFlow's rectangular tensor structure, so we will have to improvise.

We need to know what we are going to tile and how many multiples we are making of it. First, we expand the first element of the gathered indices tensor, which are our key indices, by adding a dimension to the end of that. The multiples will then be the first element of the size tensor, which is the grouped-by batch index counts. This is just the very first tiling to seed the while loop we are going to do later.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Need to replace the first column of selected_indices with keys
    # So we first need to tile gathered_indices
    tiling = tf.tile(input = tf.expand_dims(gathered_indices[0], -1),
                    multiples = tf.expand_dims(size[0], -1))

    # We have to repeatedly apply the tiling to each example in the batch
    # Since it is jagged we cannot use tf.map_fn due to the stacking of the TensorArray.
    # So we have to create our own custom version
    def loop_body(i, tensor_grow):
        return i + 1, tf.concat(values = [tensor_grow,
            tf.tile(input = tf.expand_dims(gathered_indices[i], -1),
                multiples = tf.expand_dims(size[i], -1)), axis = 0]

    _, result = tf.nn.whilst(lambda i, tensor_grow: i < length, loop_body,
        [tf.constant(1, dtype = tf.int64), tiling])
    ...
```

Google Cloud

We have to repeatedly apply this tiling to each example in the batch. However, because there is a variable length of users or items, it is jagged and we cannot use `tf.map_fn`, due to how it internally does the stacking of the `TensorArray`. So instead, we are going to create our own version of `tf.map_fn` by creating a loop body that we will use in a while loop in the next step.

First, we define a loop body function that takes the loop index as the first parameter, and the tensor that we are growing as the second parameter, that eventually will contain the tiled item IDs, or used IDs, from our stored keys. We will return two values: the loop variable incremented and the tensor we are iteratively growing, by concatenating the previous tensor with the next tiling by sliding along the key indices and the batch index counts tensors.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Need to replace the first column of selected_indices with keys
    # So we first need to tile gathered_indices
    tiling = tf.tile(input = tf.expand_dims(gathered_indices[0], -1),
                     multiples = tf.expand_dims(size[0], -1))

    # We have to repeatedly apply the tiling to each example in the batch
    # Since it is jagged we cannot use tf.map_fn due to the stacking of the TensorArray
    # So we have to create our own custom version
    def loop_body(i, tensor_grow):
        return i + 1, tf.concat(values = [tensor_grow,
                                          tf.tile(input = tf.expand_dims(gathered_indices[i], -1),
                                                multiples = tf.expand_dims(size[i], -1))], axis = 0)

    result = tf.nn.nested_loop(lambda i, tensor_grow: i < length, loop_body,
                               [tf.constant(1, dtype = tf.int64), tf.constant(1, dtype = tf.int64)])
    ...
```

The loop body we just created won't run itself, so we need to call it using `tf.nn.nested_loop`. We call it with our conditions, so that it loops over the full length of our tensors using the loop body function we created, and using our initial loop_vars of 1 for our loop variable, i, and our initial one shot tiling we did two steps back. After the loop completes, it will return the tensor we grew through repeated tiling as our new tensor named result, which will be our key values, item or user indices, expanded out for the correct number of examples that they correspond with.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Concatenate tiled keys with the 2nd column of selected indices
    selected_indices_fixed = tf.concat([tf.expand_dims(result, -1),
                                       tf.expand_dims(selected_indices, -1), -1])
    axis = 1

    # Combine everything together back into a SparseTensor
    remapped_sparse_tensor = tf.SparseTensor(indices = selected_indices_fixed,
                                             values = selected_values,
                                             dense_shape = sparse_tensor.dense_shape)

    return remapped_sparse_tensor
```

We're near the end! We have two correct but separate tensors. One contains our keys, and one contains our data that is associated with each key. This is the step in which we finally fix the overwriting of the itemID or userID by the batch index that happened during batching.

To do this, we will concatenate the tiled keys we just made with the second column of the indices that we fixed earlier, of the actual data. We have to expand the dimensions of each tensor by one at the end of each tensor, and then we concatenate along columns.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Concatenate tiled keys with the 2nd column of selected_indices
    selected_indices_fixed = tf.concat([tf.expand_dims(result, -1),
                                       tf.expand_dims(selected_indices[:, 1], -1)],
                                       axis = 1)

    # Combine everything together back into a SparseTensor
    remapped_sparse_tensor = tf.SparseTensor(indices = selected_indices_fixed,
                                             values = selected_values,
                                             dense_shape = sparse_tensor.dense_shape)

    return remapped_sparse_tensor
```

Awesome! All we need to do now is put everything back in SparseTensor format, so that the WALS input function can use it correctly. The indices will be the indices we just fixed in the last step by concatenating the tiled keys with our fixed indices. The values will be the values that we fixed by removing the dummy zeros we added in the decode_example function. The dense shape will be the same dense shape of the input sparse tensor to the remap_keys function.

Remap keys to SparseTensor to fix re-indexing after batching

```
def remap_keys(sparse_tensor):
    ...
    # Concatenate tiled keys with the 2nd column of selected_indices
    selected_indices_fixed = tf.concat([tf.expand_dims(result, -1),
                                       tf.expand_dims(selected_indices[:, 1], -1)],
                                       axis = 1)

    # Combine everything together back into a SparseTensor
    remapped_sparse_tensor = tf.SparseTensor(indices = selected_indices_fixed,
                                             values = selected_values,
                                             dense_shape = sparse_tensor.dense_shape)

    return remapped_sparse_tensor
```

We're done. Return the remapped sparse tensor, and we will return to the `parse_tfrecords` function scope, where it will call the one shot iterator, and then go back up in scope to the `WALS` input function, for `INPUT_ROWS` and `INPUT_COLS`. This was a lot of work to remap the keys back into the input data `SparseTensors`, but it was completely essential. Without it, the `WALSMatrixFactorization` estimator will not perform the necessary sweeps, because the batch index keeps resetting back to zero every batch, and the model will not train. With this, the sweeps will perform correctly, and the model will learn from the data.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Wow, we now know how to remap the keys to fix the re-indexing by the batching process, so let's see how much we've learned. Here's sample data for items_for_user with the following schema: user_index, item_indices, ratings.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[0,0],[0,1],[0,1]]
values_batch1=[2.5,0.5,2.5,0.5]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[0,2],[0,3]]
values_batch2=array([2.5,1.5,4.5,0.5,3.5,0.5])
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size equals 2.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

Match the tensor elements to what they represent based on color.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

The correct answer is D!

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

The red highlighted elements in the indices tensors are the batch_indices, which are the first dimension of the indices tensor. Notice that the values range from 0 to 1 because our batch_size is 2, so there should only be two values.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

The green highlighted elements are the item_indices in the indices tensors. This is what we wrote and read in as indices with the TF Records.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

The blue highlighted elements are the user_indices in the indices tensors. This is what we wrote and read in as key with the TF Records, and concatenated these to the indices tensor in the decode_example function.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

The yellow highlighted elements are the ratings. This is what we wrote and read in as values with the TF Records.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[0,0],[1,0],[1,1],[1,1]]
values_batch1=[2.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

Match tensor elements to names based on color.

