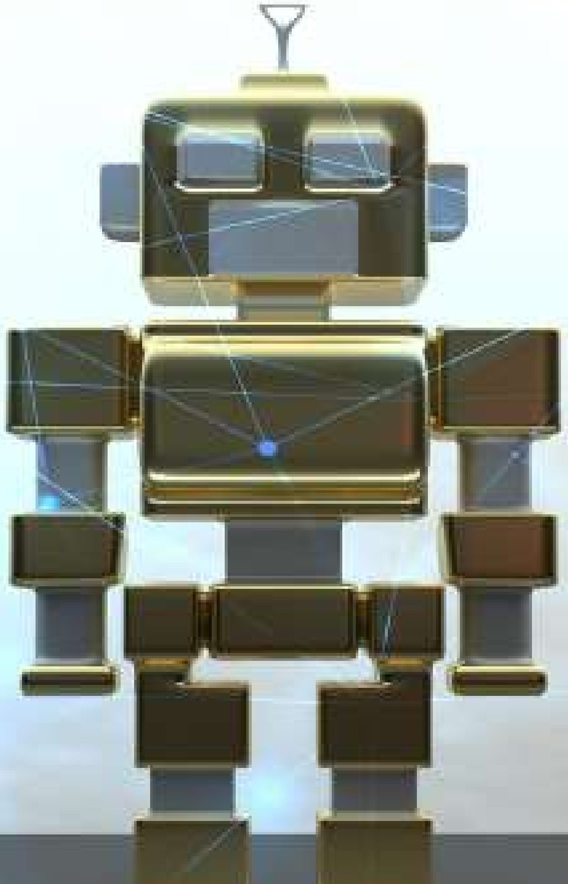
BUILDING RECOMMENDER SYSTEMS

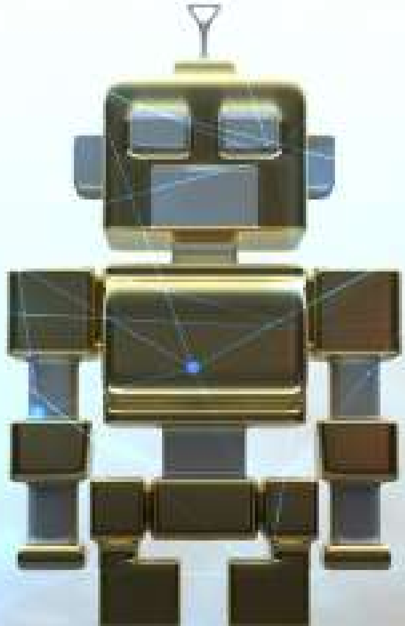
with Machine Learning and AI



FRANK KANE

BUILDING RECOMMENDER SYSTEMS

with Machine Learning and AI



FRANK KANE

Building Recommender Systems with Machine Learning and AI

Frank Kane

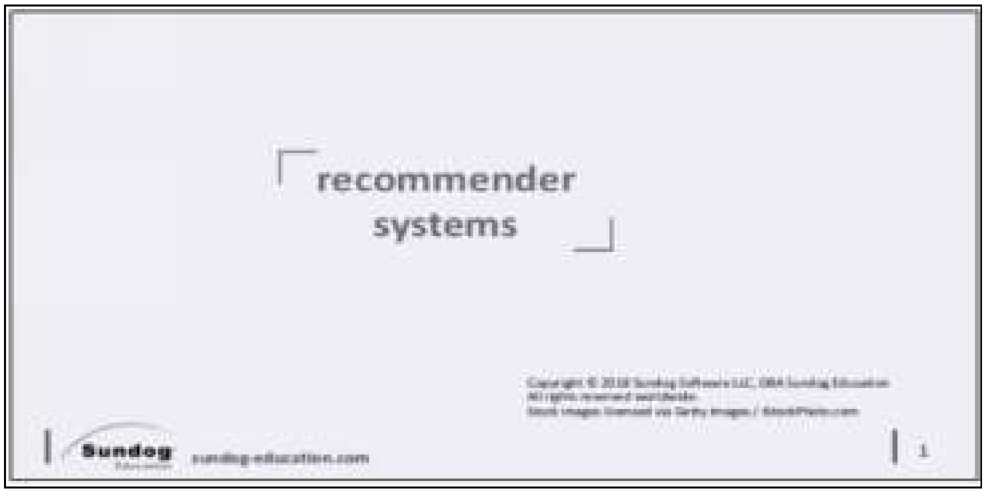
Sundog Education

http://www.sundog-education.com/

Contents

Getting Started

Introduction



recommender
systems

Copyright © 2018 Sundog Software LLC, DBA Sundog Education
All rights reserved worldwide.
Stock images licensed via Getty Images / iStockPhoto.com

Sundog Education sundog-education.com

1

Hello! If you're reading this, you've purchased this course on recommender systems either in PDF or book form. Thank you! If you're also interested in the video version of it, you'll find it at www.sundog-education.com.

I'm Frank Kane, CEO of Sundog Education. I spent nine years at Amazon.com, where I dedicated most of my career to building various parts of their recommendation systems and managing teams responsible for them. You know "people who bought this also bought?" Yeah, I ran that for awhile, along with their "personalization platform" team. I have a lot of real-word experience to share with you, combined with the latest research I've done on new developments in the field since I left Amazon. Recommendations are one of the most fascinating applications of machine learning, and also one of the most lucrative – recommending products is central to Amazon's success, recommending movies is central to YouTube and Netflix, and you can even think of Google as just recommending web pages and ads to people.

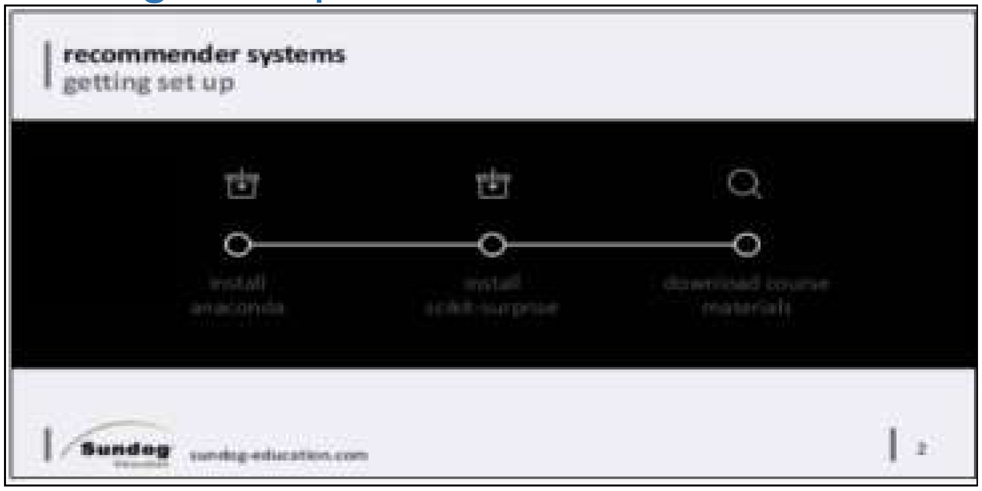
This isn't going to read like a typical book. What you have here are the slides I've used for presenting this information in video or live form, along with the script I prepared to accompany these slides. So for each topic, you'll see an image of the slide associated with it, along with the text I wrote to explain each slide. But, it works surprisingly well – it has all the textual information you'd get in a typical book, but with many more visual aids, and a more casual, conversational tone than you'd find in a textbook. It's ideal for visual learners who just find reading material a lot more efficient than listening to someone read it aloud in a video. I've written this script with this written version of the course in mind, and have attempted to make sure everything works just as well in print as it does in a video presentation.

You'll find that code walk-throughs work a little bit differently in this format than what you may be used to from other technical books. Early in this course, you'll be directed to download all of the code that accompanies it. When we get to slides that review this code, you'll want to pull up that code on your computer as directed in those lectures, and refer to it alongside these written notes that explain what each section of the code does. Really, there's no other practical way to do it – recommender systems involve a fair amount of code, and in most cases it simply won't fit on one written page. I promise to be very specific about what parts of the code I'm talking about as we go through it, so you won't get lost.

Oh, and if Amazon's lawyers are reading this – don't worry. I've been careful to only cover algorithms and techniques that have appeared publicly, in print. I'm not revealing any inside, confidential information here – although most of what we did at Amazon has been published at this point, anyhow .

I know you're itching to go hands-on and produce some recommendations on your own, so let's dive right in and get all the software and data you need installed!

Getting Set Up



recommender systems
getting set up

install anaconda

install scikit-surprise

download course materials

Sundog sundog-education.com

2

In the next few minutes, you're going to install a Python development environment on your PC if you don't have one already. Then, you'll install a package for Python called Surprise that makes developing recommender systems easy. Finally, we'll download the course materials including some real movie rating data, and we'll make movie recommendations for a real person – right here in lecture 1.

let's do this.

Sundog sundog-education.com

3

So, let's do this! The first thing you need is some sort of scientific Python environment that supports Python 3. That means a Python environment that's made for data scientists, like Anaconda or Enthought Canopy. If you already have one, then great – you can skip that step. But if not, let's get Anaconda installed on your system, and we'll also get the course materials you need while we're at it.

sundog-education.com/RecSys

Sundog sundog-education.com

4

Now, if you're the sort of person who prefers to just follow written instructions for things like this, you can head over to my website at the URL shown here. Pull it up anyhow, as we're going to refer to this page as we set things up in this video. Remember to pay attention to capitalization – the "R" and "S" in RecSys need to be capitalized.

You'll also have a chance to join the Facebook group for this course, where you can collaborate with fellow students, and you'll be offered a chance to stay in touch with me as well.

setup walkthrough

Sundog sundog-education.com

5

In this course, we're going to use the Python programming language, as it's pretty easy to pick up. So if you don't already have a development environment for Python 3 installed, you'll need to get one. I recommend Anaconda – it's free and widely used. Let's head to www.anaconda.com/download , and select the installer for whatever operating system you're using. For me, that's Windows 64-bit – and be sure to select the Python 3 version, not Python 2. Once it downloads, go through the installer, making sure to install it on a drive that has plenty of space available – at least 3 GB .

Now that Anaconda is installed, you can launch it…

.. And select the "environments" tab here. To keep things clean, let's set up an environment just for this course. Click on "create" and let's call it "RecSys" – that's shorthand for recommender systems, by the way. We want a Python environment, for whatever current version of Python3 is offered to you. It will take a few moments for that environment to be created.

Next we need to install a Python package that makes developing recommender systems easier, called "Surprise". To do that, click on the arrow next to the RecSys environment you just made, and open up a terminal from it. Now in the terminal, run:

```
 conda install –c conda-forge scikit-surprise
```

If prompted, hit y to continue, and let it do its thing.

When it's done, we can close this terminal window.

Next, we need to download the scripts and data used in this course. From our course setup page at sundog-education.com/RecSys, you'll find a link to the materials. Let's go ahead and download that. When it's done, we'll unzip it, and put it somewhere appropriate, like your documents folder.

Throughout this course, we're going to build up a large project that recommends movies in many different ways. So we're going to need to data to work with – back at our course setup page, you'll find a link to the MovieLens data set. It’s a subset of 100,000 real movie ratings from real people, along with some information about the movies themselves. Download that, and unzip it .

When it’s unzipped, move the resulting ml-latest-small folder inside the course materials folder you made earlier.

Now, we have everything we need! Let’s make some movie recommendations. Back at Anaconda Navigator, make sure the RecSys environment we made is still selected, and now click on the Home icon.

The code editor we’re going to use is called Spyder, so under Spyder hit install, and let that finish.

Once Spyder is done installing, hit launch.

Now, open up the GettingStarted folder inside your course materials, and open the GettingStarted.py script file.

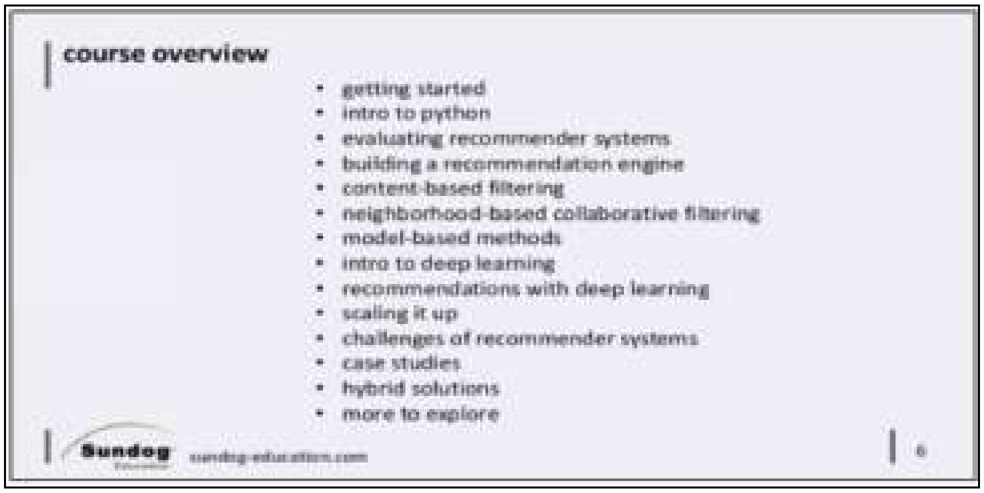
Take a quick look at the code – we’ll come back to it later and walk through it all, but the interesting thing right now is that we’re going to be applying a fairly advanced recommendation technique called Singular Value Decomposition, or SVD for short, on 100,000 movie ratings with just about 60 lines of code. How cool is that? This isn’t necessarily hard from a coding standpoint.

Hit the green “play” button, and that will kick off the script. It starts by choosing an arbitrary user, user number 85 in our case – whom we’re going to get to know very well - and summarizing the movies he loved, and the movies he hated, so you can get a sense of his tastes. He seems pretty typical – someone who likes good action and sci-fi movies, and hates movies that really miss the mark, like Super Mario Brothers. If you haven’t seen that movie, it’s pretty painful, especially if you’re a Nintendo fan! Next we run the SVD algorithm on our movie ratings, and use the ratings from everyone else to try and find other movies user 85 might like that he hasn’t already seen !

It came up with some interesting stuff – for movies you've heard of, like Gladiator, that recommendation makes intuitive sense based on what we know about this user. But it's hard to tell if a movie you've never heard of is a good recommendation or not – indeed, you'll learn in this course that just defining what makes a good recommendation is a huge problem that's really central to the field of recommender systems. And in many ways, building recommender systems is more of an art than a science – you're trying to get inside peoples' heads, and build models of their preferences. It's a very hard problem, but also a very fun one to solve!

Anyways, congratulations! You just ran your first movie recommendation system using some real movie rating data, from real people! There's so much more for us to dive into here, so keep going! This is a really interesting field, and even after spending over 10 years on it myself, it just doesn't get old.

Course Overview



course overview

- getting started
- intro to python
- evaluating recommender systems
- building a recommendation engine
- content-based filtering
- neighborhood-based collaborative filtering
- model-based methods
- intro to deep learning
- recommendations with deep learning
- scaling it up
- challenges of recommender systems
- case studies
- hybrid solutions
- more to explore

Sundog sundog-education.com 6

This is a pretty big course, so it's worth setting the stage about how all the different parts of it fit together.

To get started, we talk about the different kinds of recommender systems, the problems they try to solve, and the general architecture they tend to follow.

We'll then have a quick introduction to Python for people who have some programming experience, but might be new to the Python language.

The next topic is evaluating recommender systems; defining what makes a "good" recommendation is in itself a complicated question, and it's important to decide what you're optimizing for .

We'll then build a software framework for generating and evaluating recommendations using real data, and using different recommendation algorithms. Throughout the rest of the course, we'll use this framework to quickly implement and evaluate new ideas.

"Content-based filtering" will show us how we can recommend items to people just based on the attributes of the items themselves. "Neighborhood-based collaborative filtering" gets into the classic behavior-based recommendation approach pioneered by Amazon that’s still in widespread use today.

We then move into more modern, model-based methods that generally rely on matrix factorization approaches to generate recommendations.

Then, it’s time to bust out the neural networks. If you’re new to deep learning, we have a section that will get you up to speed on modern neural networks. And then we’ll go through several examples of using deep learning to generate recommendations, and the challenges that are specific to recommender systems when using deep learning.

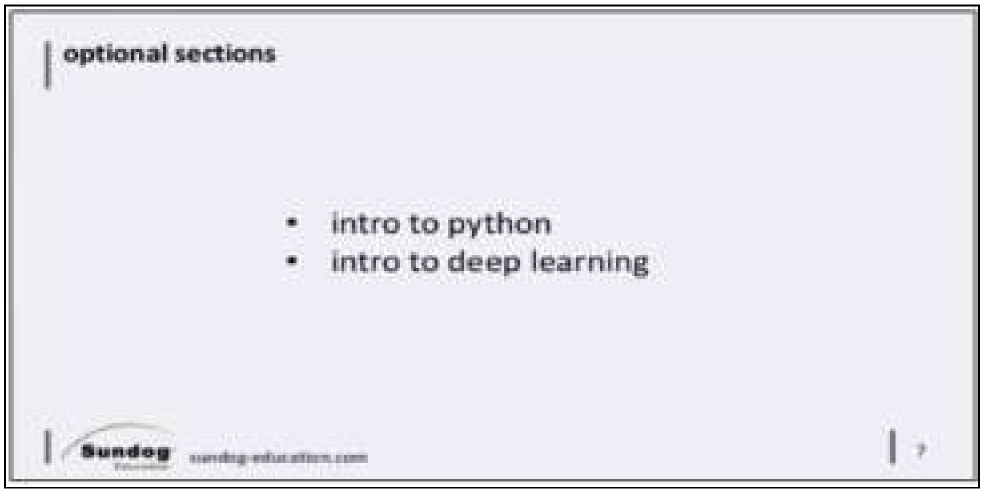
Next, we’ll scale things up to massive data sets. We’ll learn how to use Amazon Web Services, Apache Spark, and some exciting new technology open-sourced by Amazon to generate recommendations for millions of people across catalogs of millions of items.

We’ll then discuss some of the real-world challenges you’ll encounter when deploying recommender systems for real items, to real people – and some of the solutions that work for them.

We continue to keep it real by looking at case studies of how recommender systems work at YouTube and Netflix, which ties together many of the concepts covered earlier in the course .

Finally, we’ll look at hybrid solutions that allow us to combine many different recommender systems together into one that’s even more powerful, and we’ll wrap things up with tips on how you can continue your education and research in recommender systems as the field continues to evolve over time.

This is a very comprehensive course, covering some of the latest research and practical implementation tips in the field. And throughout the course, you’ll have hands-on examples all along the way to see what you’ve learned in action, and individual exercises to allow you to practice what you’ve learned on your own.



optional sections

- intro to python
- intro to deep learning

Sundog sundog-education.com 7

A quick note about how to use this course, depending on your previous experience. This course is intended to be usable by anybody with some background in computer science, but not necessarily people who already know Python specifically or who have prior experience with neural networks and deep learning .

If you already have experience in these fields, it’s OK to skip these sections of the course. Nothing happens in the “intro to Python” or “intro to deep learning” sections that later sections build upon. If you already know Python, you can skip that whole section and proceed through the rest of the course just fine. And if you are already familiar with neural networks, Tensorflow, and recurrent neural networks, you can skip the “intro the deep learning” section – or at least the portions of it you’re already familiar with.

Overview of Recommender Systems

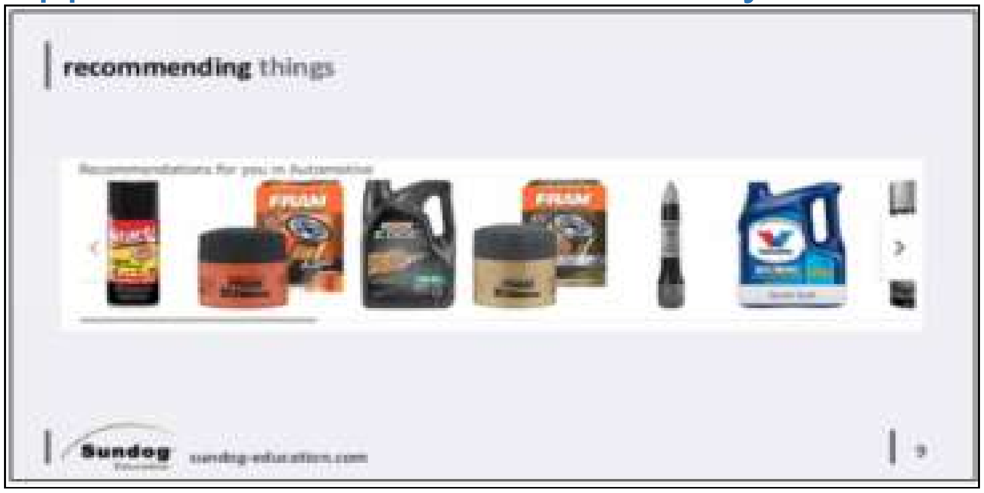
many flavors of recommenders

Sundog

sundog-education.com

Before we start diving into code and algorithms, let's talk a little about recommender systems at a high level. You encounter them every day, and might not even think about it. They can take many different forms, and serve many different purposes.