- A. batch_index, userId, itemId, dummy_value, rating
- B. userId, batch_index, itemId, rating, dummy_value
- C. itemId, batch_index, userId, rating, dummy_value
- D. batch_index, itemId, userId, rating, dummy_value
- E. batch_index, userId, itemId, rating, dummy_value

The purple highlighted elements are the dummy_values. We concatenated these to the values tensor in the decode_example function to ensure that the indices and values tensors had the same length.

Quiz

items_for_user schema: userId, itemId, rating

```
0 [2] [1.5]
1 [0,1] [3.5, 2.5]
2 [0,1,2] [2.5,1.5,4.5]
3 [2] [3.5]
```

Here are the first two batches of SparseTensors passed to remap_keys with batch_size = 2.

```
indices_batch1=[[0,2],[1,0],[1,0],[1,1],[1,1]]
values_batch1=[1.5,0,3.5,2.5,0]
indices_batch2=[[0,0],[0,1],[0,2],[0,2],[1,2],[1,2],[1,3]]
values_batch2=array([2.5,1.5,4.5,0,3.5,0])
```

SparseTensors fixed after remap_keys function:

```
indices_batch1=[[0,2],[1,0],[1,1]]
values_batch1=[1.5,3.5,2.5]
indices_batch2=[[2,0],[2,1],[2,2],[3,2]]
values_batch2=array([2.5,1.5,4.5,3.5])
```

The remap_keys function wants to replace the batch_indices with their corresponding user_indices and drop the added elements of the user_indices and dummy_values. Here at the bottom is what the SparseTensors will look like after the remap_keys function fixes them. Notice that the batch_indices have been replaced with the correct user_indices, and we eliminated the extraneous rows with the user_indices and dummy_values.

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):  
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))  
    dataset = tf.data.TFRecordDataset(files)  
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))  
    dataset = dataset.repeat(num_epochs)  
    dataset = dataset.batch(args['batch_size'])  
    dataset = dataset.map(lambda x: remap_keys(x))  
    return dataset.make_one_shot_iterator().get_next()  
  
def _input_fn():  
    features = {  
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',  
args['nitems']),  
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])  
    }  
    return features, None
```

Excellent, we have our SparseTensors fixed now after our remap keys function!

The input function has to read the files and create sparse tensors for the rows and for the columns

```
def parse_tfrecords(filename, vocab_size):
    files = tf.gfile.Glob(os.path.join(args['input_path'], filename))
    dataset = tf.data.TFRecordDataset(files)
    dataset = dataset.map(lambda x: decode_example(x, vocab_size))
    dataset = dataset.repeat(num_epochs)
    dataset = dataset.batch(args['batch_size'])
    dataset = dataset.map(lambda x: remap_keys(x))
    return dataset.make_one_shot_iterator().get_next()

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user',
args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers'])
    }
    return features, None
```

There are a few items of business left to take care of before we are done with our WALSMatrix Factorization input function, namely how many times we should repeat and if we can add additional features.

During evaluation, go through dataset only once and specify whether we are recommending items or targeting users

```
def parse_tfrecords(tfrecords, vocab_size):
    if mode == tf.estimator.ModeKeys.TRAIN:
        num_epochs = None # indefinitely
    else:
        num_epochs = 1 # end-of-input after this
    ...
    dataset = dataset.repeat(num_epochs)
    ...

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user', args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers']),
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return features, None
```

At the top of our `parse_tfrecords` function, we need to determine the number of epochs we are going to go through, because we are using `num_epochs` in `dataset.repeat`. As usual, if we are in training, we want to cycle through the dataset over and over until the estimator reaches the number of train steps. For evaluation, we want to run through the dataset only once.

During evaluation, go through dataset only once and specify whether we are recommending items or targeting users

```
def parse_tfrecords(tfrecords, vocab_size):
    if mode == tf.estimator.ModeKeys.TRAIN:
        num_epochs = None # indefinitely
    else:
        num_epochs = 1 # end-of-input after this
    ...
    dataset = dataset.repeat(num_epochs)
    ...

def _input_fn():
    features = {
        WALSMatrixFactorization.INPUT_ROWS: parse_tfrecords('items_for_user', args['nitems']),
        WALSMatrixFactorization.INPUT_COLS: parse_tfrecords('users_for_item', args['nusers']),
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return features, None
```

Usually when we think about recommendation systems, we think of a model that recommends items to users, whether that is items to buy, places to visit, or movies to watch. But why should we always choose a user and return items? Couldn't we also provide the transpose, since WALs solves both users and items simultaneously? We could be developing a “targeting” model; in such models, we might be trying to decide which users to send coupons for an item to, or who best to call for polling, etc. To flip our recommendations from users to items, we can add a feature to our input function's feature dictionary called `WALSMatrixFactorization.PROJECT_ROW`. This is a boolean tensor that, if set to true, will project rows, and if set to false, will project columns during inference.

The train_and_evaluate loop is typical of tf.contrib Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']))
    batch_size = 10

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

        export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
            serving_input_fn=create_serving_input_fn(args))
        )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

So now that we've gone through the entire input function, let's see how it fits in with the estimator. Here is our train input function, where we pass the TRAIN estimator modekeys, so that we'll go an indefinite number of epochs through the dataset until we reach our number of train steps, calculated above.

The train_and_evaluate loop is typical of tf.contrib Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']) /
args['batch_size'])

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

    export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
        serving_input_fn=create_serving_input_fn(args))
    )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

And here is our eval input function, where we will go through the dataset only once.

The train_and_evaluate loop is typical of **tf.contrib** Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']) /
args['batch_size'])

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
serving_input_fn=create_serving_input_fn(args))
        )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

Remember, at the moment, WALS is a **contrib**.Estimator so I'm using Experiment and learn_runner.

The current implementation of WALS handles batching, but not distribution. When it is updated to also handle distribution, it will move to core. The goal is that all core Estimators distribute to multiple machines without any code changes. Here is the call within Experiment to set up the WALSMatrixFactorization estimator.

The train_and_evaluate loop is typical of tf.contrib Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']) /
args['batch_size'])

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

            export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
                serving_input_fn=create_serving_input_fn(args)
            )
        )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

We still need to create a serving input function for making recommendations, but we'll get to that later.

The train_and_evaluate loop is typical of tf.contrib Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']) /
args['batch_size'])

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

            export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
                serving_input_fn=create_serving_input_fn(args))
        )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

Lastly, kick off the action with the learn_runner running our experiment function.

Typically, just store the top K items per user.

Proprietary + Confidential

$$A \approx U \times V^T$$

Google Cloud

Using the WALS Matrix Factorization model, it is possible to predict the ratings between all items and users, if given the row and column factors. However, typically we just store the top K items per user or users per item to save space and computation by not filling out the entire rating matrix, which can be very large.

Finding top K items for a user

Proprietary + Confidential