Applications of Recommender Systems


recommending things

Recommendations for you in Automotive

Sundog sundog-education.com 9

My own claim to fame is recommending things – I worked on Amazon's early recommender systems, which look at your past purchasing behavior to recommend other products you might like. For example, I recently bought a gas-powered generator, and Amazon is automatically recommending things like starter fluid and motor oil to me. You'd be amazed how effective these recommendations can be – billions of dollars have resulted from them, and they've helped people find the things they need. The beauty of it is that it's all data-driven – recommender systems find relationships between users and between items just based on actions – usually, there's no human curation involved at all. It knows that statistically, people who buy generators also buy starter fluid – and it can use those historical patterns to show people stuff they want before they even know they want it.

But you don't have to limit yourself to recommending things – you can also recommend content. Here's a recommender system used by the New York Times, which is a popular newspaper in the US. It looks at articles you've read in the past, to recommend other articles you might enjoy reading. Same idea, just looking at patterns in the articles people read, instead of patterns in the stuff people buy.

Music recommendations is its own special case. Sure, you can treat it just like any other content, and recommend music that other people have listened to who share your tastes. But services like Pandora take it a step further, and analyze the waveforms of the music itself to find similarities between tempos, musical styles, and song structures. This is an example of content-based recommendations, where your recommendations aren't just based on user behavior, but on the properties of the things you're recommending themselves – in this case, the musical properties of the songs you like. Look up Pandora's "music genome project" if you're interested in more depth on that.

recommending
people

Sundog sundog-education.com 12

Why stop at products, content, and music – you can even recommend people! That's basically what online dating websites do – they are recommender systems as well. As a married man though, it's not really my area of expertise!

recommending
search results

Sundog sundog-education.com 13

There's even a fine line between search engines and recommender systems. Modern search results tend to be personalized; it's not just doing information retrieval, it's looking at your past behavior as an individual to figure out what search results are most relevant to you. Think of it as recommending web pages, just like you would recommend books, music, or newspaper articles. In this example, I typed in the name of my favorite sushi restaurant – and although that name means many different things to different people, Google knows where I live, where I've been, and the websites I've visited in the past – and it can use all of that information to pull up results about the restaurant down the street from me, instead of pages about Yuki Hana's Japanese translation of "snow flower."

So you see, recommender systems are everywhere, and they are responsible for a huge portion of the modern economy. It's not something you hear about often, but people who know how to build them are in very high demand because recommender systems can have a direct influence on a company's sales. Whether you're trying to recommend things, articles, music, movies, people, or web pages – recommender systems are the technology behind it.

Gathering Interest Data

understanding you

Sundog sundog-education.com

14

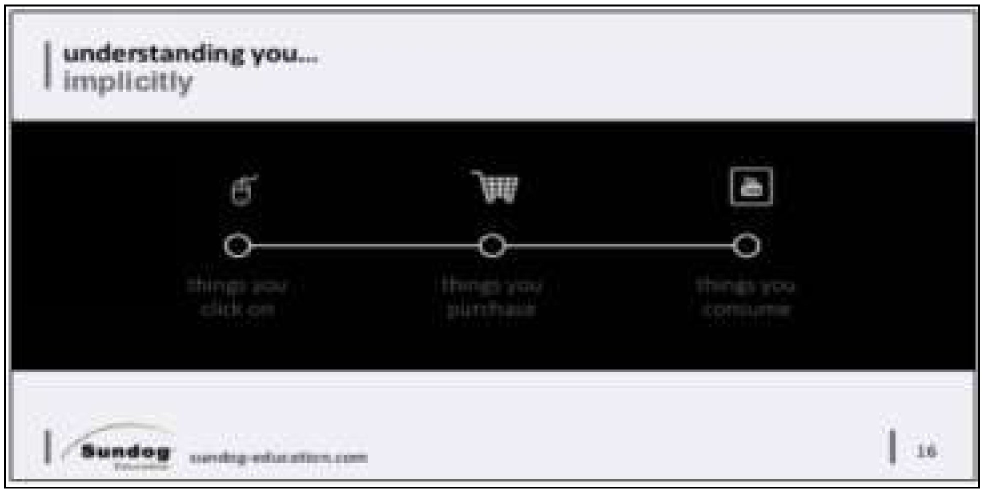
So how do recommender systems work? Well, it comes down to understanding you. Not just you, but every customer or visitor to a website or maybe even a network of websites that share data with each other. A recommender system starts with some sort of data about every user that it can use to figure out that user's individual tastes and interests. Then, it can merge its data about you with the collective behavior of everyone else like you to recommend stuff you might like. But where does that data about your unique interests come from?

understanding you...
explicitly

Sundog sundog-education.com

15

One way to understand your users or customers is through explicit feedback. For example, asking users to rate an online course like this one on a scale of one to five stars. Or, rating content they see with a “like” or a “thumbs up” or “thumbs down.” In these cases, you’re explicitly asking your users “do you like this thing you’re looking at”, and you use that data to build up a profile of that user’s interests. The problem with explicit ratings or feedback is that it requires extra work from your users – not everyone can be bothered to leave a rating on everything they see for you, so this data tends to be very sparse. When your data is too sparse, it leads to low-quality recommendations. Another problem with explicit ratings is that everyone has different standards, so the meaning of a four-star review might be different between two different people – some people might just be more critical than others. I’ve even noticed differences in how people rate my courses between different countries, so there can even be cultural differences in how people rate things. There are ways to compensate for that, though, that we’ll talk about.



understanding you...
implicitly

things you click on
things you purchase
things you consume

Sundog sundog-education.com 16

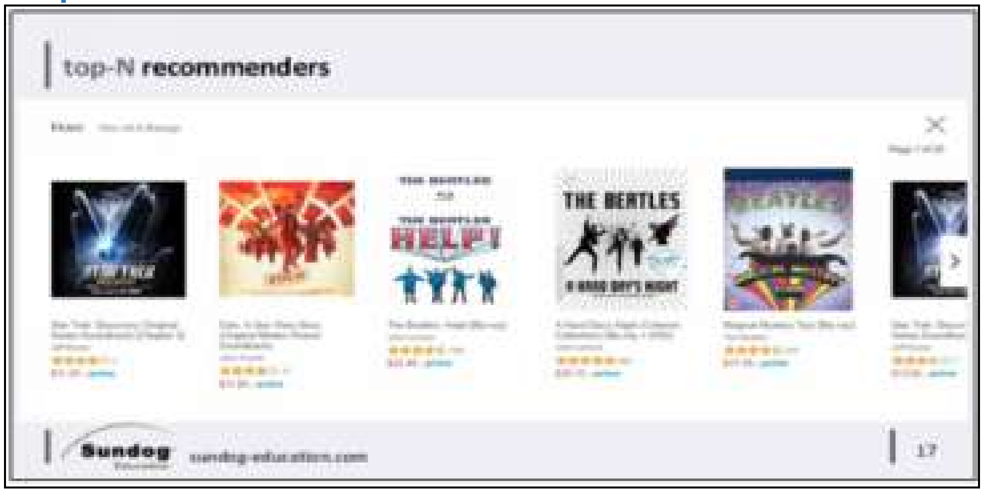
Another way to understand your individual tastes is through your implicit behavior. This is looking at the things you do anyhow, and interpreting them as indications of interest or disinterest. For example, if you click on a link on a web page, we could consider that an implicit positive rating for that link and the content it points to. Or, if you click on an ad, it might tell the ad network that you might find other ads similar to that ad appealing. Click data is great because there is so much of it, so you don’t have problems with data sparsity. But clicks aren’t always a reliable indication of interest. People often click on things by accident, or they might be lured into clicking on something just because some sexy image is associated with it. I've learned the hard way that building recommender systems that use click data alone just end up being pornography detection engines! Click data is also highly susceptible to fraud, because there are a lot of bots out there on the Internet doing nefarious things that can pollute your data.

Things you purchase are a much better indication of interest, because actually opening up your wallet and spending your hard-earned money is a sure sign that you're interested in something. Well, unless it's a gift, but that's its own problem. Using purchases as implicit positive ratings is also great because it's very resistant to fraud – someone trying to manipulate a recommender system based on purchase behavior will find it to be prohibitively expensive, because they'd have to buy a lot of stuff to affect its results. A big part of why Amazon's recommendations are so good is just because they have so much purchase data to work with, and it's such a great signal for someone's interests. They don't even need great recommendation algorithms, because the data they're working with is so great to begin with.

Another implicit source of interest data is things you consume. For example, YouTube can look at how many minutes you spent watching a video as an indication of how much you liked it. Consumption data doesn't require consumption of your money like purchase data does, but it does require a consumption of your time – so it's also a pretty reliable indicator of interest compared to click data. That's why YouTube uses minutes watched heavily in its recommendation algorithms, and why YouTubers try as hard as they can to get you to actually watch their entire video all the way through.

So, in sum – explicit ratings are great if you can convince your users to give them to you. And, that's what we're dealing with in this course with the MovieLens data set we're working with – explicit movie ratings left by people who rated movies from one to five stars. But **implicit** data can give you much more data to work with, and in the case of purchase data, it might even be better data to start with. When designing a recommender system, it's important to think about what sources of data you have about your user's interests, and start from there – because as you'll see, even the best recommender system can't produce good results unless it has good data to work with, and lots of it .

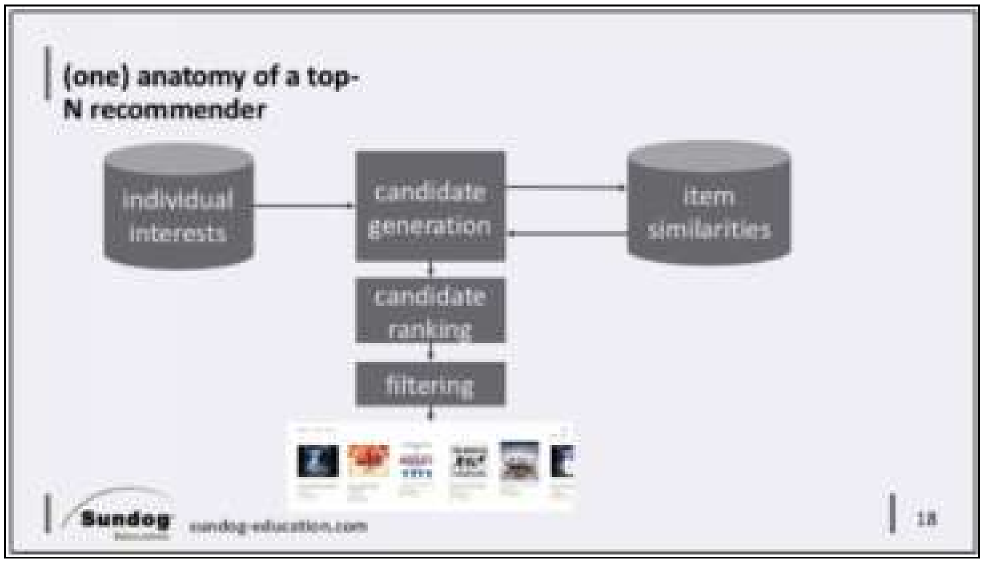
Top-N Recommenders



Let's get some terminology out of the way – all of the recommender systems we've looked at so far are what's called "Top-N" recommender systems. That means that their job is to produce a finite list of the best things to present to a given person. Here's a shot of my music recommendations on Amazon, and you'll see it's made up of 20 pages of five results per page – so this is a top-N recommender, where N is 100.

As you'll soon see, a lot of recommender system research tends to focus on the problem of predicting a user's ratings for everything they haven't rated already, good or bad, but that's very different from what recommender systems need to do in the real world. Usually, users don't care about your ability to predict how they'll rate some new item. That's why the ratings you see in this widget are the aggregate ratings from other users, and not the ratings the system thinks you'll give them. Customers don't want to see your ability to predict their rating for an item – they just want to see things they're likely to love !

Throughout this course, it's important to remember that ultimately our goal is to put the best content we can find in front of users in the form of a top-N list like this one. Our success depends on our ability to find the best top recommendations for people -so it makes sense to focus on finding things people will love, and not our ability to predict the items people will hate. It sounds obvious, but this point is missed by a lot of researchers. Those top 5 or 10 recommendations for each user are what really matters.



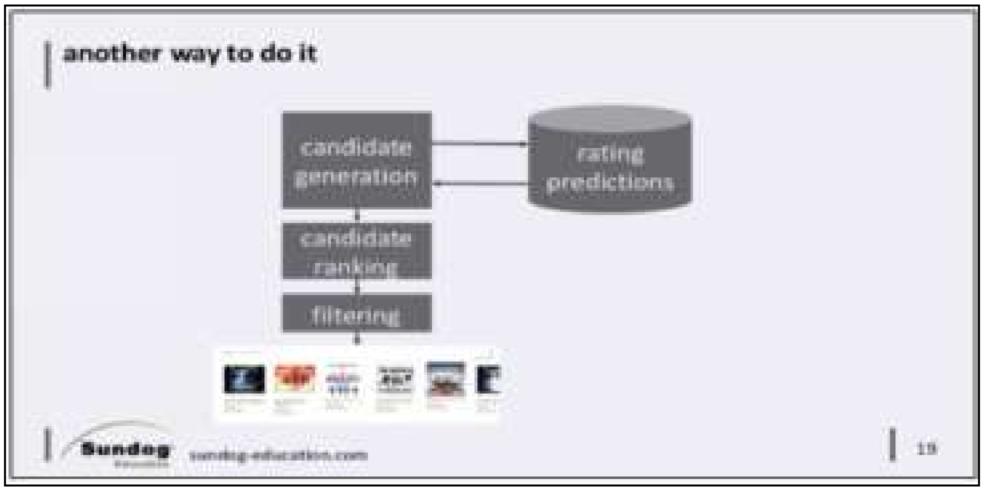
Here's one way a top-N recommender system might work – and there many ways to do it. But generally, you start with some data store representing the individual interests of each user, for example their ratings for movies they've seen, or implicit ratings such as the stuff they've bought in the past. In practice this is usually a big, distributed "NoSQL" data store like Cassandra or MongoDB or memcache or something, because it has to vend lots of data but with very simple queries. Ideally, this interest data is normalized using techniques such as mean centering or z scores to ensure that the data is comparable between users, but in the real world your data is often too sparse to normalize it effectively.

The first step is to generate recommendation candidates – items we think might be interesting to the user based on their past behavior. So the candidate generation phase might take all of the items a user indicated interest in before, and consult another data store of items that are similar to those items based on aggregate behavior. Let’s take an example – let’s say you’re making recommendations for me. You might consult my database of individual interests and see that I’ve liked Star Trek stuff in the past. Based on everyone else’s behavior, I know that people who like Star Trek also like Star Wars. So, based on my interest in Star Trek, I might get some recommendation candidates that include Star Wars stuff. In the process of building up those recommendations, I might assign scores to each candidate based on how I rated the items they came from, and how strong the similarities are between the item and the candidates that came from them. I might even filter out candidates at this stage if their score isn’t high enough.

Next, we move to candidate ranking. Many candidates will appear more than once, and need to be combined together in some way – maybe boosting their score in the process since they keep coming up repeatedly. After that, it can just be a matter of sorting the resulting recommendation candidates by score to get our first cut at a “top-N” list of recommendations – although much more complicated approaches exist, such as “learning to rank” where machine learning is employed to find the optimal ranking of candidates at this stage. This ranking stage might also have access to more information about the recommendation candidates that it can use, such as average review scores, that can be used to boost results for highly-rated or popular items for example .

Some filtering will be required before presenting the final, sorted list of recommendation candidates to the user. This filtering stage is where we might eliminate recommendations for items the user has already rated, since we don’t want to recommend things the user has already seen. We might also apply a stop-list here to remove items that are potentially offensive to the user, or remove items that are below some minimum quality score or minimum rating threshold. It’s also where we apply the “N” in top-N recommenders, and cut things off if we have more results than we need. The output of the filtering stage is then handed off to your display layer, where a pretty widget of product recommendations is presented to the user. Generally speaking, the candidate generation, ranking, and filtering will live inside some distributed recommendation web service that your web front-end talks to in the process of rendering a page for a specific user.

This diagram is a simplified version of what we call “item-based collaborative filtering,” and it’s the same algorithm Amazon published in 2003. You can see it’s not really that complicated from an architecture standpoint; the hard part is building up that database of item similarities, really – that’s where the magic happens. Again, this is just one way to do it – we’re going to explore many, many others throughout the course!


another way to do it

Another architecture that’s popular with researchers, and that we’ll see in this course, is to build up a database ahead of time of predicted ratings of every item by every user. The candidate generation phase is then just retrieving all of the rating predictions for a given user for every item, and ranking is just a matter of sorting them.

This requires you to look at every single item in your catalog for every single user, however, which isn’t very efficient at runtime. In the previous slide, we only started with items the user actually liked, and worked from there – instead of looking at every item that exists.

The reason we see this sort of architecture is because people like to measure themselves on how accurately they can predict ratings, good or bad. But as we’ll see, that’s really not the right thing to focus on in the real world. If you have a small catalog of items to recommend, however, this approach isn't entirely unreasonable.

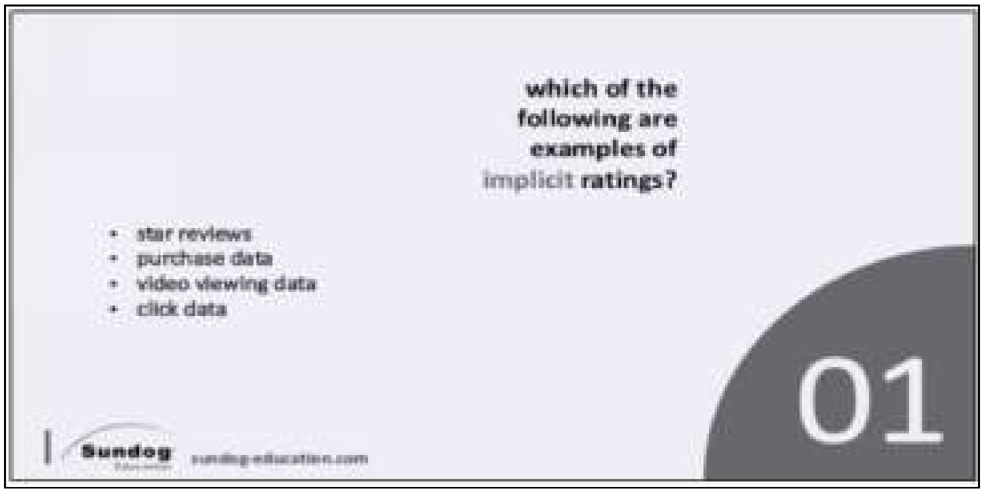
Quiz

quiz time

Sundog sundog-education.com

20

We're not at the point yet where I can turn you loose to start practicing writing your own code for recommender systems, but we **will** get there soon. Meanwhile, let's do a short quiz to reinforce the things we've talked about so far.

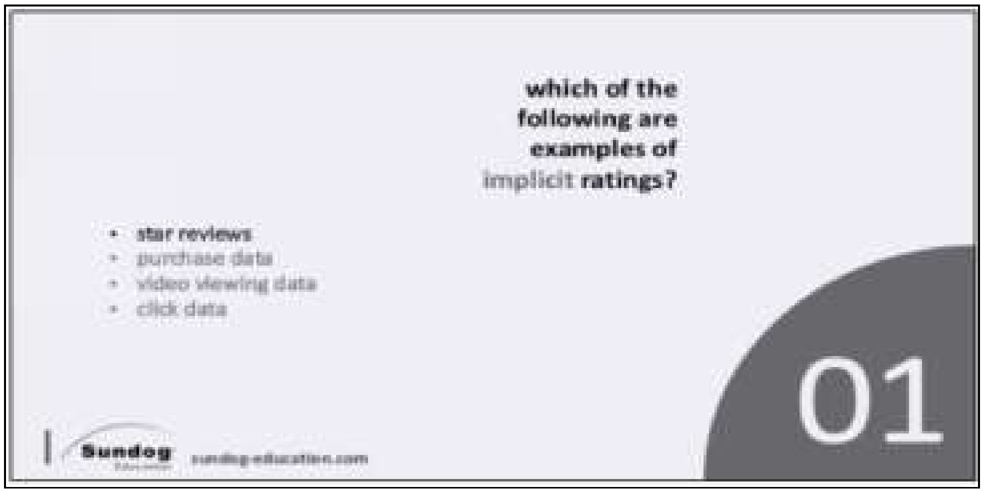

which of the following are examples of implicit ratings?

- star reviews
- purchase data
- video viewing data
- click data

01

Sundog sundog-education.com

We talked about the difference between implicit and explicit indications of interest. Which of these are implicit ratings, as opposed to explicit? Star reviews, like when you're rating a movie – purchase data, like when you buy something on Amazon – video viewing data, like how many minutes of a YouTube video you watch – and click data, like the record of online ads you’ve clicked on from an ad network.



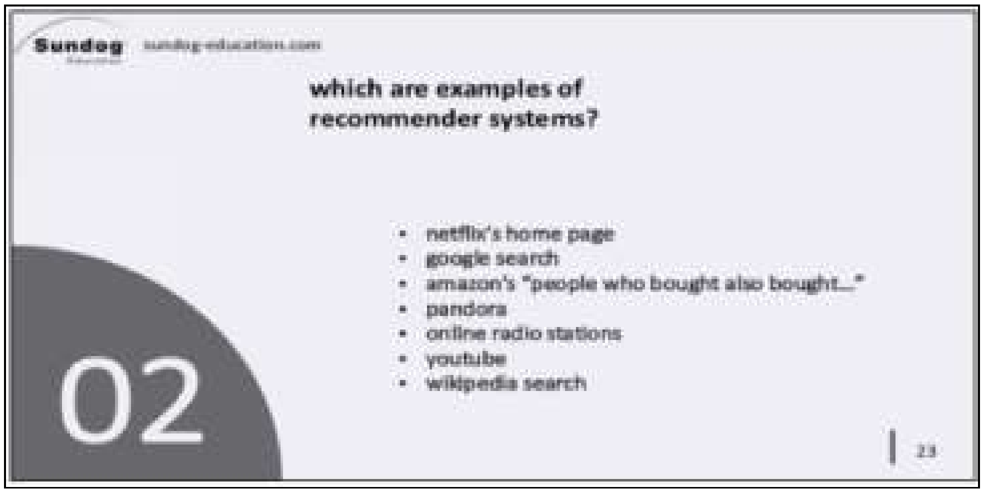
which of the following are examples of implicit ratings?

- star reviews
- purchase data
- video viewing data
- click data

01

Sundog Education sundog-education.com

The only example of explicit ratings here is star reviews, where you are explicitly asking a user for their opinion of something. Purchasing, viewing, and clicking are all implicit ratings – we can infer you like something because you chose to interact with it somehow. While explicit ratings provide very good data for recommender systems, they require extra work from your users, so it's hard to get enough of them to work with. Implicit data however, is just a byproduct of your users' natural behavior – so it's easy to get lots and lots of it. For a good recommender system, you need both quality and quantity of interest data.


Sundog sundog-education.com

which are examples of recommender systems?

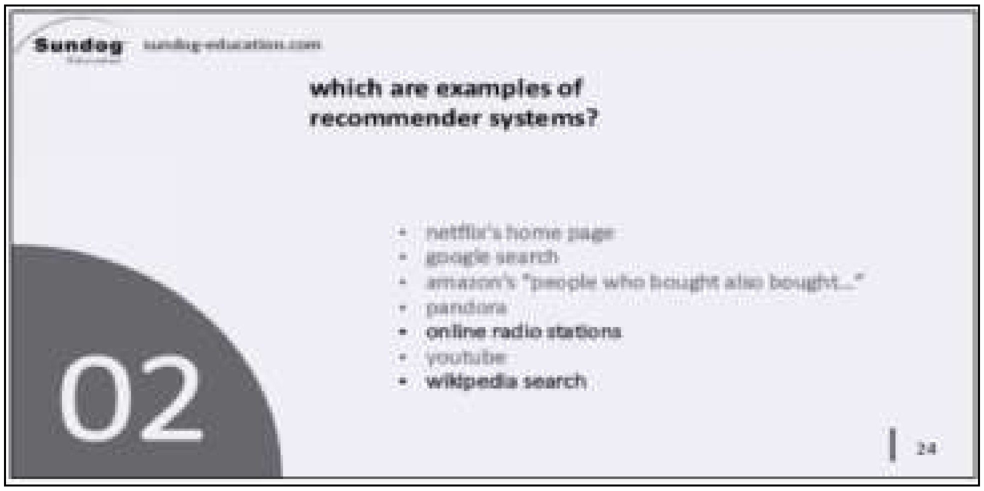
- netflix's home page
- google search
- amazon's "people who bought also bought..."
- pandora
- online radio stations
- youtube
- wikipedia search

02

23

Which of the following are examples of recommender systems? Some of these are a little tricky, but ask yourself which of these experiences are personalized to your individual past expressions of interest, be they explicit or implicit.

We have the movie recommendation widgets you might see on Netflix's home page, the search results you get when using Google, Amazon's "people who bought this also bought" widget when you view a product, Pandora's music streaming system, online radio stations where you might be listening to a stream of an over-the-air radio station, YouTube, and the search results you get from Wikipedia.


Sundog sundog-education.com

which are examples of recommender systems?

- netflix's home page
- google search
- amazon's "people who bought also bought..."
- pandora
- online radio stations
- youtube
- wikipedia search

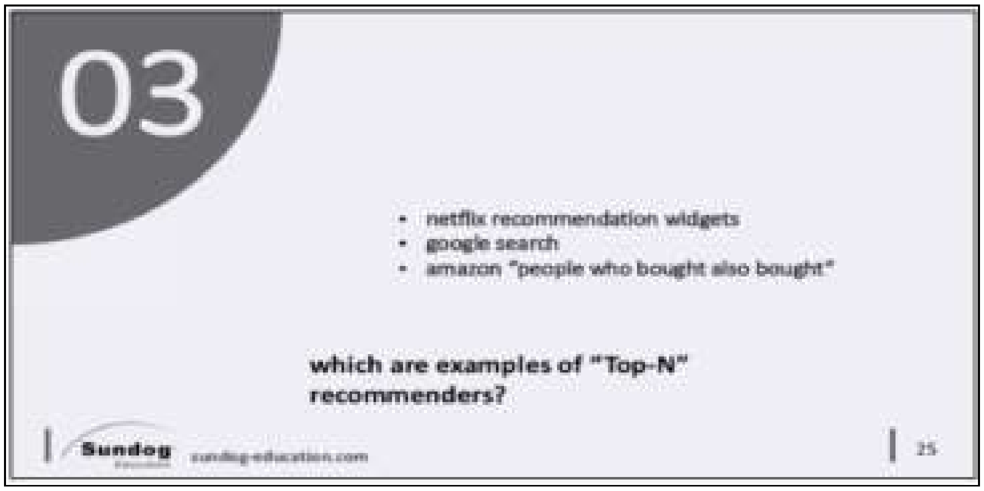
02

24

Netflix definitely takes your past viewing and rating behavior into account when recommending things for you to watch, as does

YouTube. Your Google search results are personalized based on what Google knows about you, so it too counts as a recommender system – but Wikipedia gives the same search results to everyone, so it's really just an information retrieval system, and not a recommender system. We know that Pandora chooses songs for you based on your explicit ratings, so it's a recommender system – but if you listen to your local radio station online, that's not a personalized experience, so it doesn't involve recommendation technology. Amazon's "people who bought also bought" also counts as a recommender system, because even though it doesn't involve explicit ratings, it does involve the implicit indication of interest you gave to the item this feature appears on, just by viewing it.

As you can see, there are gray areas here – but the important point is that what you learn about building recommender systems can be applied to a very wide variety of problems in the real world. And even for the items we didn't select as "correct answers," you could imagine building a recommender system for which online radio station you should listen to – and maybe personalizing Wikipedia search with recommender systems isn't so bad of an idea.

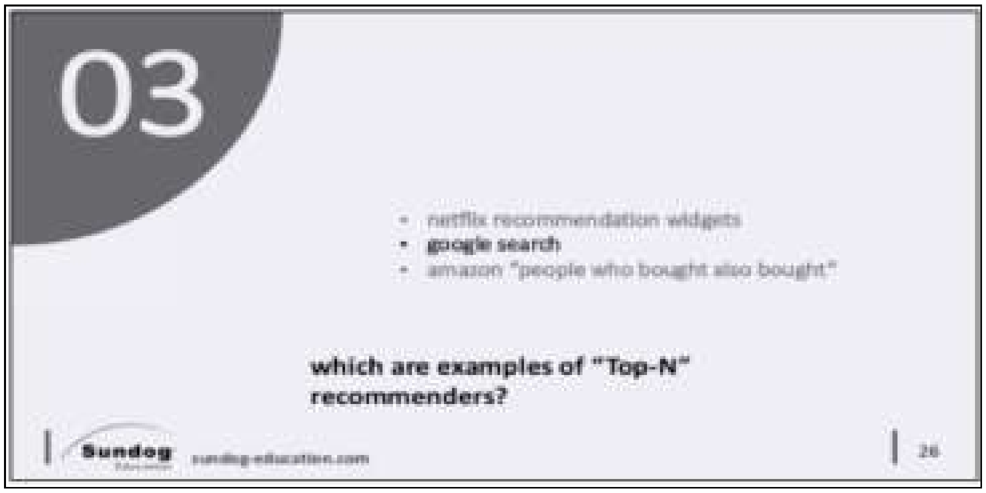

03

- netflix recommendation widgets
- google search
- amazon "people who bought also bought"

which are examples of "Top-N" recommenders?

Sundog sundog-education.com 25

Which of these are examples of "top-N" recommender systems, where our aim is just to produce a short list of the best recommendations we can come up with for a given user?



03

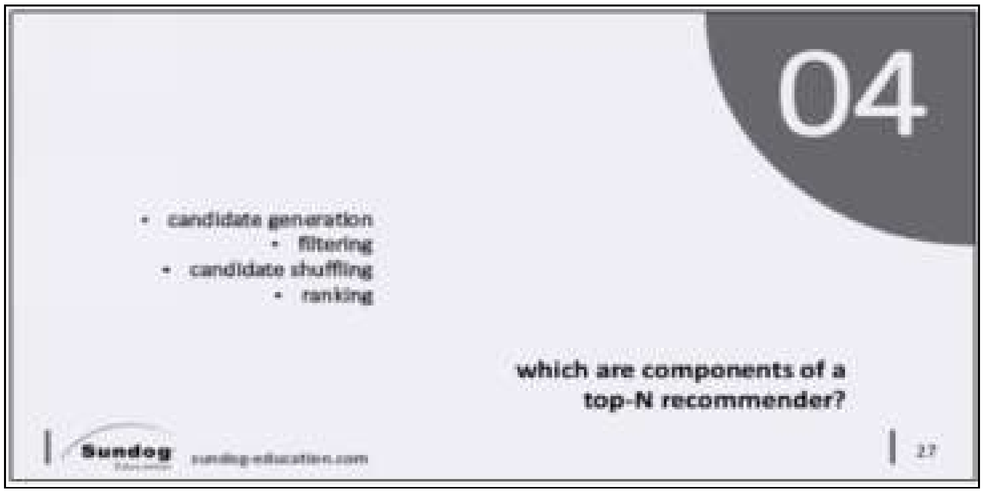
- netflix recommendation widgets
- google search
- amazon "people who bought also bought"

which are examples of "Top-N" recommenders?

Sundog sundog-education.com 26

Recommendation widgets for Netflix or Amazon generally top out at 20 or so results, and so they qualify as "top N" recommender systems. Google however can return as many results as you would like, so it needs to rank many more pages for you.

Although, if you restricted Google to only one or two pages of personalized search results, you could also frame it as a top-N recommender. So again, it's a bit of a gray area – but we use the idea of "top-N" recommenders just to focus our work on what matters, and in the case of top-N, it means we want to focus just on the top results for each user, and not our ability to predict ratings on everything for everybody.



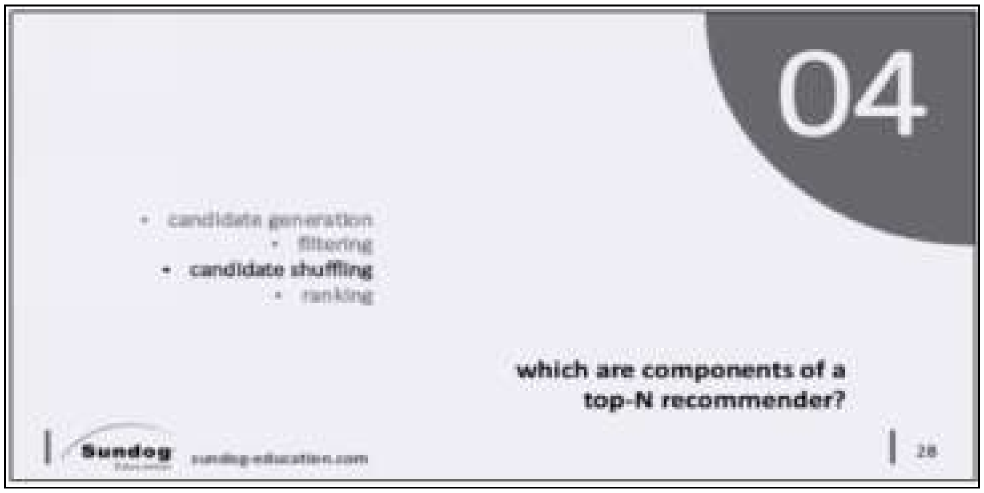
04

- candidate generation
- filtering
- candidate shuffling
- ranking

which are components of a top-N recommender?

Sundog sundog-education.com 27

Thinking back to our architecture slide for top-N recommender systems, which of the following were boxes in that diagram?



04

- candidate generation
- filtering
- candidate shuffling
- ranking

which are components of a top-N recommender?

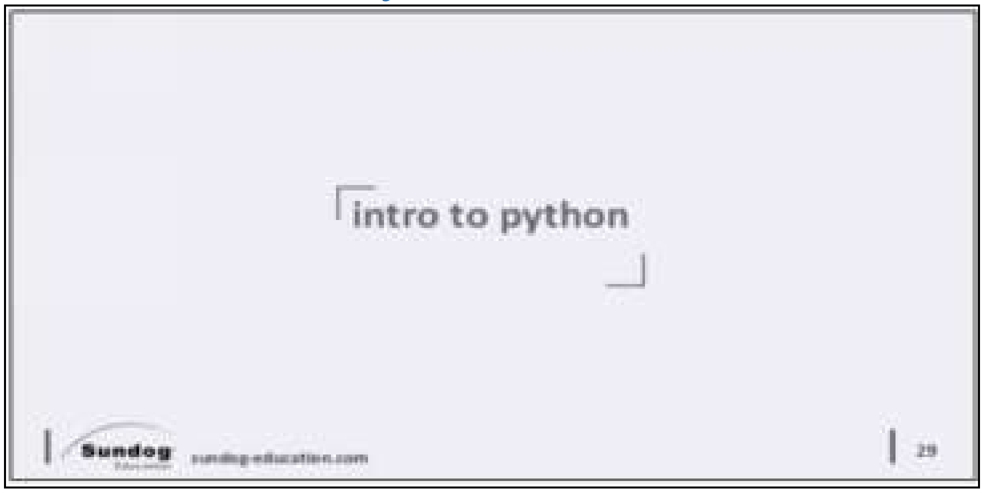
Sundog sundog-education.com

28

Well, candidate shuffling is something I just made up – although we will in fact build recommender systems that just recommend random, shuffled stuff as a baseline to compare against. Everything else is something we have to do in a top-N recommender – we need to generate recommendation candidates based on the user's interests, we need to rank those candidates, and then filter them down to the N results we want.

And that's it for this quiz, and for this initial section of the course! You've already learned a lot about the concepts, challenges, and architecture of recommender systems. Keep on going, as there's a lot more detail and hands-on practice ahead for you.

Introduction to Python


intro to python

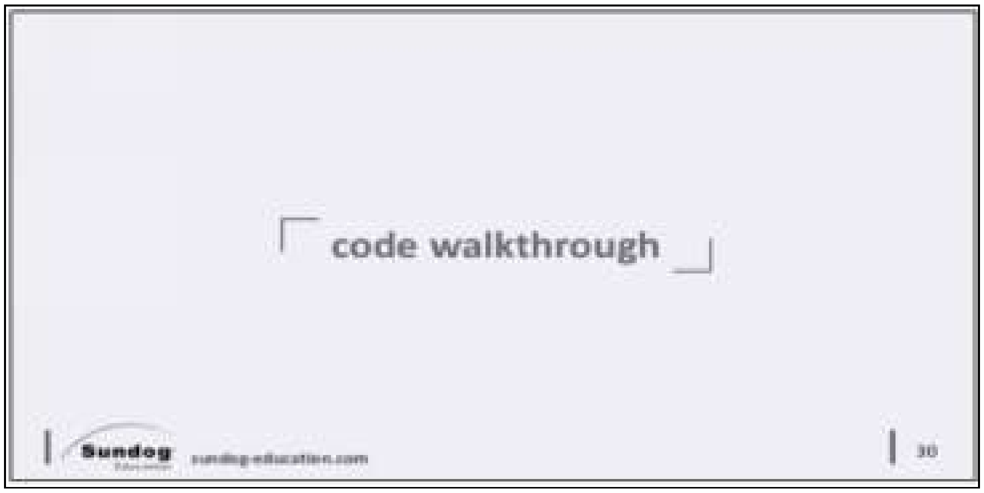
Sundog Education sundog-education.com

29

We're going to use the Python programming language in this course, as it's pretty easy to learn and use, and it's widely used in the world of data science and machine learning. In the real world, you'll get much better performance using languages like Java or C++, but I don't want to shut out students who aren't comfortable with lower-level languages like that.

Anyhow, if you already know Python, you can totally skip this section. But if you're new to Python or need a refresher, I've got a very quick crash course for you here. You are going to need some experience in programming or scripting for this to make sense – I can't teach you to program from scratch in just one section of a course. So if you've never programmed anything before, it would be a good idea to go grab a deeper introductory course on Python and go through that first, before coming back to this course.

Otherwise, let's shoulder on and learn the basics of Python!


code walkthrough

Sundog sundog-education.com 30

For our intro to Python, we're going to use a format called a Notebook – this lets us intersperse code with nicely formatted text that explains what's going on, and lets us run our code one little chunk at a time. So, the first thing we need to do is install a component into Anaconda called Jupyter Notebook.

To do that, just open up Anaconda Navigator, click on "Environments," and select our RecSys environment. Now, click on Home, and under Jupyter Notebook, hit install.

Once it's done, hit "Launch," and it should bring up a page in your web browser. Now, hit the upload button, and navigate to the Getting Started folder in your course materials. In there select the Python101.ipynb file. It should now appear at the top of your file list; hit the "upload" button next to it. Once it's uploaded, find it in the file list and select it to open it.

If you've never seen a Jupyter notebook before, the way it works is you can click in any of these blocks of code, hit the "run" button (or shift-enter), and it will execute that code right from your web browser. Let's try it with this first block; click inside of it to select it, and hit shift-enter.

Now, we're just going to cover the syntax of Python here and the main ways in which it differs from other languages. So let's take a closer look at this code. One thing with Python is that whitespace is really important; any nesting of code, like for iteration or conditional expressions, relies on the number of tabs to group code together, instead of curly brackets like other languages.

So, here we have a “list of numbers” – in Python, a list is like an array or vector in other languages. We define a list of the numbers 1-6 by putting them between square brackets, separated by commas. In Python, there is no character required to terminate a line; you just hit enter when you’re done. Let’s add 7 to the list, just to prove that running this actually does something. (Hit shift-enter) – yeah, we’ve got results for 1 through 7 now.

Next we have an example of a “for” loop in Python. This statement iterates through the list named “ listOfNumber s ”, storing the current iteration in the variable “ numbe r ” each time. A “for” statement does have to end with a colon, like this.

But now we use indents to indicate what code lives within this “for” block. And here we have an example of an “if / else” clause – if the number is evenly divisible by 2, we print that it’s even, otherwise we print that it’s odd. And again, we use indents to indicate what code lives within each if or else clause here.

We remove all indents to get out of the “for” loop, and print “all done” at the end .

Notice that we never had to declare any variables ahead of time, nor did we have to define their types. Python is what’s called a dynamically typed language. It tries to infer the data type of your variables based on how you initially use them, but internally they do have types. You can also explicitly cast variables to different types if you need to, but variables won’t be re-cast automatically as they would in weakly typed languages. Sometimes this can lead to unexpected errors, and we’ll see some of that as we go through the course.

Let’s move on to the next block, which shows how to import external modules into your Python scripts. You just use the import command for this, and you can define an alias for the module if you’d like to save yourself some typing, too. So, here we’re importing the numpy module so we can refer to it within our script, and we're importing it under the name "np". This lets us then use numpy's random function by just typing np.rando m – and in this case, we're asking numpy to give use ten normally-distributed random values with a given mean and standard deviation.

Next let's discuss lists in more depth, since we use them a lot. If you need to know how many elements are in a list, you can use the built-in le n function to get that, like so.

Often you also need to slice lists in certain ways, to extract values in a given range within the list. The colon is used for this; in this first example, we use : 3 to ask for the first three elements of the list…

And similarly, we can use 3 : to ask for everything after the third element, like so…

We can also do something like -2 : to ask for the last two elements of the list …

And, if you want to append a list to another list, that's what the extend function is for, like this… this will add the list containing 7 and 8 to our original list.

If you want to append a single value to a list, you can use the append function, like so.

Another cool thing about Python is that lists can contain just about any type you want. You can even make a list of lists! So let's do that right now – we'll make a new list called y , and make a new " listOfList s " that contains our newly-grown X list and this new Y list.

To retrieve an element of a list, just use the bracket operator, like this. Here we're getting back element 1 of the y list. This is zero-based, so y[1] actually gives you back the second element, not the first one. y[0] would give you the first element, which is the number 10 in this example.

Lists also have a built-in sort( ) function you can use to sort the list in-place, like so.

And, if you want to sort in reverse order, you just pass reverse=Tru e into the sor t function. This is also a good time to mention that there are a couple of ways to pass parameters into functions; you can just pass in a list of values like you would in most languages, but you can also pass them in by name. Often Python functions will have a lot of parameters that have default value assigned to them, and you just specify the ones you care about by specifying them by name.

OK, let's keep going, and talk about Tuples next. Tuples are a lot like lists, but the main difference is that they are immutable – once you create a tuple, you can't change them. They're handy for people doing functional programming, or for interfacing with systems like Apache Spark that are developed on functional programming languages – we'll do that later in the course.

The only real difference is that you enclose tuples with parentheses instead of square brackets. So here's a tuple of the values 1, 2, and 3. We can use len( ) on it, just like we would with a list…

You can reference elements in a tuple in the same way as you would in a list, as well. Again, it's zero-based, so y[2] gives us back the third element of the list, not the second one.