```
def find_top_k(user_factors, item_factors, k):  
    all_items = tf.matmul(tf.expand_dims(user_factors, 0),  
                           tf.transpose(item_factors))  
    topk = tf.nn.top_k(all_items, k=k)  
    return tf.cast(topk.indices, dtype=tf.int64)
```

Google Cloud

Let's define a function called `find_top_k` that takes the `user_factors` for a particular user, `U`, and the `item_factors` of all items, `V`, that the WALS Matrix Factorization model solved for. We also want to pass to the function `k`, so it knows how many to return.

Finding top K items for a user

Proprietary + Confidential

```
def find_top_k(user_factors, item_factors, k):  
    all_items = tf.matmul(tf.expand_dims(user_factors, 0),  
                           tf.transpose(item_factors))  
    topk = tf.nn.top_k(all_items, k=k)  
    return tf.cast(topk.indices, dtype=tf.int64)
```

Google Cloud

Instead of multiplying a vector from each factor matrix to get a specific user-item rating prediction, as we did earlier in our user-movie example, we can matrix multiply instead to get all of the ratings for the user. Because this is just for one user, `user_factors` is a vector specific to that user. Before we can do the matrix multiply, we need to expand dims of `user_factors` to make it a rank 2 tensor. Thus, we're matrix multiplying a 1 x latent factors matrix with a latent factors by items matrix, which will result in a 1 x items matrix.

Finding top K items for a user

Proprietary + Confidential

```
def find_top_k(user_factors, item_factors, k):  
    all_items = tf.matmul(tf.expand_dims(user_factors, 0),  
                           tf.transpose(item_factors))  
    topk = tf.nn.top_k(all_items, k=k)  
    return tf.cast(topk.indices, dtype=tf.int64)
```

Google Cloud

Now that we have a tensor of all the item ratings for a user, we can use `tf.nn.top_k` to get a tensor object of the top k values and their associated indices, which would be the itemIDs in this case.

Finding top K items for a user

Proprietary + Confidential

```
def find_top_k(user_factors, item_factors, k):  
    all_items = tf.matmul(tf.expand_dims(user_factors, 0),  
                           tf.transpose(item_factors))  
    topk = tf.nn.top_k(all_items, k=k)  
    return tf.cast(topk.indices, dtype=tf.int64)
```

Google Cloud

Lastly, we want to return the top k items for our user, so we'll return the indices member tensor of our topk object. Because these are indices, we'll cast them to integers before we exit the function.

Finding top K items for all users can be done as a batch prediction job

Proprietary + Confidential

```
def batch_predict(args):  
    # Create TF Session as sess  
    estimator = tf.contrib.factorization.WALSMatrixFactorization(  
        num_rows=args['nusers'], num_cols=args['nitems'],  
        embedding_dimension=args['n_embeds'],  
        model_dir=args['output_dir'])  
    user_factors = tf.convert_to_tensor(estimator.get_row_factors()[0]) # (nusers, nembeds)  
    item_factors = tf.convert_to_tensor(estimator.get_col_factors()[0]) # (nitems, nembeds)  
    # for each user, find the top K items  
    topk = tf.squeeze(tf.map_fn(lambda user: find_top_k(user, item_factors, args['topk']),  
user_factors,  
dtype=tf.int64))  
    with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:  
        for best_items_for_user in topk.eval():  
            f.write(' '.join(str(x) for x in best_items_for_user) + '\n')
```

Google Cloud

If we want to find the top K items for all users, we can create a batch prediction job. Note the session creation. This is a complete tensorflow function. It is not part of any other graph.

Finding top K items for all users can be done as a batch prediction job

Proprietary + Confidential

```
def batch_predict(args):
    with tf.Session() as sess:
        estimator = tf.contrib.factorization.WrappedFactorization
        num_rows=args['nusers'], num_cols=args['nitems'],
        embedding_dimension=args['n_embeds'],
        model_dir=args['output_dir'])

        user_factors = tf.convert_to_tensor(estimator.get_row_factors()[0]) # (nusers, nembeds)
        item_factors = tf.convert_to_tensor(estimator.get_col_factors()[0])# (nitems, nembeds)
        # for each user, find the top K items
        topk = tf.squeeze(tf.map_fn(lambda user: find_top_k(user, item_factors, args['topk']),
        user_factors,
            dtype=tf.int64))
        with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
            for best_items_for_user in topk.eval():
                f.write(' '.join(str(x) for x in best_items_for_user) + '\n')
```

Google Cloud

We define our estimator, which will load the model from our output directory we wrote to during training.

Finding top K items for all users can be done as a batch prediction job

Proprietary + Confidential

```
def batch_predict(args):
    with tf.Session() as sess:
        estimator = tf.contrib.factorization.WALSMatrixFactorization(
            num_rows=args['nusers'], num_cols=args['nitems'],
            embedding_dimension=args['n_embeds'],
            model_dir=args['output_dir'])
        user_factors = tf.convert_to_tensor(estimator.get_row_factors()[0]) # (nusers, nembeds)
        item_factors = tf.convert_to_tensor(estimator.get_col_factors()[0]) # (nitems, nembeds)
        # for each user, find the top K items
        topk = tf.squeeze(tf.map_fn(lambda user: find_top_k(user, item_factors, args['topk']),
            user_factors,
            dtype=tf.int64))
        with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
            for best_items_for_user in topk.eval():
                f.write(', '.join(str(x) for x in best_items_for_user) + '\n')
```

Google Cloud

We extract the user factors from our estimator by getting the row factors which will be a rank 2 tensor, with shape number of users by number of embedding dimensions.

Finding top K items for all users can be done as a batch prediction job

Proprietary + Confidential

```
def batch_predict(args):
    with tf.Session() as sess:
        estimator = tf.contrib.factorization.WALSMatrixFactorization(
            num_rows=args['nusers'], num_cols=args['nitems'],
            embedding_dimension=args['n_embeds'],
            model_dir=args['output_dir'])
        user_factors = tf.convert_to_tensor(estimator.get_row_factors()[0]) # (nusers, nembeds)
        item_factors = tf.convert_to_tensor(estimator.get_col_factors()[0]) # (nitems, nembeds)
        # for each user, find the top K items
        topk = tf.squeeze(tf.map_fn(lambda user: find_top_k(user, item_factors, args['topk']),
            user_factors,
            dtype=tf.int64))
        with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
            for best_items_for_user in topk.eval():
                f.write(' '.join(str(x) for x in best_items_for_user) + '\n')
```

Google Cloud

Likewise, we extract the item factors from our estimator by getting the column factors which will be a rank 2 tensor, with shape number of items by number of embedding dimensions.

Finding top K items for all users can be done as a batch prediction job

Proprietary + Confidential

```
def batch_predict(args):
    with tf.Session() as sess:
        estimator = tf.contrib.factorization.WALSMatrixFactorization(
            num_rows=args['nusers'], num_cols=args['nitems'],
            embedding_dimension=args['n_embeds'],
            model_dir=args['output_dir'])
        user_factors = tf.convert_to_tensor(estimator.get_row_factors()[0]) # (nusers, nembeds)
        item_factors = tf.convert_to_tensor(estimator.get_col_factors()[0]) # (nitems, nembeds)
        # for each user, find the top K items
        topk = tf.squeeze(tf.map_fn(lambda user: find_top_k(user, item_factors, args['topk']),
            user_factors,
            dtype=tf.int64))
        with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
            for best_items_for_user in topk.eval():
                f.write(' '.join(str(x) for x in best_items_for_user) + '\n')
```

Google Cloud

We can reuse the `find_top_k` function we just made and put it in a map function. We'll use our entire `user_factors` matrix as the elements to our map function, where it will pass a row or user from the matrix via the lambda function, to our `find_top_k` function as user. This will create a stack of matrices so we will squeeze it to get a rank 2 tensor with shape users by topk.

Finding top K items for all users can be done as a batch prediction job

Proprietary + Confidential

```
def batch_predict(args):
    with tf.Session() as sess:
        estimator = tf.contrib.factorization.WALSMatrixFactorization(
            num_rows=args['nusers'], num_cols=args['nitems'],
            embedding_dimension=args['n_embeds'],
            model_dir=args['output_dir'])
        user_factors = tf.convert_to_tensor(estimator.get_row_factors()[0]) # (nusers, nembeds)
        item_factors = tf.convert_to_tensor(estimator.get_col_factors()[0]) # (nitems, nembeds)
        # for each user, find the top K items
        topk = tf.squeeze(tf.map_fn(lambda user: find_top_k(user, item_factors, args['topk']),
            user_factors,
            dtype=tf.int64))
        with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
            for best_items_for_user in topk.eval():
                f.write(','.join(str(x) for x in best_items_for_user) + '\n')
```

Google Cloud

We'll create a FILE IO stream to our batch prediction output file and will create a loop where we evaluate the topk within the session we created. Each row in the file, in this case, will be a comma-delimited string of the topk best items for that user.

Quiz

We saw how to recommend the top K items for users, but what if we wanted to instead recommend the top K users for items? What would be the correct change in our batch prediction function?

```
topk = tf.squeeze(tf.map_fn(lambda ____:  
    find_top_k(____, ____, args['topk']),  
    ____, dtype=tf.int64))
```

- A. user, user, item_factors, user_factors
- B. item, item, user_factors, item_factors
- C. user, item, user_factors, item_factors
- D. item, item, item_factors, user_factors
- E. item, user, item_factors, user_factors
- F. user, user, user_factors, item_factors

Let's now test your knowledge! We saw how to recommend the top k items for users, but what if we wanted to instead recommend the top k users for items? Which would be the correct change in our batch prediction function?

Choose the answer that best fills in the blanks.

Quiz

We saw how to recommend the top K items for users, but what if we wanted to instead recommend the top K users for items? What would be the correct change in our batch prediction function?

```
topk = tf.squeeze(tf.map_fn(lambda ____:  
    find_top_k(____, ____, args['topk']),  
    ____, dtype=tf.int64))
```

- A. user, user, item_factors, user_factors
- B. item, item, user_factors, item_factors**
- C. user, item, user_factors, item_factors
- D. item, item, item_factors, user_factors
- E. item, user, item_factors, user_factors
- F. user, user, user_factors, item_factors

Google Cloud

The correct answer is B! Our changing argument through the lambda function to our `find_top_k` function will be **item** for both. We will be matrix multiplying this item vector, which we will expand dims into a matrix by the **user_factors** matrix. Lastly, the elements we will be mapping from will be from our **item_factors** matrix, of shape number of items by number of embedding dimensions, where it will pull a row per map from.

Lab

Collaborative Filtering with Google Analytics Data

Ryan Gillard

Google Cloud

Now that we have learned how WALS matrix factorization works for collaborative filtering, let's put our new knowledge to action. This lab demonstrates how to implement a WALS matrix factorization approach to do collaborative filtering using Google Analytics data. We'll be using most of the code that we learned earlier. In later modules, we will combine this and content-based with a deep neural network and then learn how to productionize and automate our solution using the Google Cloud Platform ecosystem.

Lab steps

- Complete TODOs in wals.ipynb.
 - Complete `decode_example` function.
 - Add `WALSMatrixFactorization` Estimator within `batch_predict`.

In this lab, we will be completing the TODOs in the wals notebook. Specifically, we will need to complete the `decode_example` function where we decode the TF Record protos and store the result in our dataset. Also, we need to add the `WALSMatrixFactorization` estimator within the `batch_predict` function, making sure to include calls to the input function, serving input function, and our vocabularies.

Our batch predictions were problematic ...

Proprietary + Confidential

```
!head wals_trained/batch_pred.txt
```

```
525,800,5316  
3781,1557,3565  
800,4460,1061  
3096,5146,3781  
4072,3260,3133  
4586,3703,4079  
3432,361,525  
3521,3339,3961  
492,3151,3339  
492,1839,5547
```

Is there a problem?

Google Cloud

So we have our batch predictions and we're ready to continue , right? Well, unfortunately the output is not very human-readable.

For instance, what is itemId 800 on the 3rd line? Who is userId 2, which isn't even listed but we know it is the third line, which corresponds to user index 2? What's the problem here?

We enumerated the users and items, but what we really need are visitorID and contentID, which we can tie back to our original data. We mapped from visitorID and contentID, but we need a reverse mapping to get it back from our enumerations.

What we really need are visitorId and contentId in our prediction files, not the enumerated userId and itemId

```
!head wals_trained/batch_pred.txt
```

6167894456739729438	298997422,262707977,263058146
3795498541234027150	296993188,97034003,298989783
3419059466801506820	299784650,99132425,216326131
1237559046481705676	871054,789235,244589322
6571298744617006597	756838,175262203,223620964
8767400997210088447	299826775,299432251,299086761
8531948287293212270	299826775,299432251,299086761
6669381117463560223	299930675,299809748,294040727
6577198766965438926	299959410,13934718,299695396
6251363516890829952	299324114,293637423,298336025

visitorId

top 3 contentId

What we really need are visitorID and contentID in our prediction files, not the enumerated userID and itemID. Of course, we can do this reverse mapping in post-processing, but can we do it while writing the file instead?

Doing grouping in Pandas may be problematic for large datasets

Proprietary + Confidential