You can make a list of tuples if you so desire.

Another common use of tuples is in passing around groups of variables that you want to keep together. For example, the split function on a string will give you back a bunch of string values extracted from that string, and we can assign those values to elements in a tuple as a quick way of naming each one. Look at this example, we have two numbers separated by a comma, and we know the first value represents an age and the second an income value. We can extract them right into variables named age and income, like so.

Moving on, another useful data structure in Python in the dictionary. In other languages, you might know this as a map or a hash table. It's basically a lookup table, where you store values associated with some unique set of key values. It makes more sense with an example.

You declare a dictionary using curly brackets, so let’s make a dictionary mapping starship names to the names of their captains. We’ll call this dictionary “captains” .

Now, to create an entry in the dictionary, we use square brackets to specify a key value that we’re interested in assigning. So to assign the value “Kirk” to the key “Enterprise,” we can just say captains[“Enterprise”] = “Kirk”.

We do the same for the other starships we know about. Then, retrieving a dictionary element is done in the same way – just use square brackets to get back the value you want. So, to get the captain of the USS Voyager, we do it like so.

But what happens if you try to retrieve a value for a key that doesn’t exist? You’d get an exception in that case. One way to avoid that is to use the get function on the dictionary. So, you can see you can retrieve the captain of the Enterprise successfully as it exists in the dictionary.

But if we try to “get” a ship that isn’t in our dictionary, it returns the special value Non e that you can test for and deal with however you want. And if you know that the captain of the NX-01 is Jonathan Archer, you’re my new best friend.

You can iterate through all of the keys in a dictionary just like you’d iterate through a list, like this – for ship in captain s will give you back each key in the captains dictionary, and name it ship for you, so we can then print it out, like so.

OK, moving on to functions now! Fortunately they work pretty much as you’d expect in Python. The syntax looks like this: you define a function with the de f keyword, followed by the function name, followed by any parameters you want to pass in with their names. Be sure to end the function definition with a colon .

After that, your code within the function needs to be indented – Python's all about the whitespace, remember. So here we have a super simple function called SquareI t that just takes in one value, calls it x , and returns its square.

Calling the function works exactly as you would expect – just type SquareI t then 2 in parentheses to get the square value of 2, for example, Let's run it…

There are a few funky things you can do with functions in Python. For example, you can actually pass in another function to a function as a parameter. This is something that allows Python to play nice with functional programming languages, which you encounter a lot in distributed computing. We can create a function like this called DoSomethin g that takes in some function f , some parameter x , and returns the result of f(x). So, we can actually say DoSomething(SquareIt, 3) as a really convoluted way of getting back the square of 3, but it works!

By the way, did you notice that this block of code still knows that the SquareI t function exists, even though we defined it in a previous block? Jupyter notebooks run within a single kernel session, so anything you define or modify before the block you're running will still be in memory within that kernel. Sometimes this can lead to confusing behavior, especially if you're running blocks within a notebook out of order, or repeatedly. If weird things start happening, you can always start from a clean slate by selecting Kernel / restart from the menu.

Let's also give some attention to lambda functions – these allow you to pass around simple functions inline, without even giving them a name. You see this pretty often in the world of data science. It's easiest to understand with an example – Here, we're calling our DoSomethin g function with a lambda function that computes the cube of a value instead of its square. Take a close look at the syntax here – we say lambd a followed by the function parameter, then a colon after which we can do what we want to that parameter. Whatever value is computed after the colon is implicitly the return value of the lambda function. So, this says to create a lambda function that take in something called x , and return x * x * x . We pass that lambda function into DoSomethin g with a value of 3, which then executes our lambda function for the value 3. And if we run this, you can see we do in fact get back 27, which is 3 cubed.

OK, the other stuff is pretty straightforward. Here's what the syntax looks like for Boolean expressions. You can use a double equals sign to test for equality, and you can also just use the word "is". There's also a triple equals sign you can use to compare actual objects together instead of their values, but that's seen less commonly. True and false are a little weird in that they are capitalized in Python, but only the first letter, like so. And for operators like and, or, and not, you just use the words and, or and not, instead of special symbols.

You can do something like a cas e or switc h statement by using Boolean expressions with an "if / elif / else" structure, like this. In this example, we print "how did that happen" if 1 is 3, which is always false. Otherwise, we move on to test whether 1 is greater than 3, which should also be met with disbelief if it were true. When that test fails, we fall back to our final else clause, which prints "all is well with the world." Again we use indents to associate code with specific clauses here.

We touched on looping earlier; let's go into more depth there. You can use the rang e function to automatically build up a list of values within a given range, like so: Again, we start counting from 0 here, so range(10) gives us back the values 0 through 9 .

The continu e keyword within a loop means to skip the rest of the iteration and go straight to the next one, and the brea k keyword means to stop the iteration early. In this example, we use continue to skip over the printing of each value if the value is 1, and we use break to stop after we reach the value 5… study the output of 0, 2, 3, 4, 5, and you'll see that's exactly what it did.

An alternative syntax is the “while” loop, which iterates until some Boolean expression is false. Here we set up a counter variable called x , and loop through printing x only while x is less than 10. Once it hits 10, the loop ends, as you can see in its output.

So, try out a really simple activity that pieces together some of the stuff we’ve just talked about. Your challenge is to write a block of code here that creates a list of integers, loops through each element of the list, and only prints out even numbers in the list. This should just be a matter of re-arranging some of the other code in this notebook, so even if you’re not comfortable with Python I encourage you to give it a try – you’ll feel a lot better with Python if you’ve written something yourself with it , no matter how small.

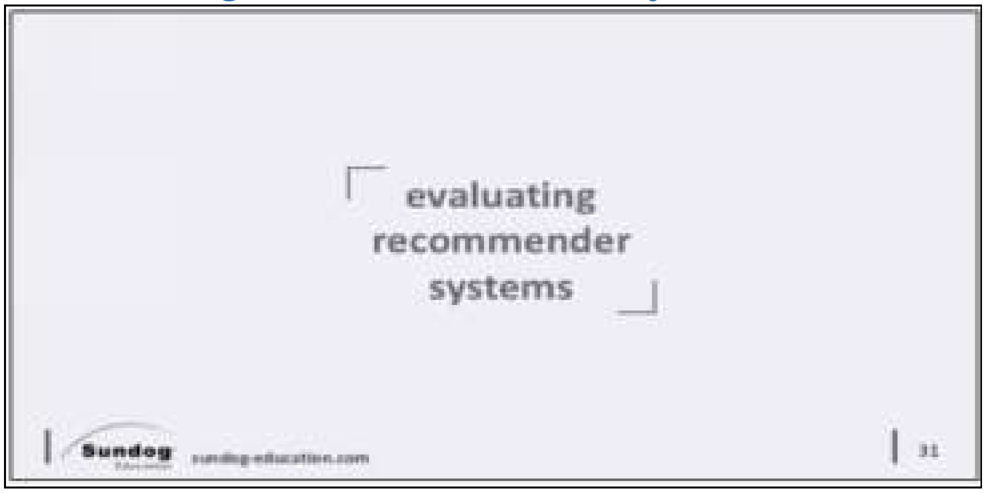
OK, so hopefully you didn’t have any trouble with that. Here’s my solution, and keep in mind there’s more than one way to do it:

```
myList = [0, 1, 2, 5, 8, 3 ]  (use any numbers you want)

for number in myList :  (you might choose different variable names)

        if number %2 is 0: (you could use == instead of “is” here)

              print(number )
```

And you can see, we do get the expected output of just the even numbers here.

OK, so that’s everything that’s peculiar about Python, for the most part. If you’re coming from some other language, that should give you enough knowledge to understand the Python code I’ll be showing you in this course, and to work with that code yourself. But if you’re really befuddled at this point, I do encourage you to grab a larger introductory Python course before moving forward in this one. An ability to read Python code is going to help this course make a lot more sense to you. But if you think you’ve got the basics here, let’s move forward!

Evaluating Recommender Systems

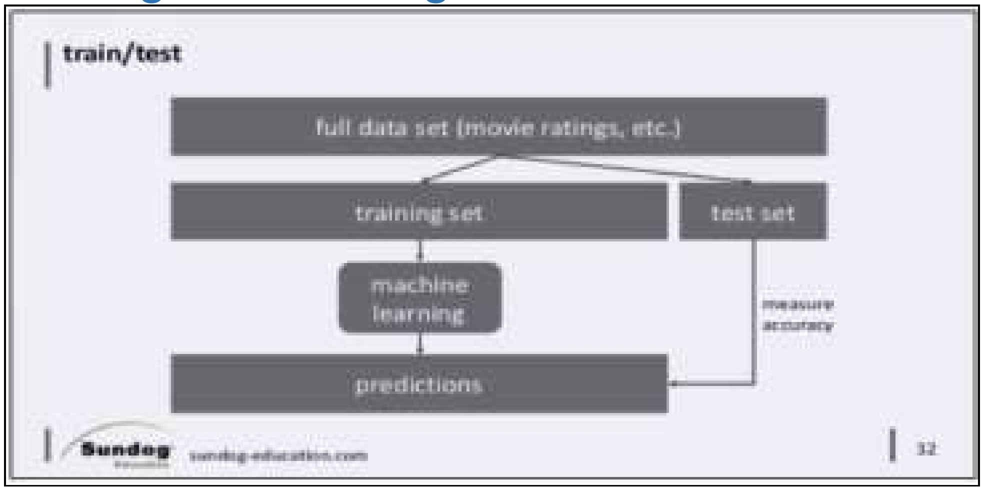

evaluating recommender systems

Sundog sundog-education.com 31

A big part of why recommender systems are as much art as they are science is that it's difficult to measure how good they are. There is a certain aesthetic quality to the results they give you, and it's hard to say whether a person considers a recommendation to be good or not – especially if you're developing algorithms offline.

People have come up with a lot of different ways to measure the quality of a recommender system, and often different measurements can be at odds with each other. But let's go through the more popular metrics for recommender systems, as they all have their own uses .

Testing Methodologies

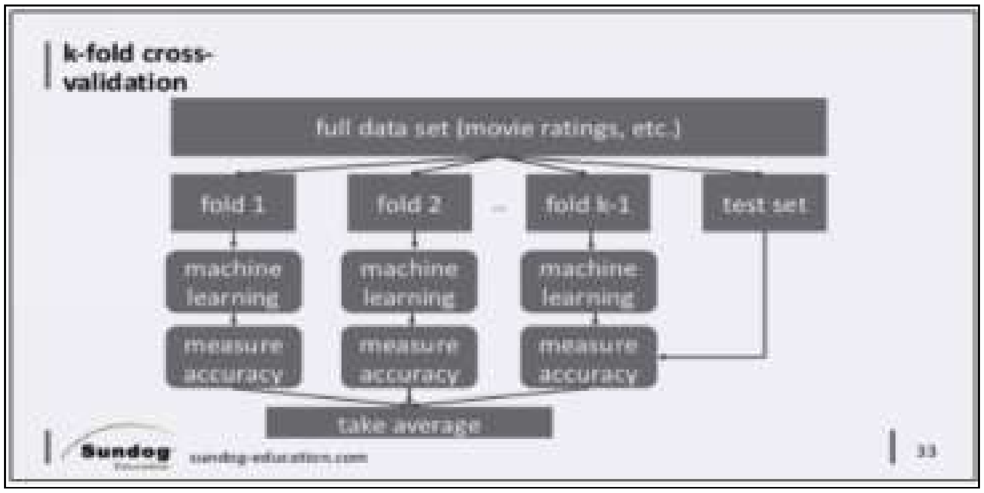


First, let’s talk about the methodology for testing recommender systems offline. If you’ve done machine learning before, you’re probably familiar with the concept of train/test splits. A recommender system is a machine learning system – you train it using prior user behavior, and then use it to make predictions about items new users might like. So on paper at least, you can evaluate a recommender system just like any other machine learning system.

Here’s how it works – you measure your recommender system’s ability to predict how people rated things in the past. But to keep it honest, you start by splitting up your ratings data into a training set, and a testing set. Usually the training set is bigger, say 80 or 90 percent of all of your data – and you randomly assign ratings into one or the other.

So, you train your recommender system only using the training data. This is where it learns the relationships it needs between items or between users. Once it’s trained, you can ask it to make predictions about how a new user might rate some item they’ve never seen before .

So to measure how well it does, we take the data we reserved for testing. These are ratings that our recommender system has never seen before, so that keeps it from “cheating.” Let’s say one rating in our test set says that the user actually rated the movie “Up” 5 stars. We just ask the recommender system how it thinks this user would rate “Up” without telling it the answer, and then we can measure how close it came to the real rating. If you do this over enough people, you can end up with a meaningful number that tells you how good your recommender system is at recommending things – or more specifically, recommending things people already watched and rated. That’s really all you can do if you can’t test things out in an online system.



If you really want to get fancy, it's possible to improve on a single train/test split by using a technique called k-fold cross validation. It's the same idea as train/test, but instead of a single training set, we create many randomly assigned training sets. Each individual training set, or "fold" is used to train your recommender system independently, and then we measure the accuracy of the resulting systems against your test set. So, we end up with a score of how accurately each fold ends up predicting user ratings, and we can average them together.

This obviously takes a lot more computing power to do, but the advantage is that you don't end up "overfitting" to a single training set. If your training data is small, you run the risk of optimizing for the ratings that are specifically in your training set instead of the test set – so k-fold cross-validation provides some insurance against that, and ensures you create a recommender system that works for any set of ratings, not just the ones in the training set you happened to choose.

Accuracy Measures

measuring accuracy

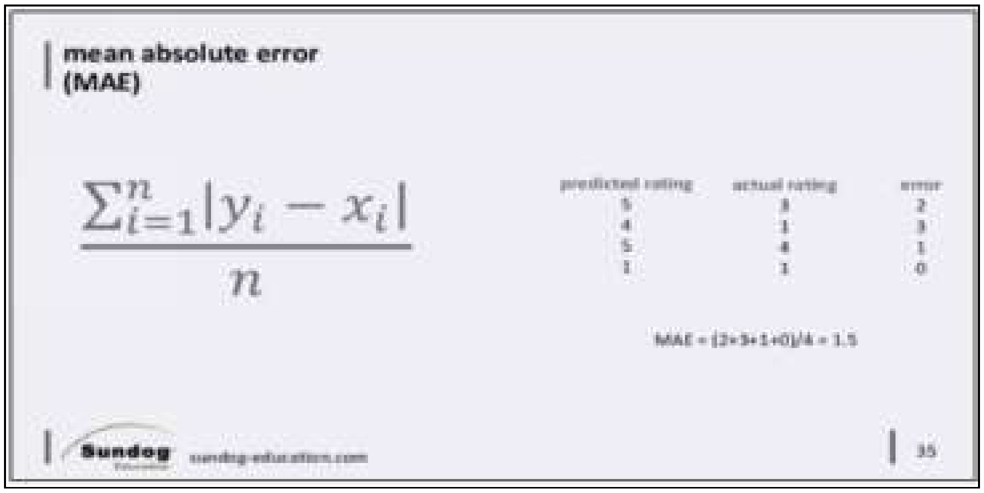
Sundog sundog-education.com

34

To reiterate, train/test and k-fold cross validation are ways to measure the accuracy of your recommender system. That is, how accurately you can predict how users rated movies they have already seen and provided a rating for.

But, that’s an important point! By using train/test, all we can do is test our ability to predict how people rated movies they already saw! That’s not the point of a recommender system – we want to recommend new things to people that they haven’t seen but find interesting! However, that’s fundamentally impossible to test offline, so researchers who can’t just test out new algorithms on real people on Netflix or Amazon or whatever have to make do with approaches like this.

We haven’t talked about how to actually come up with an accuracy metric when testing your recommender system, so let’s cover a couple of different ways to do it.



mean absolute error (MAE)

$$\frac{\sum_{i=1}^{n}|y_i - x_i|}{n}$$

predicted rating	actual rating	error
5	3	2
4	1	3
5	4	1
1	1	0

MAE = (2+3+1+0)/4 = 1.5

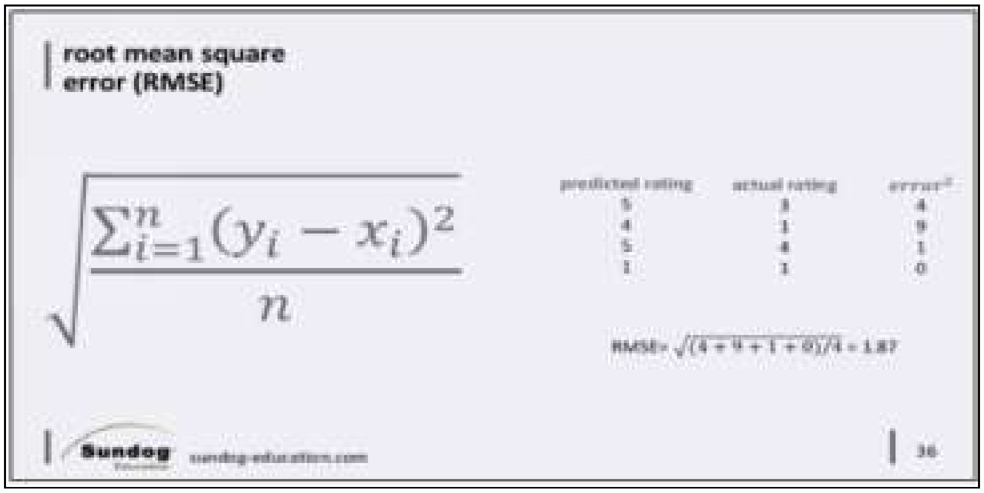
Sundog sundog-education.com

35

The most straightforward metric is mean absolute error, or MAE. Here's the fancy mathematical equation for how to compute it – it's not as complicated as it looks, so let's break it down. Let's say we have n ratings in our test set that we want to evaluate. For each rating, we can call the rating our system predicts y, and the rating the user actually gave x. Just take the absolute value of the difference between the two to measure the error for that rating prediction – it's literally just the difference between the predicted rating and the actual rating .

We sum those errors up across all n ratings in our test set, and divide by n to get the average, or mean. So, mean absolute error is exactly that – the mean, or average absolute values of each error in rating predictions. Remember error is bad, so you want the lowest MAE score you can get, not the highest.

Let's look at a simple example. Let's say we have just 4 ratings in our test set. On the first one, our recommender system predicted a rating of 5, when the actual rating was 3. The absolute error in this case is 2. We just do that for every rating, add them all up, and divide by four. So the MAE score in this case is 1.5. It's just that simple.



root mean square error (RMSE)

$$\sqrt{\frac{\sum_{i=1}^{n}(y_i - x_i)^2}{n}}$$

predicted rating	actual rating	$error^2$
5	3	4
4	1	9
5	4	1
1	1	0

RMSE= $\sqrt{(4+9+1+0)/4} = 1.87$

Sundog Education sundog-education.com

36

A slightly more fancy way to measure accuracy is root mean square error, or RMSE. This is a more popular metric for a few reasons, but one is that it penalizes you more when your rating prediction is way off, and penalizes you less when you were reasonably close.

The difference is that instead of summing up the absolute values of each rating prediction error, we sum up the squares of the rating prediction errors instead. Taking the square ensures we add up positive numbers, like absolute values do, and it also inflates the penalty for larger errors. When we're done, we take the square root to get back to a number that makes sense.

So again, you can just take the name of the metric literally. "Root" means we take the square root of the whole thing when we're done, "mean" refers to an average, and "square error" is the square of each individual rating prediction error.

Let's walk through that same example – in the first rating in our test set, the error between the predicted rating of 5 and actual rating of 3 is 2, and the square of that error is 4. We just keep doing that for each rating in our test set and add them all up. Then divide by the number of ratings, take the square root, and this time we end up with 1.87. That's higher than the MAE score for this same data of 1.5, because it's penalizing us hard for predicting a rating of 4 on an item the user hated with a rating of 1. Again, remember high RMSE's are bad – you want low error scores, not high ones.

how did we get here?

Sundog sundog-education.com

37

Now like we said, accuracy isn't really measuring what we want recommender systems to do. Actual people couldn't care less what your system thinks you should have rated some movie you already saw and rated. Rating predictions themselves are really of limited value; you don't really care if my system thinks you'll rate "Up" 3 stars. You care mostly about what my system thinks the best movies for you to go see are, and that's a very different problem.

So how did we end up focusing on accuracy and RMSE scores in the field of recommender system research? Well, a lot of it dates back to 2006, when Netflix offered a one million dollar prize to the first person or group to achieve a 10% improvement on Netflix's own RMSE score on a data set of movie ratings it made public. This was called the Netflix Prize, and that million dollar prize spurred a lot of research into recommender systems.

Since the prize was based on achieving low RMSE scores, that's what everyone focused on, and that focus has rippled through time even today. And to be fair, there's not a lot of ways to measure a recommender system given only historical data to work with. Ideally you want to measure how real people react to recommendations for things they've never seen before, and you just can't do that offline.

If you're curious, the prize was awarded in 2009 to a team called "BellKor," and we'll talk about how they did it later on. But Netflix didn't actually end up using their research, because they realized that RMSE just doesn’t matter much in the real world. What does matter is which movies you put in front of users in a top-N recommender list, and how users react to those movies when they see them recommended .