```
import tensorflow as tf  
grouped_by_items = mapped_df.groupby('itemId')
```

Google Cloud

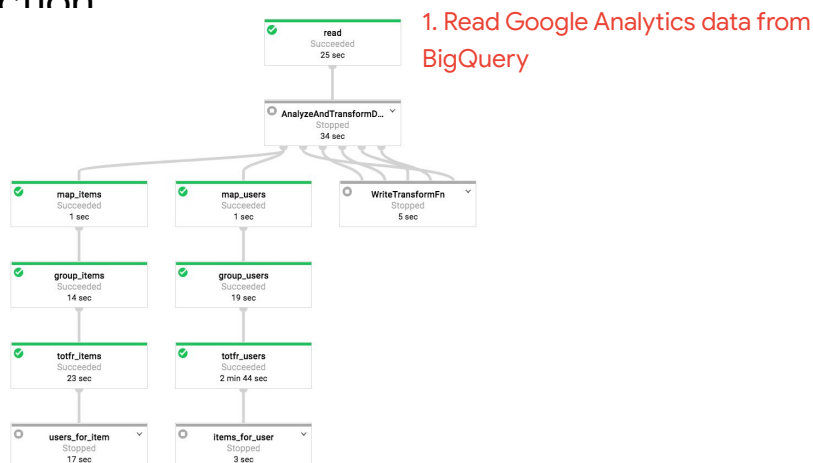
Even though we typically will do training on a periodic basis, on a nightly or hourly snapshot based on some time window, the datasets can be too large to be able to take advantage of using in-memory tools like Pandas, so we need a way to do this and be able to handle the scale.

We should use TensorFlow-Transform to:

1. Create the group-by dataset.
2. Create the vocabulary of visitorId->userId.
3. Use the vocabulary when doing predictions.

We should use TensorFlow Transform to first create the group by dataset, then create the mapping vocabulary of visitorID to userID; then we can use that vocabulary to reverse map when doing predictions. Let's see how that would look in a Dataflow graph.

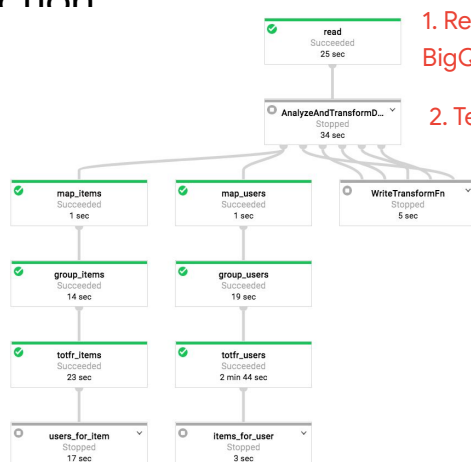
Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction



TensorFlow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction. Here is this example's DataFlow graph. Let's walk through each step.

First, since we were using Google Analytics data from BigQuery, we will read that in.

Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction

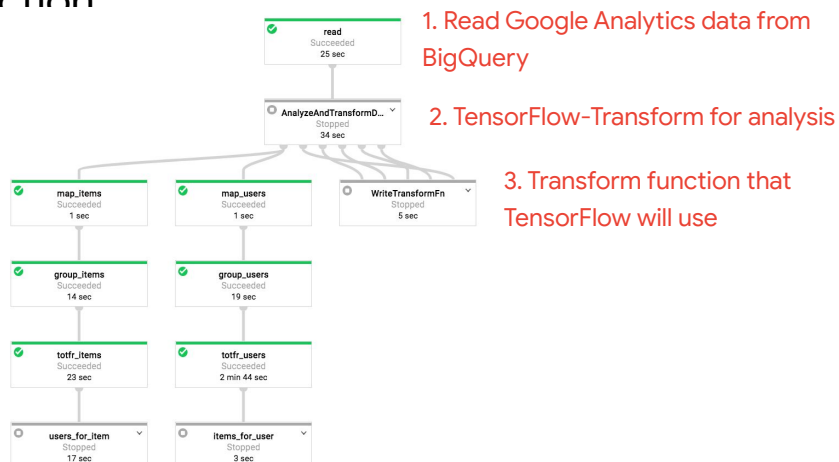


1. Read Google Analytics data from BigQuery

2. TensorFlow-Transform for analysis

Next we'll use TensorFlow Transform for the analysis to get the proper statistics like building the vocabularies.

Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction



Then we'll create the transform function that TensorFlow will use to do the correct mapping.

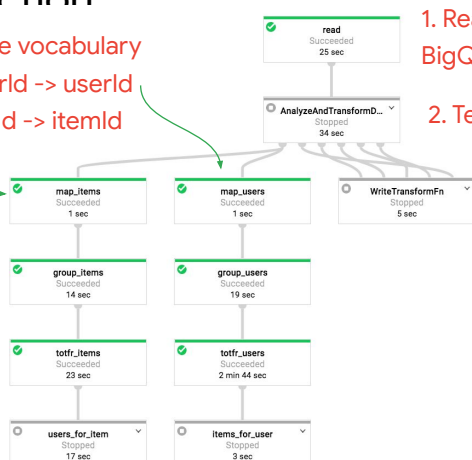
Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction

4. Create vocabulary of visitorId -> userId
contentId -> itemId

1. Read Google Analytics data from BigQuery

2. TensorFlow-Transform for analysis

3. Transform function that TensorFlow will use



This will create the vocabularies of visitorID to userID and contentID to itemID that we can use for the map and reverse map.

Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction

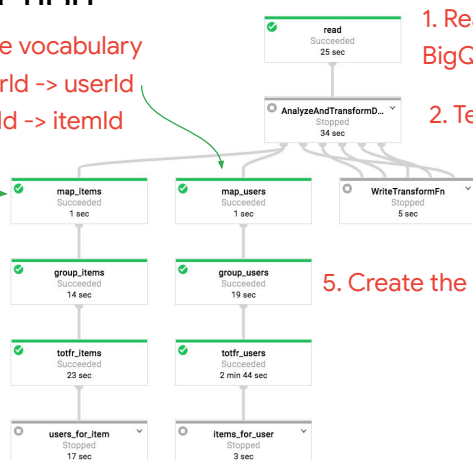
4. Create vocabulary of visitorId -> userId
contentId -> itemId

1. Read Google Analytics data from BigQuery

2. TensorFlow-Transform for analysis

3. Transform function that TensorFlow will use

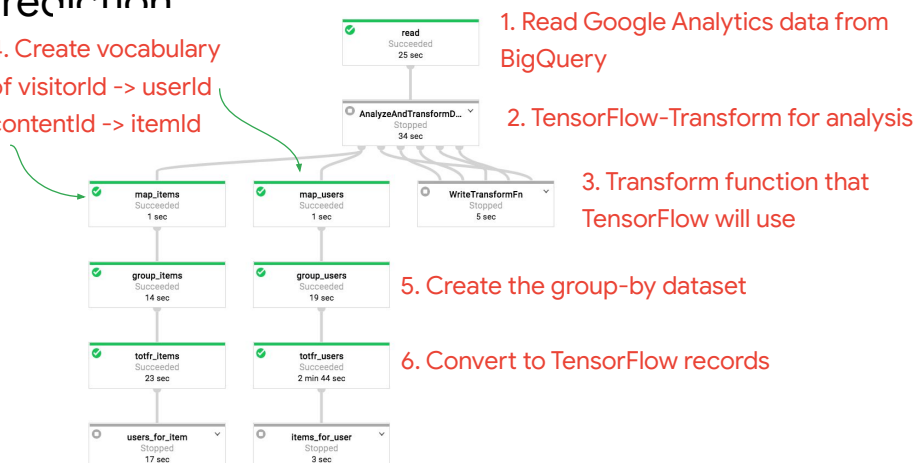
5. Create the group-by dataset



Now that we have the vocabularies, we can create the group-by datasets for both users and items.

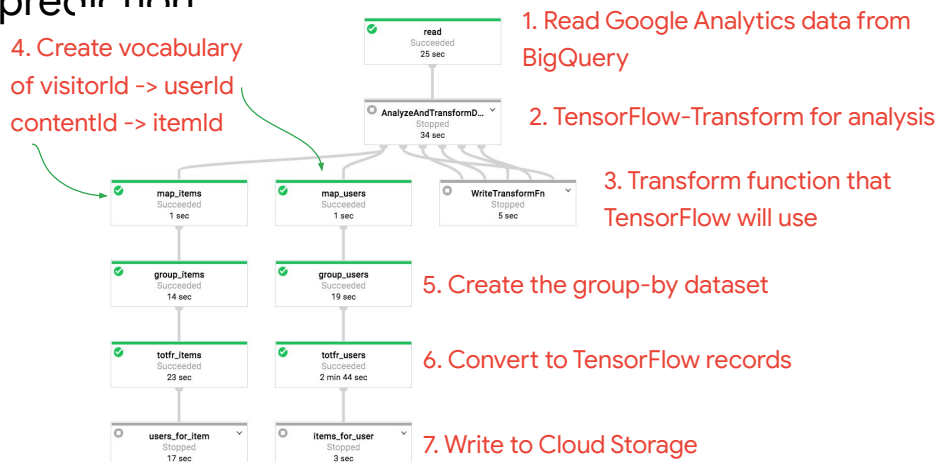
Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction

4. Create vocabulary of visitorId -> userId
contentId -> itemId



Just as before when we were using Pandas, DataFlow will convert our group by datasets into TensorFlow records.

Tensorflow Transform uses Cloud DataFlow in the analysis stage to create assets that TensorFlow uses in training and prediction



Finally, we will write our preprocessed files to Cloud Storage so that they can be used by our TensorFlow model.

The TensorFlow training code remains the same, but we convert back to contentId and visitorId when writing out

```
def create_lookup(filename):
    from tensorflow.python.lib.io import file_io
    dirname = os.path.join(args['input_path'], 'transform-to-transform-pr/assets/')
    with file_io.FileIO(os.path.join(dirname, filename), mode='r') as ifp:
        return [x.rstrip() for x in ifp]

originalItemIds = create_lookup('vocab_items')
originalUserIds = create_lookup('vocab_users')

with tf.Session() as sess:
    ...
    with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
        for userId, best_items_for_user in enumerate(topk.eval()):
            f.write(originalUserIds[userId] + '\t') # write userId \t item1,item2,item3...
            f.write(', '.join(originalItemIds[itemId] for itemId in best_items_for_user) + '\n')
```

Transform function assets are
used to read in the vocabularies

The training code remains the same. Instead of Pandas preprocessing our data, the Dataflow job writes out tfrecords containing itemId and userId. Our batch prediction code takes that, looks it up in the vocabulary, and writes out files that contain contentId and visitorId.

We just need to make a few additions to the batch prediction code, and everything will work perfectly. First we'll define a new function, create_lookup. It uses the transform function assets to read in the vocabularies.

The TensorFlow training code remains the same, but we convert back to contentId and visitorId when writing out

```
def create_lookup(filename):
    from tensorflow.python.lib.io import file_io
    dirname = os.path.join(args['input_path'], 'transform_fn/transform_fn/assets/')
    with file_io.FileIO(os.path.join(dirname, filename), mode='r') as ifp:
        return [x.rstrip() for x in ifp]

originalItemIds = create_lookup('vocab_items')
originalUserIds = create_lookup('vocab_users')

with tf.Session() as sess:
    ...
    with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
        for userId, best_items_for_user in enumerate(topk.eval()):
            f.write(originalUserIds[userId] + '\t') # write userId \t item1,item2,item3...
            f.write(''.join(originalItemIds[itemId] for itemId in best_items_for_user) + '\n')
```

Store original IDs for reverse
mapping

We'll call our create_lookup function for both item and user vocabularies to use for our reverse mapping.

The TensorFlow training code remains the same, but we convert back to contentId and visitorId when writing out

```
def create_lookup(filename):
    from tensorflow.python.lib.io import file_io
    dirname = os.path.join(args['input_path'], 'transform_fn/transform_fn/assets/')
    with file_io.FileIO(os.path.join(dirname, filename), mode='r') as ifp:
        return [x.rstrip() for x in ifp]

originalItemIds = create_lookup('vocab_items')
originalUserIds = create_lookup('vocab_users')