So, in hindsight, maybe Netflix should have based the Netflix Prize on a metric that’s more focused on top-N recommenders. It turns out there are a few of them. One is called “Hit Rate”, and it’s really simple.

You generate top-N recommendations for all of the users in your test set. If one of the recommendations in a users’ top-N recommendations is something they actually rated, you consider that a “hit”. You actually managed to show the user something that they found interesting enough to watch on their own already, so we’ll consider that a success.

Just add up all of the “hits” in your top-N recommendations for every user in your test set, divide by the number of users, and that’s your hit rate.

leave-one-out cross validation

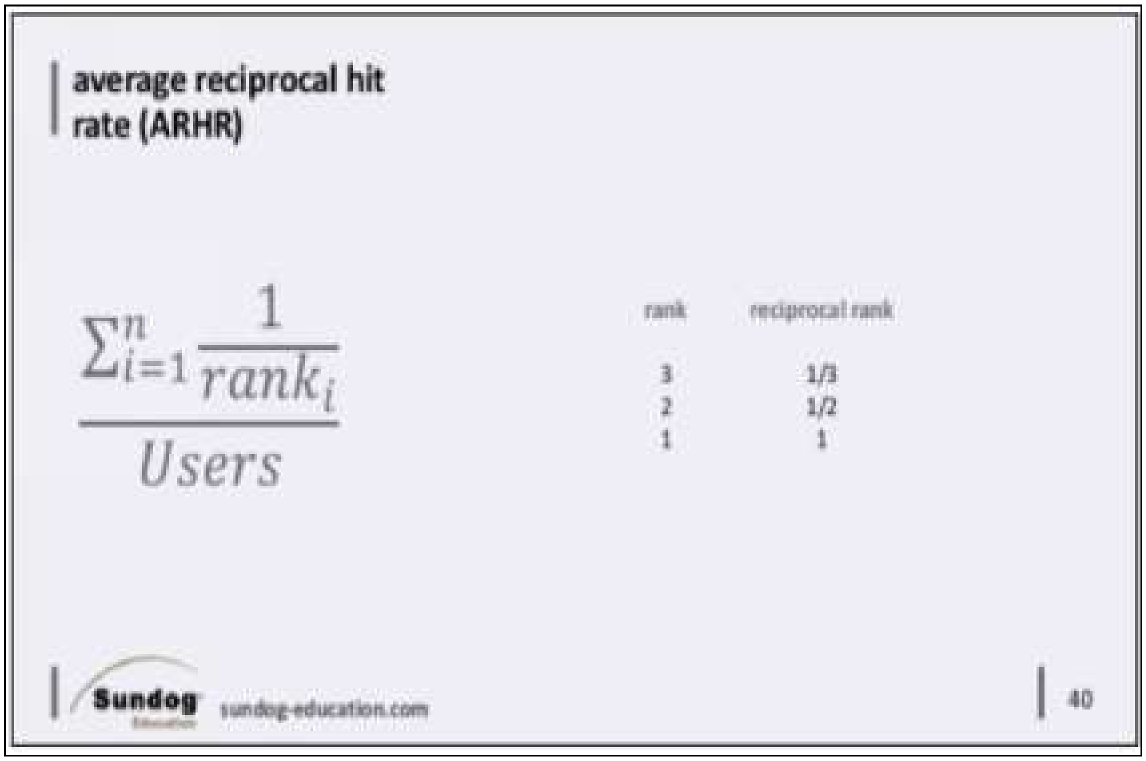
Sundog Education sundog-education.com 39

Hit rate itself is easy to understand, but measuring it is a little tricky. We can’t use the same train/test or cross validation approach we used for measuring accuracy, because we’re not measuring the accuracy on individual ratings – we’re measuring the accuracy of top-N lists for individual users. Now, you could do the obvious thing and not split things up at all, and just measure hit rate directly on top-N recommendations created by a recommender system that was trained on all of the data you have.

But, technically that’s cheating. You generally don’t want to evaluate a system using data that it was trained with. I mean, think about it – you could just recommend the actual top 10 movies rated by each user using the training data, and achieve a hit rate of 100%.

So, a clever way around this is called “leave one out cross validation.” What we do is compute the top N recommendations for each user in our training data, and intentionally remove one of those items from that user’s training data. We then test our recommender system’s ability to recommend that item that was left out in the top N results it creates for that user in the testing phase. So, we measure our ability to recommend an item in a top-N list for each user that was left out from the training data. That’s why it’s called leave one out.

The trouble is, it’s a lot harder to get one specific movie right while testing than to just get one of the N recommendations – so hit rate with leave-one-out tends to be very small, and difficult to measure unless you have very large data sets to work with. But, it’s a much more user-focused metric when you know your recommender system will be producing top-N lists in the real world, which most of them do.



average reciprocal hit rate (ARHR)

$$\frac{\sum_{i=1}^{n} \frac{1}{rank_i}}{Users}$$

rank	reciprocal rank
3	1/3
2	1/2
1	1

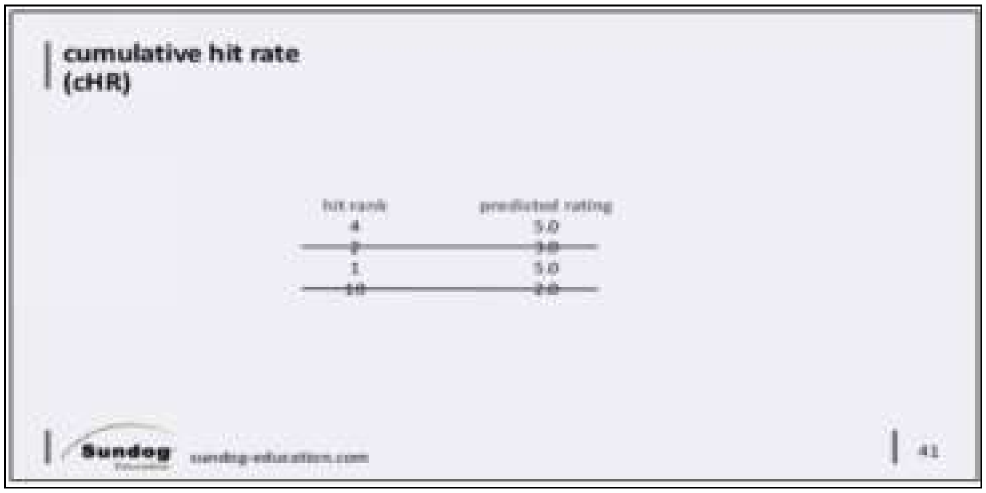
Sundog sundog-education.com 40

Hit Rate Measures

A variation on hit rate is average reciprocal hit rate, or ARHR for short. This metric is just like hit rate, but it accounts for where in the top-N list your hits appear. So, you end up getting more credit for successfully recommending an item in the top slot than in the bottom slot. Again, this is a more user-focused metric, since users tend to focus on the beginning of lists.

The only difference is that instead of summing up the number of hits, we sum up the reciprocal rank of each hit. So, if we successfully predict a recommendation in slot 3, that only counts as 1/3. But a hit in slot 1 of our top-N recommendations receives the full weight of 1.0.

Whether this metric makes sense for you depends a lot on how your top-N recommendations are displayed. If the user has to scroll or paginate to see the lower items in your top-N list, then it makes sense to penalize good recommendations that appear too low in the list, where the user has to work to find them.



cumulative hit rate (cHR)

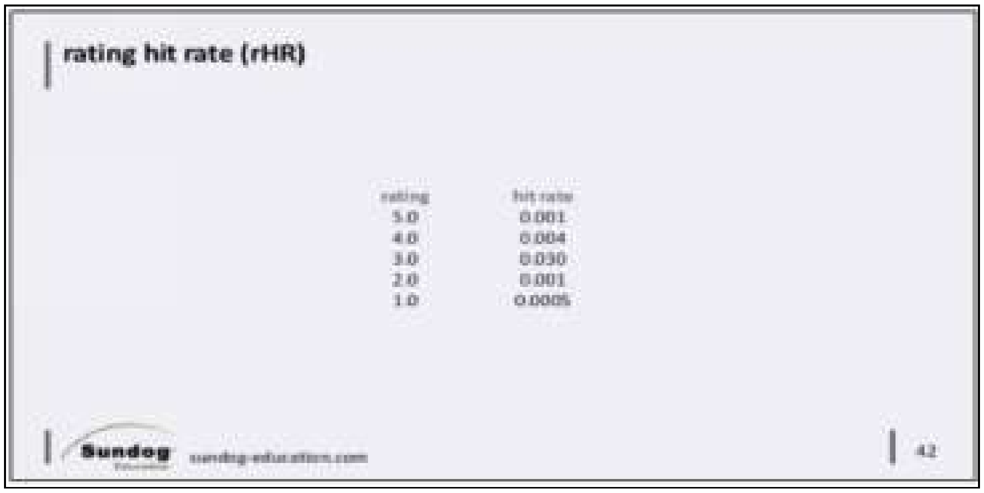
hit rank	predicted rating
4	5.0
~~2~~	~~3.0~~
1	5.0
~~10~~	~~2.0~~

Sundog sundog-education.com

41

Another twist is cumulative hit rank. Sounds fancy, but all it means is that we throw away hits if our predicted rating is below some threshold. The idea is that we shouldn't get credit for recommending items to a user that we think they won't actually enjoy.

So in this example, if we had a cutoff of 3 stars, we'd throw away the hits for the second and fourth items in these test results, and our hit rate metric wouldn't count them at all.



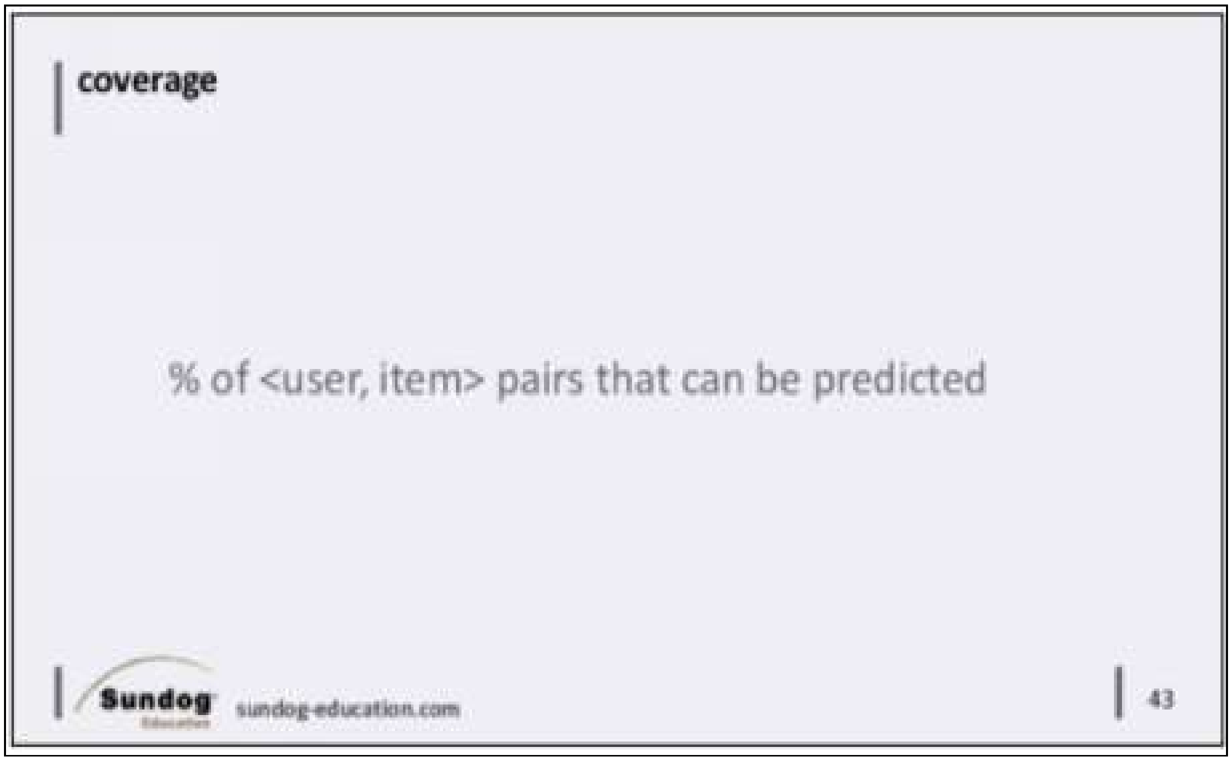
rating hit rate (rHR)

rating	hit rate
5.0	0.001
4.0	0.004
3.0	0.030
2.0	0.001
1.0	0.0005

Sundog sundog-education.com 42

Yet another way to look at hit rate is to break it down by predicted rating score. It can be a good way to get an idea of the distribution of how good your algorithm thinks recommended movies are that actually get a hit. Ideally you want to recommend movies that they actually liked, and breaking down the distribution gives you some sense of how well you're doing in more detail. This is called rating hit rate, or rHR for short.

So, those are all different ways to measure the effectiveness of top-N recommenders offline. The world of recommender systems would probably be a little different if Netflix awarded the Netflix Prize on hit rate instead of RMSE! It turns out that small improvements in RMSE can actually result in large improvement to hit rates, which is what really matters – but it also turns out that you also can build recommender systems with great hit rates, but poor RMSE scores – and we'll see some of those later in this course. So, RMSE and hit rate aren't always related.



coverage

% of <user, item> pairs that can be predicted

Sundog

sundog-education.com

43

Coverage

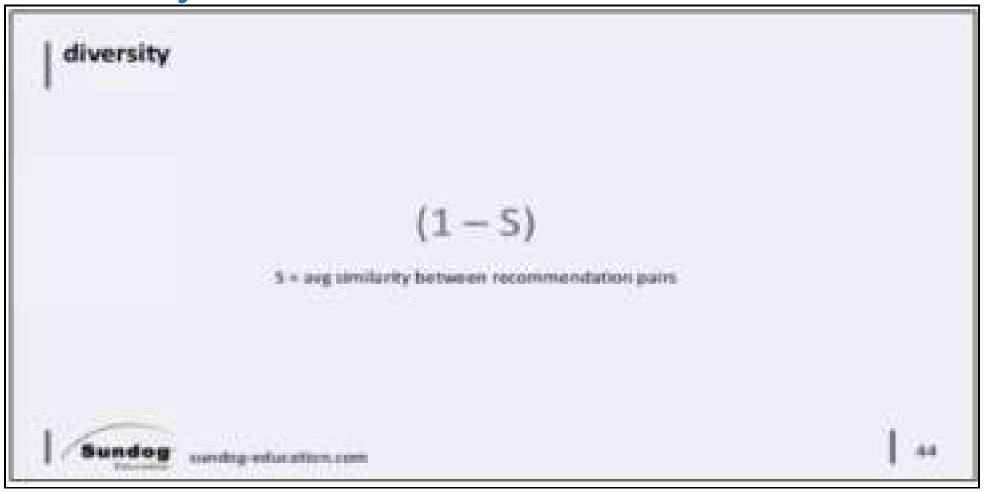
Accuracy isn’t the only thing that matters with recommender systems; there are other things we can measure if they are important to us.

For example, coverage. That’s just the percentage of possible recommendations that your system is able to provide. Think about the MovieLens data set of movie ratings we’re using in this course; it contains ratings for several thousand movies, but there are plenty of movies in existence that it doesn’t have ratings for. If you were using this data to generate recommendations on, say , IMDb, then the coverage of this recommender system would be low – because IMDb has millions of movies in its catalog, not thousands.

It’s worth noting that coverage can be at odds with accuracy. If you enforce a higher quality threshold on the recommendations you make, then you might improve your accuracy at the expense of coverage. Finding the balance of where exactly you’re better off recommending nothing at all can be delicate .

Coverage can also be an important to watch, because it gives you a sense of how quickly new items in your catalog will start to appear in recommendations. When a new book comes out on Amazon, it won't appear in recommendations until at least a few people buy it, therefore establishing patterns with the purchase of other items. Until those patterns exist, that new book will reduce Amazon's coverage metric.

Diversity



diversity

$$(1 - S)$$

S = avg similarity between recommendation pairs

Sundog Education sundog-education.com

44

Another metric is called diversity. You can think of this as a measure of how broad a variety of items your recommender system is putting in front of people. An example of low diversity would be a recommender system that just recommends the next books in a series that you've started reading, but doesn't recommend books from different authors, or movies related to what you've read.

This may seem like a subjective thing, but it is measurable. Many recommender systems start by computing some sort of similarity metric between items, so we can use these similarity scores to measure diversity. If we look at the similarity scores of every possible pair in a list of top-N recommendations, we can average them to get a measure of how similar the recommended items in the list are to each other. We can call that measure S. Diversity is basically the opposite of average similarity, so we subtract it from 1 to get a number associated with diversity.

It’s important to realize that diversity, at least in the context of recommender systems, isn’t always a good thing. You can achieve very high diversity by just recommending completely random items – but those aren’t good recommendations by any stretch of the imagination! Unusually high diversity scores mean you just have bad recommendations, more often than not. You always need to look at diversity alongside metrics that measure the quality of the recommendations as well.

Novelty

novelty

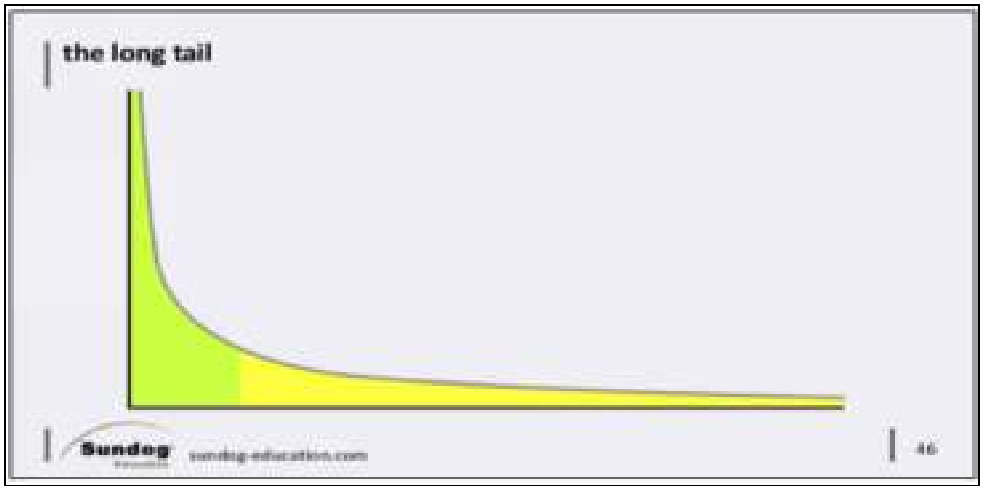
mean popularity rank of recommended items

Sundog sundog-education.com 45

Similarly, novelty sounds like a good thing, but often it isn’t. Novelty is a measure of how popular the items are that you are recommending. And again, just recommending random stuff would yield very high novelty scores, since the vast majority of items are not top-sellers .

Although novelty is measurable, what to do with it is in many ways subjective. There’s a concept of user trust in a recommender system – people want to see at least a few familiar items in their recommendations that make them say, “yeah, that’s a good recommendation for me. This system seems good.” If you only recommend things people have never heard of, they may conclude that your system doesn’t really know them, and they may engage less with your recommendations as a result. Also, popular items are usually popular for a reason – they’re enjoyable by a large segment of the population, so you would expect them to be good recommendations for a large segment of the population who hasn't read or watched them yet. If you're not recommending some popular items, you should probably question whether your recommender system is really working as it should.

This is an important point – you need to strike a balance between familiar, popular items, and what we call "serendipitous discovery" of new items the user has never heard of before. The familiar items establish trust with the user, and the new ones allow the user to discover entirely new things that they might love.



Novelty is important, though, because the whole point of recommender systems is to surface items in what we call "the long tail." Imagine this is a plot of the sales of items of every item in your catalog, sorted by sales. So, number of sales, or popularity, is on the Y axis, and all the products are along the X axis. You almost always see an exponential distribution like this… most sales come from a very small number of items, but taken together, the "long tail" makes up a large amount of sales as well. Items in that "long tail" – the yellow part, in the graph – are items that cater to people with unique, niche interests.

Recommender systems can help people discover those items in the long tail that are relevant to their own unique, niche interests. If you can do that successfully, then the recommendations your system makes can help new authors get discovered, can help people explore their own passions, and make money for whoever you're building the system for as well. Everybody wins! When done right, recommender systems with good novelty scores can actually make the world a better place.

But again, you need to strike a balance between novelty and trust – as I said, building recommender systems is a bit of an art, and this is an example of why .

Churn

churn

how often do recommendations change?

Sundog
sundog-education.com
47

Another thing we can measure is churn. How often do the recommendations for a user change? In part, churn can measure how sensitive your recommender system is to new user behavior. If a user rates a new movie, does that substantially change their recommendations? If so, then your churn score will be high.

Maybe just showing someone the same recommendations too many times is a bad idea in itself. If a user keeps seeing the same recommendation but doesn't click on it, at some point should you just stop trying to recommend it and show the user something else instead? Sometimes a little bit of randomization in your top-N recommendations can keep them looking fresh, and expose your users to more items than they would have seen otherwise.

But, just like diversity and novelty, high churn is not in itself a good thing. You could maximize your churn metric by just recommending items completely at random, and of course those would not be good recommendations. All of these metrics need to be looked at together, and you need to understand the tradeoffs between them.

Responsiveness

responsiveness

how quickly does new user behavior influence your recommendations?

Sundog sundog-education.com 48

One more metric is responsiveness – how quickly does new user behavior influence your recommendations? If you rate a new movie, does it affect your recommendations immediately – or does it only affect your recommendations the next day after some nightly job runs?

More responsiveness would always seem to be a good thing, but in the world of business you have to decide how responsive your recommender really needs to be – since recommender systems that have instantaneous responsiveness are complex, difficult to maintain, and expensive to build. You need to strike your own balance between responsiveness and simplicity.

We've covered a LOT of different ways to evaluate your recommender system – MAE, RMSE, Hit Rate in various forms, coverage, diversity, novelty, churn, and responsiveness. So how do you know what to focus on? Well, the answer is that it depends. It may even depend on cultural factors – some cultures may want more diversity and novelty in their recommendations than others, while other cultures may want to stick with things that are familiar to them. It also depends on what you're trying to achieve as a business. And usually, a business is just trying to make money – which leads to one more way to evaluate recommender systems that is arguably the most important of all …

A/B Tests

That’s doing online A/B tests to tune your recommender system using your real customers, and measuring how they react to your recommendations. You can put recommendations from different algorithms in front of different sets of users, and measure if they actually buy, watch, or otherwise indicate interest in the recommendations you’ve presented. By always testing changes to your recommender system using controlled, online experiments – you can see if they actually cause people to discover and purchase more new things than they would have otherwise. That’s ultimately what matters to your business, and it’s ultimately what matters to your users, too.

NONE of the metrics we’ve discussed matter more than how real customers react to the recommendations you produce, in the real world. You can have the most accurate rating predictions in the world, but if customers can’t find new items to buy or watch from your system, it will be worthless from a practical standpoint .

If you test a new algorithm and it’s more complex than the one it replaced, then you should discard it if it does not result in a measurable improvement in users interacting with the recommendations you present. Online tests can help you to avoid introducing complexity that adds no value, and remember complex systems are difficult to maintain.

So remember, offline metrics such as accuracy, diversity, and novelty can all be indicators you can look at while developing recommender systems offline, but you should never declare victory until you’ve measured a real impact on real users from your work. Systems that look good in an offline setting often fail to have any impact in an online setting, that is, in the real world. User behavior is the ultimate test of your work.

There is a real effect where often accuracy metrics tell you an algorithm is great, only to have it do horribly in an online test. YouTube studied this, and calls it the “surrogate problem.” Accurately predicted ratings don’t necessarily make for good video recommendations. YouTube said in one of their papers, and I quote

– “There is more art than science in selecting the surrogate problem for recommendations.” What they mean is that you can’t always use accuracy as a surrogate for good recommendations. Netflix came to the same conclusion, which is why they aren’t really using the results of that one million dollar prize for accuracy they paid out.

At the end of the day, the results of online A/B tests are the only evaluation that matters for your recommender system.

perceived quality

Sundog sundog-education.com 51

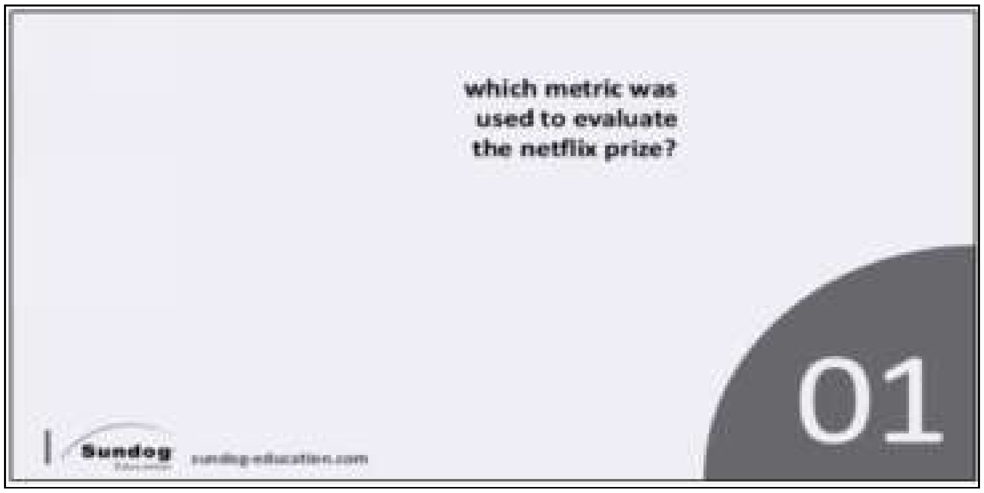
Another thing you can do is just straight up ask your users if they think specific recommendations are good. Just like you can ask for explicit feedback on items with ratings, you can ask users to rate your recommendations too! This is called measuring the “perceived quality” of recommendations, and it seems like a good idea on paper, since as you’ve learned defining what makes a “good” recommendation is by no means clear. In practice though, it’s a tough thing to do. Users will probably be confused over whether you’re asking them to rate the item or rate the recommendation, so you won’t really know how to interpret this data. It also requires extra work from your customers with no clear payoff for them, so you’re unlikely to get enough ratings on your recommendations to be useful. It’s best to just stick with online A/B tests, and measure how your customers vote with their wallets on the quality of your recommendations .

Quiz

quiz time

Sundog sundog-education.com 52

We’ve covered a lot in this section so far – but it’s important to understand what makes a good recommender system before we start trying to build one. So, let’s reinforce some of what we’ve learned with another short quiz.



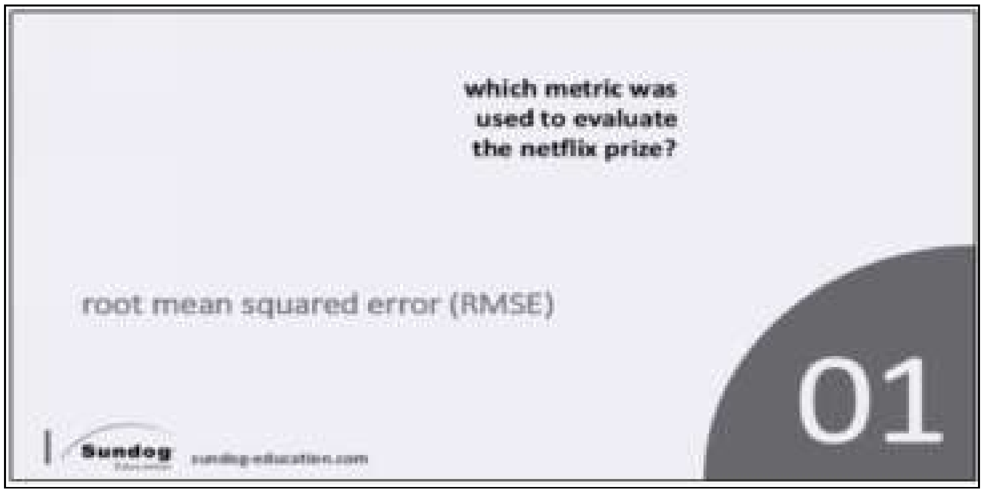
which metric was
used to evaluate
the netflix prize?

01

Sundog

sundog-education.com

Your first question is: which metric was used to evaluate the Netflix prize? By putting a one million dollar bounty on improving a specific metric, Netflix reshaped the world of recommender system research to focus on that metric, for better or worse. What was it?



which metric was used to evaluate the netflix prize?

root mean squared error (RMSE)

01

Sundog sundog-education.com

The answer is root mean squared error, or RMSE. But as we've said, accuracy isn't everything in the real world. Users don't care how accurately you can predict how they rated movies they've already seen, they care about your ability to show them new things that they will love. Arguably, Netflix should have focused on a metric more focused on top-N recommendations – which leads to our next question…

Sundog sundog-education.com

what's a metric for top-n recommenders that accounts for the rank of predicted items?

02

55

What's a metric for top-N recommenders that accounts for the rank of the predicted items? In truth, there's more than one – but there's only one that we talked about in this course so far.

Sundog sundog-education.com

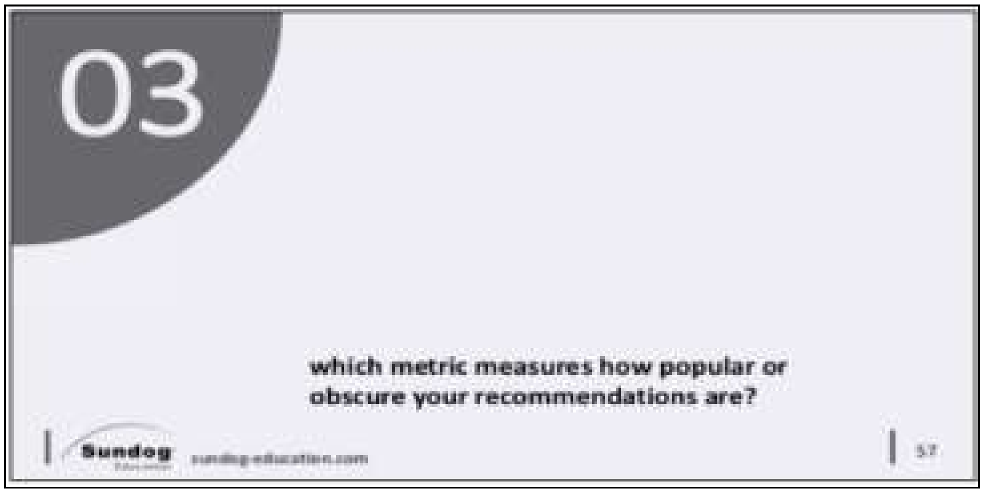
what's a metric for top-n recommenders that accounts for the rank of predicted items?

average reciprocal hit rank

02

56

The answer is average reciprocal hit rank, or ARHR for short. It measures our ability to recommend items that actually appeared in a user's top-N highest rated movies, and gives more weight to these "hits" when they appear near the top of the top-N list.

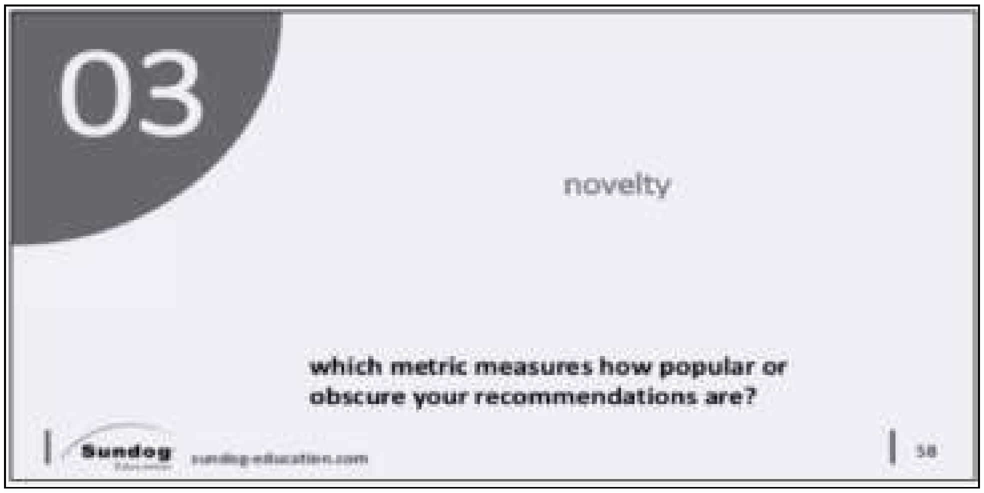

03

which metric measures how popular or obscure your recommendations are?

Sundog sundog-education.com 57

We talked about the “long tail,” and how there is real social value in recommending items that are relatively obscure, yet relevant to someone’s unique, niche interests.

What metric tells us about how popular or obscure our recommendations are?



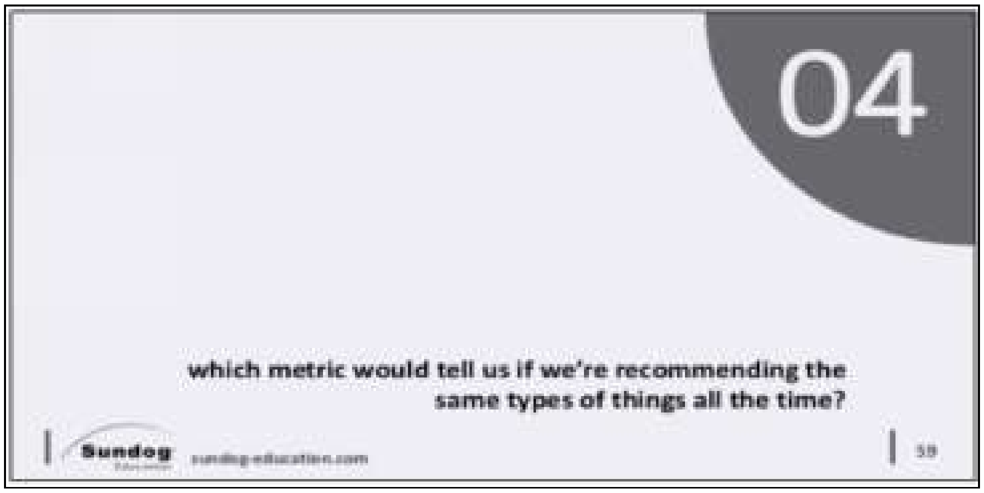
03

novelty

which metric measures how popular or obscure your recommendations are?

Sundog sundog-education.com 58

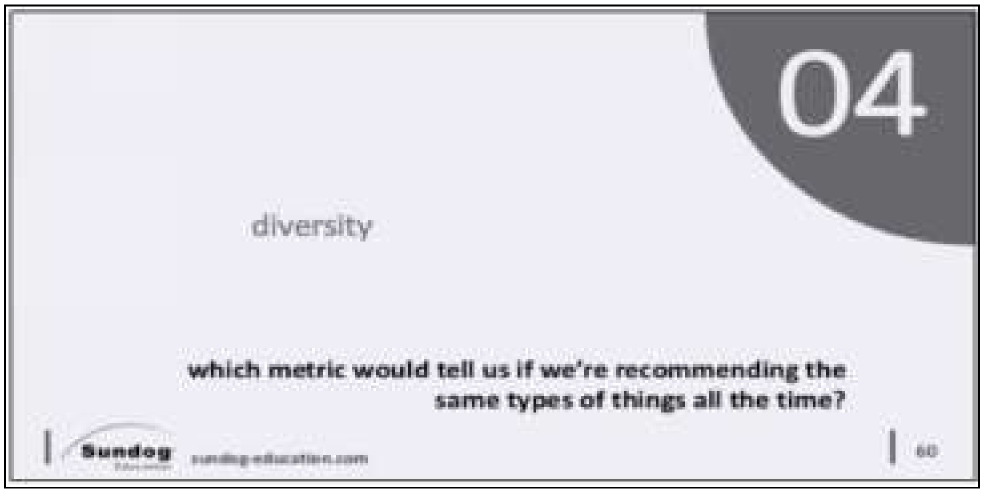
The answer is novelty, which is just a measure of the popularity rank of the items we recommend on the whole. And again, a balance must be struck between recommending popular items that establish trust in the recommender system, and more obscure items that may lead to serendipitous discovery from the users.


04

which metric would tell us if we're recommending the same types of things all the time?

Sundog sundog-education.com 59

Which metric tells us whether we're recommending the same types of things all the time? For example, recommending every Star Wars movie because I watched one of them probably isn't a very serendipitous experience for the user.



04

diversity

which metric would tell us if we're recommending the same types of things all the time?

Sundog sundog-education.com 60

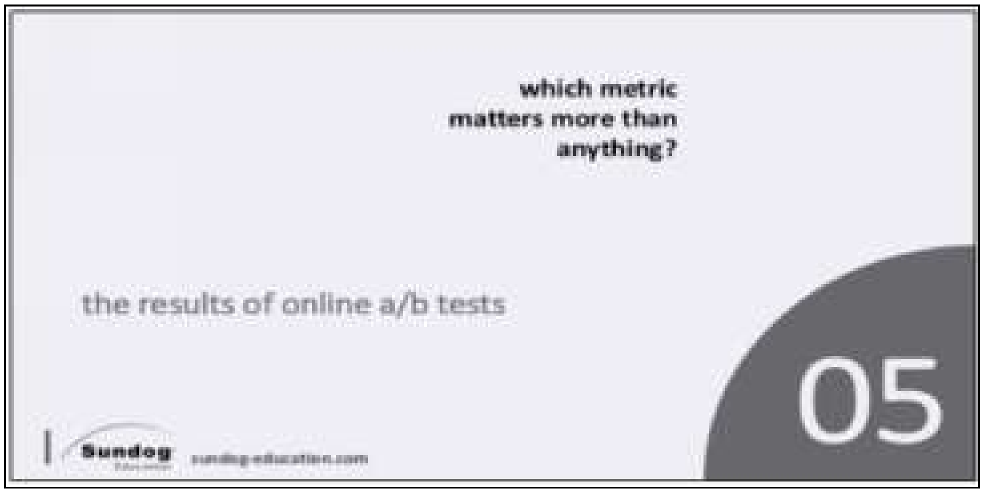
The answer is diversity, which is one minus the average similarity score between the items you recommend to each user. And again, you need to be careful with this metric. Never look at diversity alone, because you can get the highest diversity score by just recommending random stuff to people – and that's not a good recommender system. High diversity is often just a sign of bad recommendations.

which metric
matters more than
anything?

05

Sundog Education sundog-education.com

Finally, which metric matters more than anything? It bears repeating!


which metric
matters more than
anything?

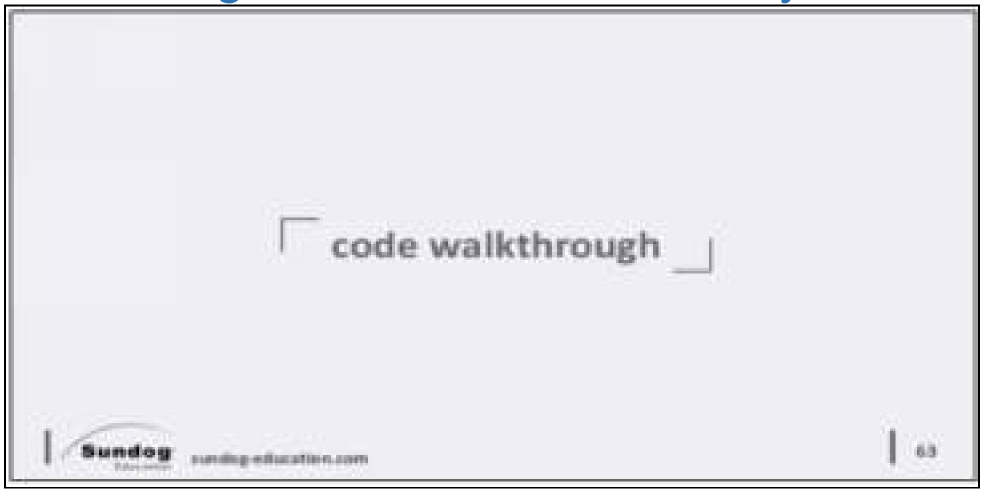
the results of online a/b tests

05

sundog-education.com

That’s the results of online a/b tests. None of the offline metrics we discussed can take the place of measuring how real users react to the recommendations you put in front of them. You just can’t simulate what goes on inside the heads of real people when they see new things – you have to try out your changes on them in the real world, and in a controlled manner, to see what works and what doesn’t.

Measuring Recommenders with Python


code walkthrough

Sundog sundog-education.com 63

So it’s time to start writing some code! We’ve talked about a lot of different ways to evaluate recommender systems, so let’s start by writing the code to make those metrics reality. Think of it as test driven development – we’re going to write our tests before we write any actual recommender systems, and that’s generally a good idea so that you focus on the results you want to achieve. We’re going to use an open source Python library called Surprise to make life easier – you should have already installed that back in the first lecture. Let’s take a look at how it works.

Let’s start by looking at SurpriseLib’s documentation online. Surprise is built around measuring the accuracy of recommender systems, and although I’ve said repeatedly that this is the wrong thing to focus on, it’s really the best we can do without access to a real, large-scale website of our own. And it can give us some information about the properties of the algorithms we’re going to work with. It also saves us a lot of work – for example, it can compute things like MAE and RMSE for us. Let’s have a look at the documentation on the accuracy metrics SurpriseLib offers .

So go ahead and start Spyder in our RecSys environment. Now, under the “Evaluating” folder in your course materials, open up RecommenderMetrics.py and let’s see what it does.

RecommenderMetrics is a Python package that just contains functions that we can use in code elsewhere. So what we're building is a library of useful functions for evaluating recommender systems.

From the SurpriseLib web site, we can dig into Github to see their actual implementations of MAE and RMSE. There's no point in coding this ourselves when somebody else has already done it and tested it for us.

Let's go back to our RecommenderMetrics package. First, I want to stress again that this course isn't setting out to teach you how to write code in Python. As such, we're not going to step through every line of code and talk about what it does, and I'm not going to make you watch me type out thousands of lines of code and make you follow along. Those are great techniques for learning how to code, but my assumption is that you're already familiar with Python, so we can just focus on the algorithms and architecture instead. If it makes zero sense to you, you really should go take an introductory Python course first. That said, let's dive into what's going on in this package.

Since Surprise is concerned mostly about making rating predictions, we have to write some code to get top-N recommendations out of it. That's what this GetTopN( ) function does. It takes in a complete list of ratings predictions that come back from some recommender, and returns a dictionary that maps user ID's to their top N ratings. We can also pass in a minimum rating threshold, to prevent us from recommending items we don't think the user will actually love. That dictionary we return is actually a defaultdic t object, which is just like a normal Python dictionary but it has a concept of default, empty values when you try to access a key that hasn't been used yet.