with tf.Session() as sess:
    ...
    with file_io.FileIO(os.path.join(args['output_dir'], 'batch_pred.txt'), mode='w') as f:
        for userId, best_items_for_user in enumerate(topk.eval()):
            f.write(originalUserIds[userId] + '\t') # write userId \t item1,item2,item3...
            f.write(''.join([originalItemIds[itemId] for itemId in best_items_for_user] + '\n'))
```

Look up the vocabulary when
writing out the userId and itemId

Lastly, we'll look up the vocabulary when writing out the userId and itemId. This is easy and doesn't require a dictionary, because we're dealing with indices here and can just return that element based on index location of our vocabulary list.

Quiz

We want a scalable way to generate predictions that directly tie back to the original data and not just enumerated indices. We should use TensorFlow-Transform to first create _____, then create _____, and lastly create _____.

- A. vocabularies, group-by datasets, predictions
- B. vocabularies, predictions, group-by datasets
- C. group-by datasets, vocabularies, predictions
- D. group-by datasets, predictions, vocabularies
- E. predictions, group-by datasets, predictions
- F. predictions, predictions, group-by datasets

Let's test your knowledge! We want a scalable way to generate predictions that directly tie back to the original data, and not just enumerated indices. We should use TensorFlow-Transform to first create BLANK, then create BLANK, and lastly create BLANK. Choose the answer that best fills in the blanks.

Quiz

We want a scalable way to generate predictions that directly tie back to the original data and not just enumerated indices. We should use TensorFlow-Transform to first create _____, then create _____, and lastly create _____.

- A. vocabularies, group-by datasets, predictions
- B. vocabularies, predictions, group-by datasets
- C. group-by datasets, vocabularies, predictions
- D. group-by datasets, predictions, vocabularies
- E. predictions, group-by datasets, predictions
- F. predictions, predictions, group-by datasets

The correct answer is C!

To use the WALSMatrixFactorization estimator, we need to convert from our human-readable user and item identifiers to enumerated indices. However, without a reverse mapping, the predictions will not be in the correct form for us to easily understand them. Therefore, we want a scalable way to generate predictions that directly tie back to the original data, and not just enumerated indices.

Quiz

We want a scalable way to generate predictions that directly tie back to the original data and not just enumerated indices. We should use TensorFlow-Transform to first create _____, then create _____, and lastly create _____.

- A. vocabularies, group-by datasets, predictions
- B. vocabularies, predictions, group-by datasets
- C. group-by datasets, vocabularies, predictions
- D. group-by datasets, predictions, vocabularies
- E. predictions, group-by datasets, predictions
- F. predictions, predictions, group-by datasets

We should use TensorFlow-Transform to first create the group-by datasets, so that each item has a list of each user that interacted with it with their corresponding ratings, and each user has a list of each item that they interacted with, also with the corresponding ratings.

Quiz

We want a scalable way to generate predictions that directly tie back to the original data and not just enumerated indices. We should use TensorFlow-Transform to first create _____, then create _____, and lastly create _____.

- A. vocabularies, group-by datasets, predictions
- B. vocabularies, predictions, group-by datasets
- C. group-by datasets, vocabularies, predictions
- D. group-by datasets, predictions, vocabularies
- E. predictions, group-by datasets, predictions
- F. predictions, predictions, group-by datasets

Then, we create the vocabularies to perform the forward mapping from the grouped-by datasets to enumerated indices, which can then be used in the WALSMatrixFactorization estimator.

Quiz

We want a scalable way to generate predictions that directly tie back to the original data and not just enumerated indices. We should use TensorFlow-Transform to first create _____, then create _____, and lastly create _____.

- A. vocabularies, group-by datasets, predictions
- B. vocabularies, predictions, group-by datasets
- C. group-by datasets, vocabularies, predictions
- D. group-by datasets, predictions, vocabularies
- E. predictions, group-by datasets, predictions
- F. predictions, predictions, group-by datasets

Lastly, we create the predictions, using the TensorFlow-Transform assets to create a lookup to do the reverse mapping from our predicted enumerated indices to our original data formatting.

Demo

Productionized WALS

Ryan Gillard

Google Cloud

Now that we have seen WALS in action, let's now see what a more productionized version of WALS looks like in this demo. In a later module, we'll be building an end-to-end productionized and automated solution using the larger Google Cloud Platform ecosystem.

For this demo, we'll be using the `wals_tft` notebook.

https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/machine_learning/deepdive/10_recommend/wals_tft.ipynb

Collaborative filtering seems powerful...
but it's not without its drawbacks.

Proprietary + Confidential

Google Cloud

So now that we've learned a lot of the details of collaborative filtering, it sure seems powerful. However, it is not without its drawbacks.

Pros:

No domain knowledge

Serendipity Collaborative filtering seems powerful...

Great starting point but is not without its drawbacks.

It's great that no domain knowledge is really required. We are just using the explicit or implicit feedback of the interactions between users and items.

There is also serendipity. Imagine that you are on a movie site and only watch sci-fi movies because you know you like them. Instead of always recommending sci-fi movies to you, which might get a bit repetitive, by looking at what other users who watch sci-fi **ALSO** watch, the site might recommend a couple of fantasy movies or maybe some action movies, because those other users also liked those types of movies.

Collaborative filtering is also a great starting point to get a baseline model and then iterate, add more interaction types, and move on to hybrid recommendation systems.

Pros:

No domain knowledge

Serendipity
Collaborative filtering seems powerful...

Great starting point
but is not without its drawbacks.

Cons:

Fresh items/users?

Context features?

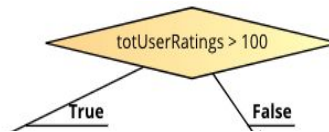
However, collaborative filtering can have a problem when recommending items to new users or recommending users to brand new items. This is the cold start problem. If there isn't interaction data through other users with that item, or other items with that user, it is hard to make similarity measures between them.

Also, basic collaborative filtering only uses interaction data to find latent factors, for users and items within it, that we can use to make recommendations based on similarity. However, we're not able to add contextual features that we might know are important or add our expert knowledge. This would lead us to need hybrid recommendation systems that use all three.

The cold-start problem affects collaborative filtering methods

Proprietary + Confidential

Have to do hybrid of content + collab



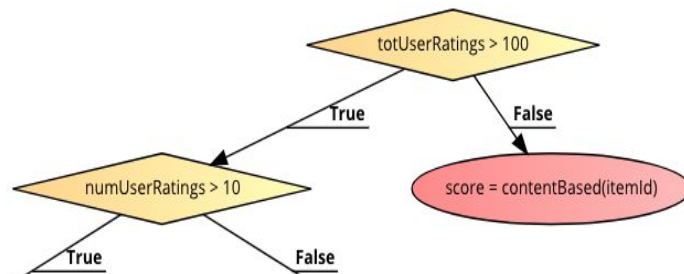
Google Cloud

The cold-start problem of users or items with a low number of, or no, interactions is a major drawback of collaborative filtering. To fix this, we can do a hybrid of content-based and collaborative filtering depending on statistics of the data, as seen in this flowchart.

The cold-start problem affects collaborative filtering methods

Proprietary + Confidential

Have to do hybrid of content + collab



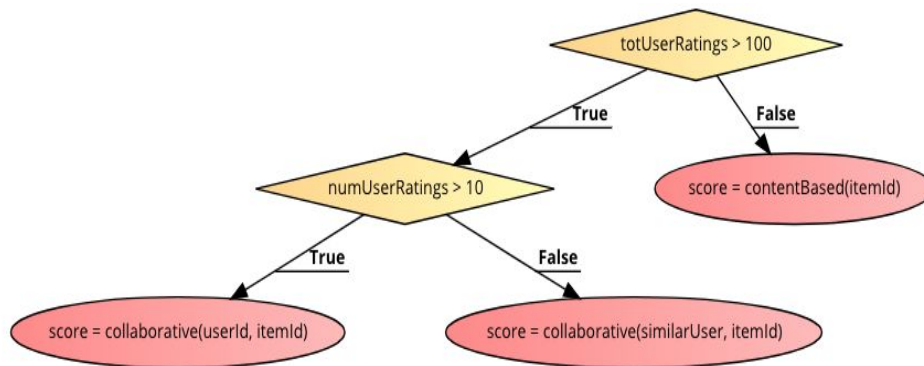
Google Cloud

For instance, if the total number of user ratings is less than or equal to 100, we don't have enough interaction data to make good collaborative filtering recommendations, so we will use content-based recommendations using data about the items that we hopefully do have.

The cold-start problem affects collaborative filtering methods

Proprietary + Confidential

Have to do hybrid of content + collab



Google Cloud

If there are enough total user ratings, we can move on to our next decision point; in this case, "does this current user have more than 10 ratings?" If not, then we should use a user similar to our user for the items that we are trying to score. Otherwise, if our user does have enough ratings, we can use them for these items.

This is a very simplified approach, and we'll see in the next module how we can use neural networks to combine content-based, collaborative filtering, and knowledge-based all into one powerful hybrid model.

The train_and_evaluate loop is typical of tf.contrib Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']) /
args['batch_size'])

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

            export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
                serving_input_fn=create_serving_input_fn(args)
            )
        )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

So far we've seen most of the WALS Matrix Factorization estimator. Let's now take a look at the serving input function in more detail.

Our solution had a serving input function to give back user factors (for all items) given a userId

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack( [users, items], axis=1 )
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

Google Cloud

Our solution had a serving input function to give back user factors for all items given a userId. We read one integer into the userId placeholder.

Our solution had a serving input function to give back user factors (for all items) given a userId

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack([users, items], axis=1)
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

Google Cloud

We then create an enumerated range of all items, which will be the item indices.

Our solution had a serving input function to give back user factors (for all items) given a userId

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack( [users, items], axis=1 )
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

We then tile the passed userId across all of the item indices.

Our solution had a serving input function to give back user factors (for all items) given a userId

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack( [users, items], axis=1 )
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

Google Cloud

We create our ratings tensor with a rating for every item. These are just dummy values and will be ignored. We just need to have the right format for the input SparseTensor.

Our solution had a serving input function to give back user factors (for all items) given a userID

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack([users, items], axis=1)
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

Google Cloud

We stack the users and items tensors along columns so that it will be a rank 2 matrix with our input userID being propagated for every itemID.

Our solution had a serving input function to give back user factors (for all items) given a userId

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack( [users, items], axis=1 )
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

Google Cloud

We then create our SparseTensor, using the rows as our indices and our dummy ratings as our values. Notice that our dense shape is number of users by number of items.

Our solution had a serving input function to give back user factors (for all items) given a `userId`

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack([users, items], axis=1)
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

Google Cloud

Our input rows then is the `SparseTensor` we just made, and the input columns will be ignored. Remember, we are serving predictions here, not training.

Our solution had a serving input function to give back user factors (for all items) given a userId

```
def serving_input_fn():
    feature_ph = {
        'userId': tf.placeholder(tf.int64, 1),
    }
    items = tf.range(args['nitems'], dtype=tf.int64) # all items
    users = feature_ph['userId'] * tf.ones([args['nitems']], dtype=tf.int64)
    ratings = 0.1 * tf.ones_like(users, dtype=tf.float32) # values ignored
    rows = tf.stack( [users, items], axis=1 )
    input_rows = tf.SparseTensor(rows, ratings, (args['nusers'], args['nitems']))
    features = {
        WALSMatrixFactorization.INPUT_ROWS: input_rows,
        WALSMatrixFactorization.INPUT_COLS: ..., # value ignored
        WALSMatrixFactorization.PROJECT_ROW: tf.constant(True)
    }
    return tf.contrib.learn.InputFnOps(features, None, feature_ph)
```

How does the solution
handle giving back
item factors given an
itemId?

Google Cloud

How does our WALS solution handle giving back item factors given an itemId?

It checks whether userId is less than zero. If it is, it uses itemId; otherwise it uses userid. It does this using a conditional, tf.cond. Check out the code again to see for yourself!

The train_and_evaluate loop is typical of tf.contrib Estimators

```
def train_and_evaluate(args):
    train_steps = int(0.5 + (1.0 * args['num_epochs'] * args['nusers']) /
args['batch_size'])

    def experiment_fn(output_dir):
        return tf.contrib.learn.Experiment(
            tf.contrib.factorization.WALSMatrixFactorization(
                num_rows=args['nusers'], num_cols=args['nitems'],
                embedding_dimension=args['n_embeds'],
                model_dir=args['output_dir']),
            train_input_fn=read_dataset(tf.estimator.ModeKeys.TRAIN, args),
            eval_input_fn=read_dataset(tf.estimator.ModeKeys.EVAL, args),
            train_steps=train_steps,
            eval_steps=1,
            min_eval_frequency=steps_in_epoch,

            export_strategies=tf.contrib.learn.utils.saved_model_export_utils.make_export_strategy(
                serving_input_fn=create_serving_input_fn(args))
        )

    learn_runner.run(experiment_fn, args['output_dir'])
```

Google Cloud

And there we have it, we have explored most of the WALS Matrix Factorization estimator. Obviously, there is still a lot left we could talk about such as passing the estimator row and column weight tensors, but for now we already have seen how it can power collaborative filtering recommendation systems.

WALS is thus a way to get user and item embeddings

Proprietary + Confidential

- These embeddings are created solely from user behavior.
- Would be nice to also use knowledge about the item (content-based) and knowledge about the user (knowledge-based).
- How would we combine multiple predictors?

Google Cloud

WALS is thus a way to get user and item embeddings that are trained simultaneously and then can be used for inference.

These embeddings are created solely from user behavior. We didn't create any special features or add expert knowledge. We just used what users interacted with.

However, it would be nice to also use knowledge about the item, like properties of its content, and knowledge about the user, which is knowledge-based.

How would we combine multiple predictors? Well we will see in the next module that we can use a neural network which is a much better solution than the flowchart we saw previously.

Quiz

If we want a recommendation system that uses collaborative filtering, what are some strategies to get around the cold start problem of fresh users/items?

- A. Use averages from the other users/items
- B. Convert to hybrid model, add item info (content-based)
- C. Convert to hybrid model, add user info (knowledge-based)
- D. Use other user-item interaction data
- E. All of the above

Now let's test your knowledge! If we want a recommendation system that uses collaborative filtering, what are some strategies to get around the cold start problem of fresh users and items?

Quiz

If we want a recommendation system that uses collaborative filtering, what are some strategies to get around the cold start problem of fresh users/items?

- A. Use averages from the other users/items.
- B. Convert to hybrid model, add item info (content-based)
- C. Convert to hybrid model, add user info (knowledge-based)
- D. Use other user-item interaction data
- E. All of the above

The correct answer is E!

If we have a fresh user, we could use averages of all of the items to show the most popular, until that user interacts with items and generates some data. If we have a fresh item, we could use averages of all of the users to give the item an estimated rating, until users eventually interact with the item and generate some data. If we have a fresh user and a fresh item, we could use the global average rating between all users and items.

We could also use item information to create a content-based model and/or use user information to create a knowledge-based model, which could be combined with our collaborative filtering model to create a hybrid recommendation system.

Additionally, we could use other user-item interaction data. Many systems have multiple ways users can interact with items, so if one way has missing data, we can use the other slices of data to fill in some of the gaps.