To compute hit rate, we need to pass in both our dictionary of top-N movies for each user ID, and the set of test movie ratings that were left out of the training data set. If you remember, we're using a technique called "leave one out cross validation" to hold back one rating per user, and test our ability to recommend that left out movie in our top N lists.

Cumulative hit rate (or CHR) works exactly the same way, except we now have a rating cutoff value so we don't count hits unless they have a predicted rating that's higher than some threshold we choose. You can see where we test for that cutoff on line 57, but otherwise this is the same as the hit rate function.

We'll also implement rating hit rate, or RHR, since we talked about that too. This works just like hit rate, but we keep track of hit rate for each unique rating value. So instead of one variable to keep track of hits and the total number of users, we use another dictionary to keep track of the hits and totals for each rating type. Then at the end, we print them all out.

The other twist on hit rate is average reciprocal hit rank, which again is similar to hit rate – the difference is that we count things up by the reciprocal of the ranks of each hits, in order to give more credit for hits that occur near the top of the top-N lists.

Next we can measure coverage, and here we are defining coverage as the percentage of users that have at least one "good" recommendation above some threshold. In the real world, you would probably have a catalog of items that is larger than the set of items you have recommendations data for, and would compute coverage based on that larger set instead. But for the data we have to work with in MovieLens, this is the most interesting measure of coverage I can come up with. The code itself is straightforward.

Next we'll tackle diversity. To measure diversity, we not only need all of the top-N recommendations from our system, we need a matrix of similarity scores between every pair of items in our data set. Although diversity is easy to explain, coding it is a little tricky. We start by retrieving the similarity matrix – this is basically a 2x2 array that contains similarity scores for every possible combination of items that we can quickly look up. Then, we go through the top-N recommendations for each user, one user at a time. This itertools.combination s call gives us back every combination of item pairs within the top-N list. We can then iterate through each pair, and look up the similarity between each pair of items. This is as good a time as any to mention that Surprise maintains internal ID’s for both users and items that are sequential, and these are different from the raw user ID’s and movie ID’s that are in our actual ratings data. The similarity matrix uses those inner ID’s, so we need to convert our raw ID’s to inner ID’s before looking up those similarity scores here. We add up all those similarity scores, take the average, and then subtract it from one to get out diversity metric. As you can imagine, running this code across every possible combination of recommended items for every user in our data set is pretty intensive computationally. In the real world on larger data sets, it might make sense to sample your data when computing this.

Finally we’ll tackle Novelty, which is a lot easier. We’ll take in a handy dictionary of popularity rankings of every item as a parameter, and then it’s just a matter of going through every user’s top-N recommendations and computing the average of all the popularity rankings of every item that was recommended .

Alright, so now we have the tools we need to evaluate recommender systems. Next, let’s try them out on a real recommender!

code walkthrough

Sundog sundog-education.com 64

So we’ve made a Python package with lots of useful functions for evaluating recommender systems. Let’s see them in action, using a real recommender system trained with real movie ratings from the MovieLens data set.

Again, we are using the “Evaluating” folder in our course materials for this section.

Let’s open up TestMetrics.py and MovieLens.py. We start off in TestMetrics by importing the modules we need, including our MovieLens module. Take a quick look at MovieLens.py to see what it does, and how it works. It’s responsible for loading up the raw MovieLens files that contain ratings and information about the movies, and converting them into data sets that Surprise can use. It also contains some useful functions for quickly looking up movie titles, and other utility functions we’ll use later .

OK, so what we’re going to do is create a recommender system around the SVD algorithm, and evaluate it across the many metrics we have designed. How SVD works isn’t important right now, we’re just seeing if our metrics work.

We start by loading up the MovieLens data as we saw before, and the popularity ranks so we can later compute novelty.

This next block builds up the item-to-item similarity scores that we need to compute diversity. It turns out that SVD does not compute this as part of what it does, so to cheat we’re going to train a different kind of recommender called KNN that does compute item similarity scores for you. We get the complete data set from MovieLens, set a few parameters on how those similarity scores will be computed, and fit a KNNBaseline recommender in Surpriselib to our MovieLens data set. The resulting algorithm can later be used to retrieve that 2x2 matrix of item similarity scores.

Just to keep our runtimes reasonable, we’re not going to do full-blown k-fold cross validation – we’ll just do a single train/test split. In our example, we will randomly set aside 25% of our data for testing, and use the remaining 75% to train our SVD recommender with.

And we do just that – create a new SVD recommender algorithm, with a fixed random seed so we get consistent results, and then train the algorithm using the fit function here.

We then process our test set on this algorithm, which gives us back a set of rating predictions for all of the test ratings it was fed. Now, our job is to measure how good those predictions are .

We'll start by using the RMSE and MAE functions in our RecommenderMetrics package to measure prediction accuracy. Then, we'll turn our attention to top-N recommendations instead.

Remember, top-N recommendations need to be tested differently, with "leave one out cross validation." Here we're setting up leave one out with just a single train/test split, again in the interest of time. What this does is set aside one rating per user, selected at random, for the test set. Our challenge is to predict top-N recommendations that include the left-out movie. Since leave one out works on a per user basis, the way the train and test data sets are created is different and we have to perform that split again.

We iterate through each train/test split from our cross validator, which in our case will only be a single split. Then we train our SVD algorithm all over again with our new training set, and test it again with our test set of left-out ratings for each user.

Now, the other thing we need is the actual complete list of top-N recommendations for each user that we want to evaluate. To do that, we need to get rating predictions for every movie each user has not already rated, and then rank them by predicted rating. By the way, you might remember that this is one of two possible architectures for a top-N recommender system, and it's not the more efficient of the two. Don't worry, we'll do it the other way as well in a bit.

But the thing that's a little tricky to wrap your head around is that we need what's called an "anti test set" to create our top-N lists for each user. That is, we only want to consider ratings for movies that users haven't already seen – so we build up our test set for all possible user, movie pairs that are not in the training set, and restrict ourselves to those .

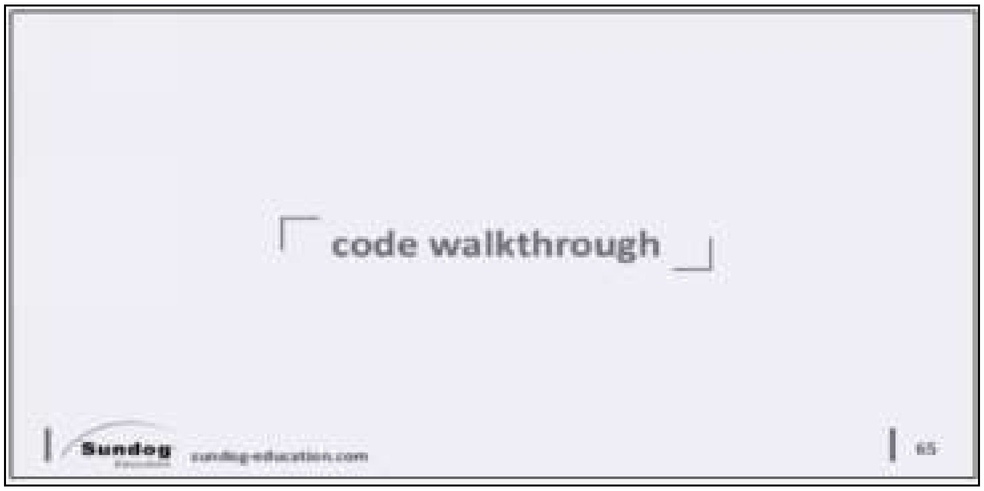
Once we have that set of predictions, we can use the GetTop N function we wrote earlier to produce top-N lists for each user, and now we can just go to town evaluating them.

We’ll do it all here - hit rate, then rating hit rate, then cumulative hit rate, then average reciprocal hit rate, and print out the results.

Next, we want to measure coverage, diversity, and novelty. These are all properties of the top-N recommendations as a whole, and don’t involve measuring accuracy between a train set and a test set. So to get the values we want, we need to train our model yet again – this time using the entire data set, and not holding anything back for testing. But again, we want to exclude recommendations for movies the users have already watched, so we actually use an “anti test set” on the full training set here to use for figuring out the top N for each user.

Once we have that, we can compute coverage, diversity, and novelty as we discussed before.

So, now that we understand what this code does, let’s kick it off and see what happens!



code walkthrough

Sundog
sundog-education.com
65

So it took a few minutes, but we got some valid results here for our SVD recommender!

Our RMSE score is about 0.9, and MAE is about 0.7. That means that on average, our guess of a rating for a given movie for a given user was off by 0.7 stars. RMSE is higher, meaning that we got penalized for being way off more often than we'd like. Again, remember higher error metrics are bad – you want these values to be as low as possible, if accuracy is your goal.

Our hit rate was just about 3% - which actually isn't that bad considering that only one movie was left out from each user's ratings to test with. But this number by itself is going to be hard to interpret as good or bad until you have other recommenders to compare it against.

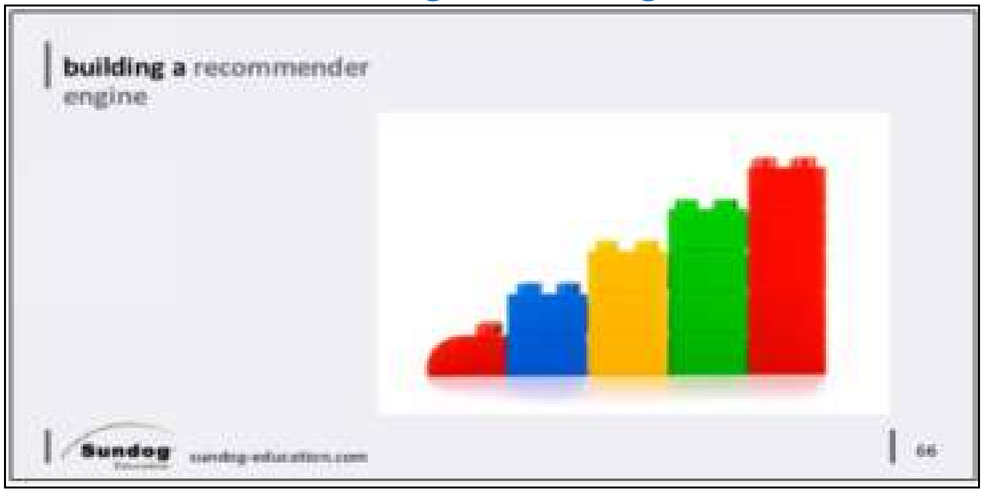
If we break it down by rating value, you can see our hit rate did better at higher rating predictions, which makes sense and is what we want to see. Cumulative hit rate with a 4.0 threshold isn't much lower that the raw hit rate, meaning that we're doing a good job of recommending items we think are good recommendations. ARHR is 0.1, and again it's just a number. It takes the ranking of this hits into account, but honestly for such a small data set we're probably looking for more data than really exists for this number to be very meaningful. And again it has no real value until we have other recommender systems to compare it against.

Finally, we look at user coverage for which we had at least one 4-star rating prediction, and it's pretty high – that's good.

But let's look at our diversity and novelty scores. Diversity is really high at 0.96, and novelty seems pretty high with an average popularity rank of 491. Remember there are only a few thousand movies in our data set to begin with. This tells us that our algorithm is going pretty deep into the "long tail" to get its recommendations, and that could be a problem in the real world. So even though our accuracy metrics look OK here, diversity and novelty is telling us that we're recommending a lot of really obscure stuff, and that could be an issue when trying to establish user trust in the system. People generally want to see a few things that at least look familiar, so I would expect that this particular algorithm wouldn't do that well in an online A/B test. But you can never know until you actually try it.

Anyhow, we know we have working code now for evaluating real recommender systems, and we’ve seen how to use it. So this will be an important foundation that we build the rest of the course on top of. We’re just getting started guys, so keep going.

Recommender Engine Design



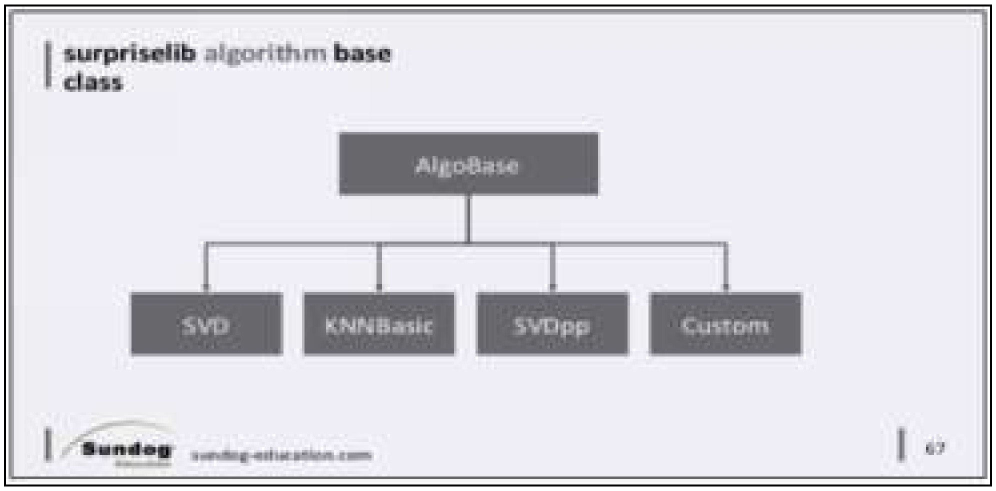
building a recommender engine

Sundog sundog-education.com

66

So, we've got this nice code to evaluate recommender systems, and we intend to try out quite a few different ones throughout the rest of this course. What we need now is a framework to let us easily experiment with new recommender system algorithms, evaluate them, and compare them against each other.

SurpriseLib provides us with some of what we need, but we're going to build on top of it. Let me show you the various pieces we're going to use and how they fit together.



surpriselib algorithm base class

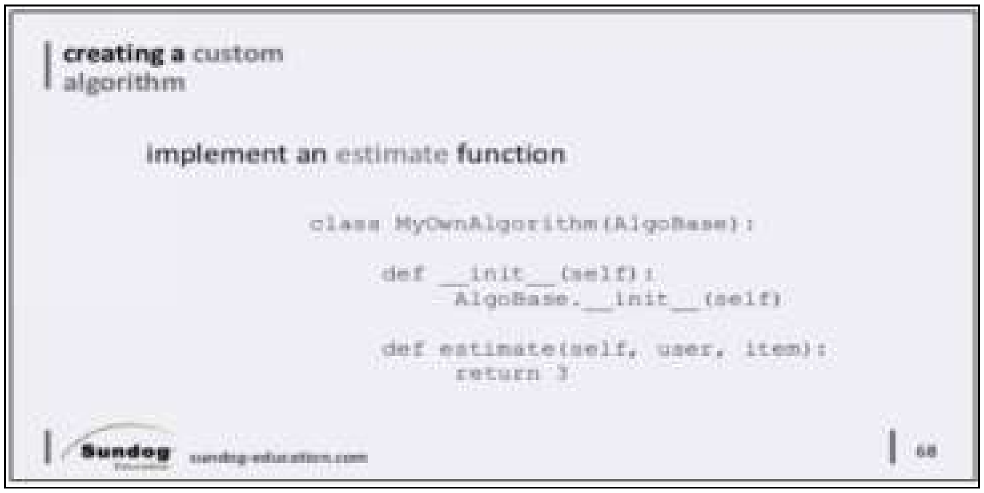
Sundog sundog-education.com

67

Surpriselib has a Python base class called "AlgoBase." If you don't know what a base class is, then you're probably new to object oriented design. It's an easy concept, though. Fundamentally, a class is just a collection of functions organized under some name. That class can also have variables associated it. And, you can instantiate an instance of a class object, which lets you keep around a copy of a class's functions and the data in its variables under a single name you can work with. It'll make more sense when we look at some code, trust me.

Object oriented design allows us to have base classes, for example "AlgoBase," that contain functions and variables that can be shared by other classes that inherit from that base class. So for example, here we have an SVD class which inherits from the AlgoBase base class. That means that our SVD class may have functions and data specific to the SVD recommender algorithm, but it also has access to the functions and data defined in the AlgoBase base class. It's just a way to maintain consistent interfaces with all of our different recommender algorithms. For example, the AlgoBase base class defines the function names "fit" and "test", which are used to train and to test each algorithm. So, no matter what the actual algorithm may be, we know we can always call "fit" or "test" on it. SVD, KNNBasic, SVDpp, and any custom algorithm we might develop that inherits from "AlgoBase" can all be used in the same way. This makes things a lot easier when you are trying to train and evaluate many different algorithms, since you can use the same code to do so.

These are just a few of the recommender algorithms that are build into surpriselib. "Custom" isn't one they offer, but it's just my way of illustrating that you may write your own recommender algorithms and make them part of the surpriselib framework – you're not limited to the ones they provide.


creating a custom algorithm

implement an estimate function

```
class MyOwnAlgorithm(AlgoBase):

    def __init__(self):
        AlgoBase.__init__(self)

    def estimate(self, user, item):
        return 3
```

Sundog sundog-education.com 68

So, how do you write your own recommender algorithm that's compatible with surpriselib? It's surprisingly easy (see what I did there? Surprisingly easy? I kill me.)

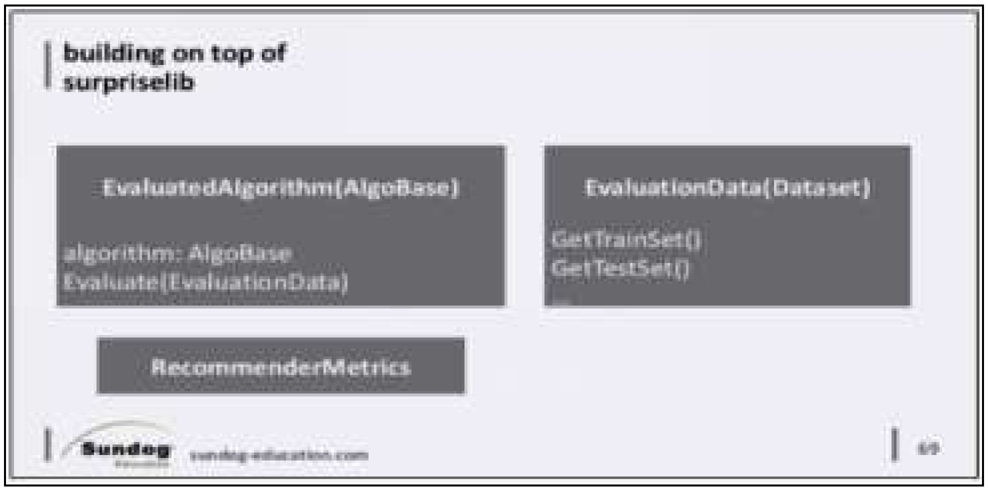
All you have to do is create a new class that inherits from AlgoBase, and as far as surpriselib is concerned, your algorithm has one job: to predict ratings. As we mentioned, surpriselib is built around the architecture of predicting the ratings of every movie for every user, and giving back the top predictions as your recommendations. It's not ideal, but it's necessary for evaluating accuracy offline.

So your class needs to implement an estimate function. That's it. Let's look at the entire code for a custom recommender algorithm that does this… yeah, that's all there is to it.

The estimate function takes in three arguments: the first, "self", is the first argument in any class method in object oriented Python. It's a reference to the instance of the object that is currently being worked with, so you can access variables associated with that instance. Next it takes in a user ID and an item ID. So, when estimate is called by the surpriselib framework, it's asking you to predict a rating for the user and item passed in. These user and item ID's are inner ID's by the way – ID's that are used internally, and must be mapped back to the raw user and item ID's in your source data.

In this simple example, our new recommender algorithm called MyOwnAlgorithm just predicts a rating of 3 stars for absolutely everything and everybody. I doubt it would perform very well, but this as simple as it gets! Note there is also some boilerplate code for initializing the MyOwnAlgorithm class object, which just calls the base class's ini t function – but if you had to do some setup in preparation for doing something more complicated than just predicting "3" all the time, that might be a good place to do it.

Make sense? If not, stare at this code a bit until it does. It's kind of important.


building on top of surpriselib

EvaluatedAlgorithm(AlgoBase)

algorithm: AlgoBase
Evaluate(EvaluationData)

EvaluationData(Dataset)

GetTrainSet()
GetTestSet()

RecommenderMetrics

Sundog sundog-education.com

69

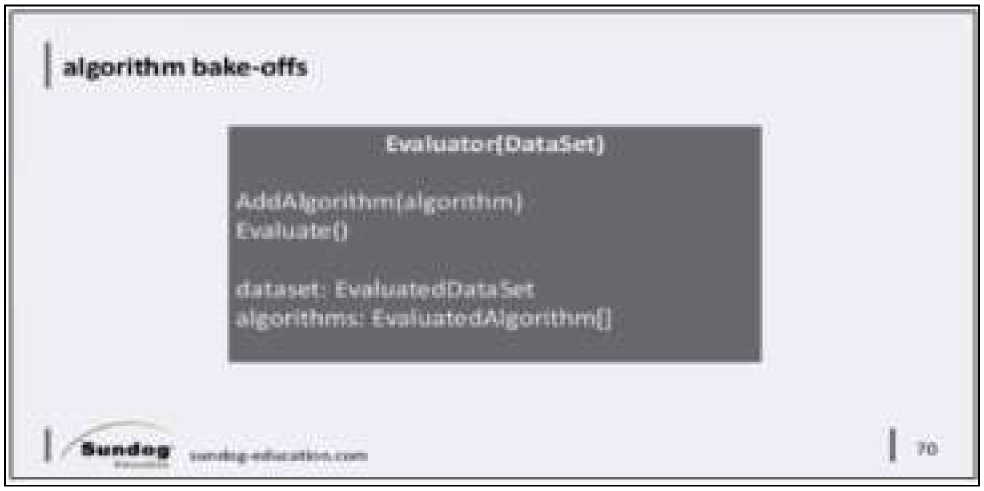
Now, we want to do more than just predict ratings. We want to make it easy to apply all those different evaluation metrics we implemented earlier in our

RecommenderMetrics class to the algorithms we work with. So to do that, we're going to create a new class called EvaluatedAlgorithm. It contains an algorithm from surpriselib, but introduces a new function called Evaluate that runs all of the metrics in RecommenderMetrics on that algorithm. So, this class makes it easy to measure accuracy, coverage, diversity, and everything else on a given algorithm. It uses the functions in the RecommenderMetrics class we wrote already to do this, so we don't have to re-write all of that code.

Now, to evaluate a recommender system in several different ways, we'll want to slice up our training data into train and test splits in various different ways. That's what our EvaluationData class is for –

it takes in a DataSet, which might come from our class that loads MovieLens data for example, and creates all the train/test splits needed by our EvaluatedAlgorithm class .

So you can see how this all starts to fit together. We can start by creating an instance of EvaluationData for a given data set, which will slice and dice that data in all the ways needed to evaluate it. Then we can test many different algorithms using that same evaluation data, by wrapping those algorithms with an EvaluatedAlgorithm instance. We then call the Evaluate function on EvaluatedAlgorithm, passing in our EvaluationData, to measure the performance of that algorithm. See how it all connects? We create an EvaluationData instance with our data set, create an EvaluatedAlgorithm for each algorithm we want to evaluate, and call Evaluate on each algorithm using the same EvaluationData. Under the hood, EvaluatedAlgorithm will use all the functions we defined in RecommenderMetrics to measure accuracy, hit rate, diversity, novelty, and coverage.


algorithm bake-offs

Evaluator(DataSet)

AddAlgorithm(algorithm)
Evaluate()

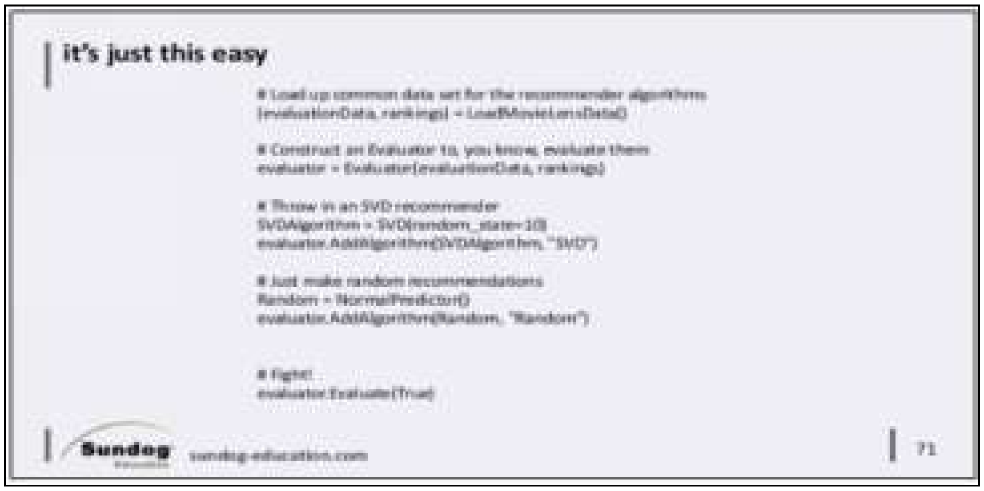
dataset: EvaluatedDataSet
algorithms: EvaluatedAlgorithm[]

Sundog sundog-education.com 70

Since what we generally want to do is compare different recommender algorithms against each other, we can make life even easier by writing a class that takes care of all that comparison for us. Ideally, we’d like to just submit algorithms we want to evaluate against each other into this class, and let it do everything from there. That’s what our Evaluator class does .

Our Evaluator class just takes in a raw dataset, say from MovieLens, and the first thing it does is create an EvaluatedDataset object from it that it uses internally. Then, you just call AddAlgorithm() for each algorithm you want to compare – this creates an EvaluatedAlgorithm under the hood within the Evaluator, and adds it to a list internally called “algorithms”. Finally, when you call Evaluate(), it evaluates each EvaluatedAlgorithm, and prints out the results from each one in a nice tabular form.

So, the beauty of this is that you don’t even have to use the EvaluatedAlgorithm or EvaluatedData classes at all when you want to start playing around with new algorithms and testing them against each other. All you need to do is use this Evaluator class, which has a really simple interface.



it’s just this easy

```
# Load up common data set for the recommender algorithms
(evaluationData, rankings) = LoadMovieLensData()

# Construct an Evaluator to, you know, evaluate them
evaluator = Evaluator(evaluationData, rankings)

# Throw in an SVD recommender
SVDAlgorithm = SVD(random_state=10)
evaluator.AddAlgorithm(SVDAlgorithm, "SVD")

# Just make random recommendations
Random = NormalPredictor()
evaluator.AddAlgorithm(Random, "Random")


# Fight!
evaluator.Evaluate(True)
```

Sundog sundog-education.com 71

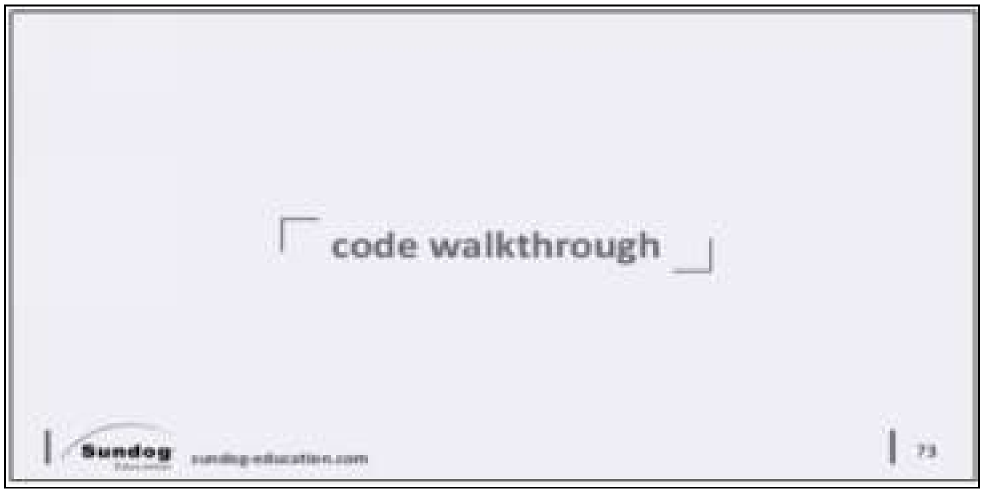
The reason we’re going through all the trouble of developing this framework is so we can do this – this is all the code it takes to compare the SVD algorithm against the Random algorithm, run every evaluation metric we have on both, and compare the two .

All we do is load up our MovieLens data set, and pass that into our Evaluator. Then we call AddAlgorithm for both our SVD and Random algorithms, call Evaluate(), and that’s it!

let's jump in

Sundog sundog-education.com 72

Let's dive into the code so we can see what it's doing in more detail, and run it and see how it works!



code walkthrough

Sundog sundog-education.com 73

At this point you're probably getting a little impatient to start delving into actual recommender algorithms, so I'm not going to spend a ton of time dissecting the code of our recommender system framework. But let's do a quick tour.

Start by launching Spyder from your RecSys environment, and open up all of the files inside the Framework folder inside the course materials.

Let's work backwards, and start from our script that actually uses and tests out this framework. That would be the RecsBakeOff.py file.

We start by importing all the modules we need – notably, all we need are the recommender algorithms themselves that we want to play with, our MovieLens module that loads our training data, and the Evaluator class we talked about. So, the only part of our framework that we really need to understand is the Evaluator module… it hides the complexity of the other modules from us, so we don't have to worry about how our EvaluatedAlgorithm or EvaluationData modules work.

The code here is pretty simple, although there are a few finer points that we didn't mention in the overview. First we have our LoadMovieLensData function, which does what it says – it uses our MovieLens module to load up a surpriselib dataset containing our movie ratings data that we'll use for training and testing each algorithm, and it also gives us back a dictionary of the popularity rankings of each movie. We need those rankings in order to compute novelty.

We've seen the MovieLens module before, but if you want to study its inner workings again, feel free to go through its code again. Data processing is often half the battle with recommender systems, and machine learning in general .

We set some random seed values, which just ensures that we get the same results every time. Then we load up our data here.

Now, we just create an Evaluator object using the MovieLens data we just loaded up. We create an SVD recommender algorithm and a random recommender algorithm as sort of a baseline, add them both into the evaluator, and then call Evaluate on it.

Really simple, right? That's the whole point. We want to make it easy to evaluate new algorithms so that in the rest of this course, we can just focus on the algorithms themselves.

So let's dive into the Evaluator module we're using here. Open up Evaluator.py to see what it does.

You can see that internally it maintains a list of algorithms – these are actually EvaluatedAlgorithms, as we'll see.

When we initialize an Evaluator, the first thing it does is take the raw dataset and popularity rankings you feed it and convert it into an EvaluationData object instead. This takes that raw data, and makes all the different train/test splits we'll need to evaluate it with. This gets stored as self.dataset – in Python, when you want to use a variable associated with a class instance, you always refer to it with "self".

Next we have AddAlgorithm, which just takes a raw surpriselib algorithm and converts it into an EvaluatedAlgorithm, then adds that to the list of algorithms we want to evaluate later on.

And, that's what Evaluate does. It just calls Evaluate on each EvaluatedAlgorithm that you gave it, and tabulates the results and prints them out nicely. Notice the " doTop N " parameter – this allows us to bypass the computation of hit rank metrics if we want to, because they can be very computationally expensive to produce.

When it's done, it just prints out a nice little legend reminding you what all of these different metrics really mean.

One more thing – we also have this SampleTopNRecs function here. Sometimes it's helpful to look at the actual recommendations being produced for a known users whose tastes you understand at a more intuitive level. It can be a helpful sanity check. So, this takes in a MovieLens data set so we can convert movie ID's into titles that actually mean something to you, a user ID that you want to get recommendations for, and how many recommendations you want to get.

So for each recommender, let's look at what's going on. First, we train the recommender algorithm using the full, complete training set, since we're not doing train/test here – we just want the best recommendations we can get.

What we do next is get an “anti-test set” for the user we’re interested in. All this does is produce a test data set of all of the movies that this user did not rate already, since we don’t want to recommend movies the user has already seen.

Next, we generate rating predictions for every single movie this user hasn’t seen, and then we sort them all by their predicted ratings. We pluck off the top N results, and print them out. Again, this isn’t the most efficient way to do it, but for evaluation purposes it’s a good way to go.

So we’ve seen that Evaluator uses both EvaluatedAlgorithm and EvaluationData. Let’s dive into EvaluatedAlgorithm first, and open that up. There’s not actually a whole lot to talk about here – it’s really just a wrapper for a bunch of calls to the RecommenderMetrics module we covered earlier. You can see that the Evaluate method has an option for doing top-N hit rate, diversity, and novelty metrics, so you can avoid the expense of computing these if all you’re interested in is accuracy. It puts all of the evaluation results into a dictionary called “metrics”, which the Evaluator then uses to print out the results of each EvaluatedAlgorithm side by side.

If you’re new to Python, I encourage you to pause and spend some more time studying this code as it may help you understand the language. But in the interest of getting into actual recommendation algorithms more quickly, let’s move on.

Open up EvaluationData.py and have a look. As you can see, it’s just generating all the various train/test splits required by our RecommenderMetrics functions. It tucks away a copy of the ranking data needed for novelty, then proceeds to slice and dice our data in various ways. First it creates a full training set used for analyzing the data as a whole, and a full anti-test set, which contains all of the missing ratings for movies users have not yet rated that need to be predicted.

Next we do a 75/25 train/test split of the data, used for measuring accuracy. Again, using k-fold cross validation might be a better idea, but it’s not really worth the extra time it takes for us.

Next we generate the train/test split needed for measuring hit rates. Remember we need to use “leave one out” cross validation, to withhold one known rating for each user so we can see if we can successfully recommend that movie in our results or not.

Finally, we create a similarity matrix that gives us similarity scores between every possible combination of movies – we need this for computing diversity. To do this, we take advantage of surpriselib’s built-in KNNBaseline algorithm, which computes this similarity matrix as part of what it does. So by training a KNNBaseline with our data, we can access that similarity matrix under its hood later on.

Then we just have a bunch of accessors for all the various splits we created. The only interesting bit of code here is GetAntiTestSetForUser, which gives back a list of empty prediction values for every movie a given user hasn’t rated already. This somewhat cryptic code on line 49 is just extracting a list of the items this user has rated already from the full training set. Then on line 50, we construct a bunch of prediction structures, each consisting of an inner user ID, an inner item ID, and a placeholder rating which for now is just the global mean of all ratings. But we check against that list of stuff the user has already rated, and exclude those to get our anti-test set for that user.

OK, that’s it! So let’s get some payoff from all this and see it in action. Open up RecsBakeoff.py again, and let’s see how the SVD recommender does against random recommendations. Click play to let it do its thing, and go get a cup of coffee, catch up on your email, or something… it will take several minutes to complete. Me, I’m going to get some lunch. So come back when it’s done running.

OK, so you can see the final results here, and they do seem to make sense… remember we’re comparing SVD, or singular value decomposition, against random recommendations here, and SVD is one of the best algorithms available right now. So it shouldn’t be a surprise that SVD beats random recommendations in accuracy and hit rate no matter how we measure it.

Remember lower RMSE and MAE scores are better, as they are measuring errors. This can be a little confusing since we talk about these metrics in terms of accuracy, and you'd think that more accuracy is better – but the actual numbers for root mean squared error or mean absolute error are measuring error, the opposite of accuracy. So it's good that SVD has lower RMSE and MAE than the random recommender.

Higher hit rates, however, are better. And no matter how we measure it – hit rate, cumulative hit rate, or average reciprocal hit rate – SVD does a better job. The actual numbers here are still quite small, however, and that's to be expected. It's really hard to get a hit with leave-one-out cross validation, especially when you have a lot of items to choose from.

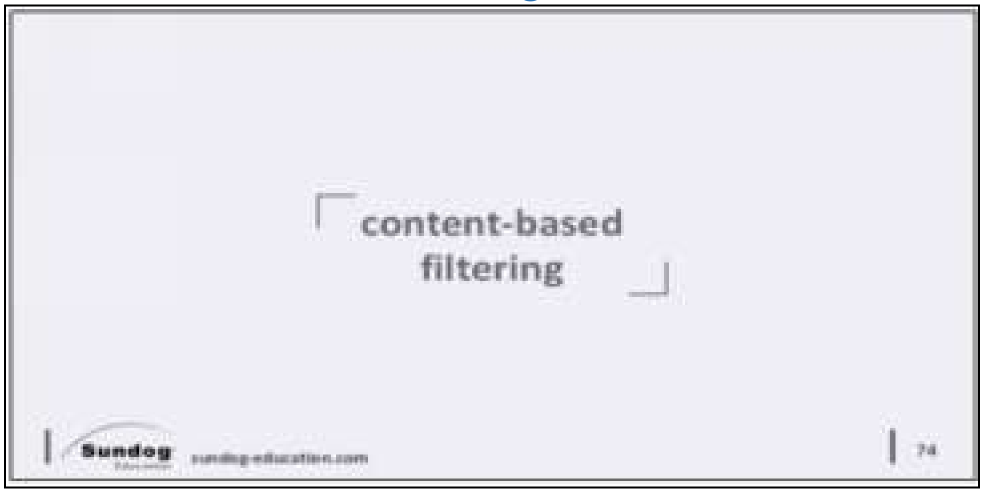
Now we get into coverage, diversity, and novelty – and these metrics are ones you need to apply some common sense to, as it's not a clear "higher is better" sort of thing. There are tradeoffs involved with these metrics, remember. Coverage for example is a bit lower, but that's just because we're enforcing a quality threshold on the top-N recommendations we're making with SVD, while our random recommender isn't actually making totally random recommendations – it's predicting movie ratings using a what's called a normal distribution centered around the average rating value, which ends up meaning all of the rating predictions it makes fall above our rating threshold, giving us 100% coverage. Having 100% coverage at the expense of having bad recommendations isn't a tradeoff worth making.

Diversity is also lower with SVD, but you can see this as a good and expected thing – since people tend to have somewhat consistent interests, you'd think that good recommendations for a specific user would be more similar to each other than ones pulled at random that are not personalized in any way.

Finally novelty is lower with SVD, but again that’s good and expected. You would expect that just picking random movies to recommend would give you a high novelty score since it’s pulling all sorts of movies out of the long tail, while SVD would end up discovering that good movie recommendations tend to be more popular movies. Popular movies are popular for a reason – they appeal to a lot of people, and are good recommendations for a lot of people if they haven’t seen them yet. So again, a lower novelty score here isn’t a bad thing, it’s just a measure of the nature of the recommendations you’re making and how obscure they are.

But anyway, it works! We have a framework now that will let us test and evaluate new recommendation algorithms as we go through them, and it makes it really easy to do so. A framework like this is something you can also use in the real world, as you’ll often want to prototype new ideas without necessarily evaluating them online right away – and it’s also useful for combining recommendations from multiple recommendation algorithms together, which can sometimes be a very powerful technique. With all of this out of the way, we can finally start experimenting with different algorithms, and go into more depth about how they work.

Content-Based Filtering



content-based filtering

Sundog sundog-education.com

74

It's finally time to start talking about some actual recommendation algorithms! We're going to start with the most simple approach – recommending items just based on the attributes of those items themselves, instead of trying to use aggregate user behavior data. For example, it can be effective to just recommend movies in the same genre as movies we know someone enjoys. Even for more advanced recommender systems that are based on machine learning, baking in some knowledge about the content itself can make them even better .

Attribute-based Recommendations

examples of movie attributes

Sundog sundog-education.com

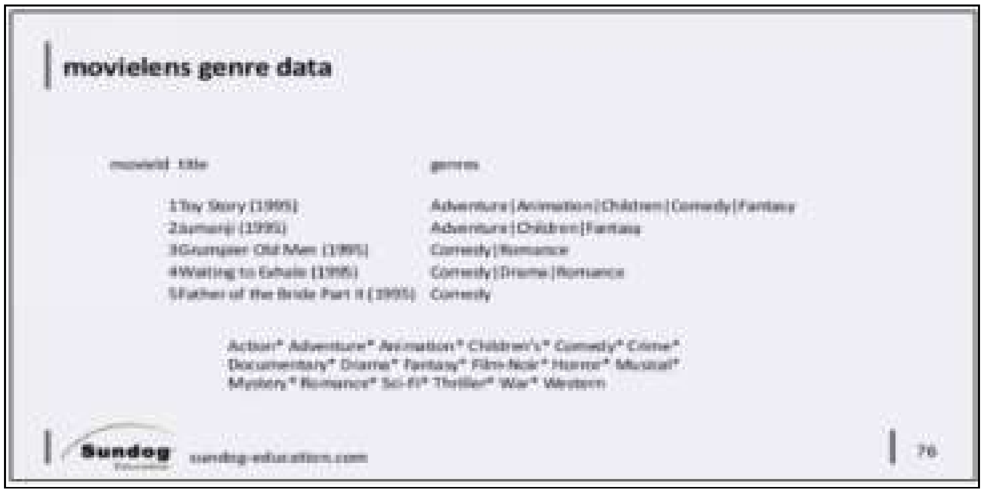
75

So let's think about recommending movies just based on the attributes of the movies themselves. The MovieLens data set doesn't give us much to work with, but one thing it does tell us is which movie genres each movie belongs to. For every movie, we're given a list of genres, like science fiction, horror, romance, children's, westerns, etc. that might apply to that movie. So if we know a given user likes science fiction movies, it's reasonable to recommend other science fiction movies to that user.

MovieLens also encodes the year of release into the movie titles, so we can use that information as well. So instead of just recommending all science fiction movies to a user who likes science fiction, we can narrow it down further to science fiction movies that were released close to the same year as the movies this person liked.

You could get even fancier, and tie each movie in MovieLens to other dataset like IMDb to get information about the directors and actors in each movie, or critic review scores. There are all sorts of attributes about movies at your disposal, and this is true for other things you might want to recommend as well, such as products or music.

So, that's a pretty simple example of doing content-based filtering – we'll just recommend movies that have similar genres and similar years to the movies each user liked. Seems reasonable enough. But how do we do that, exactly?



movielens genre data

```
movieId title                          genres
      1Toy Story (1995)                Adventure|Animation|Children|Comedy|Fantasy
      2Jumanji (1995)                  Adventure|Children|Fantasy
      3Grumpier Old Men (1995)         Comedy|Romance
      4Waiting to Exhale (1995)        Comedy|Drama|Romance
      5Father of the Bride Part II (1995) Comedy
```

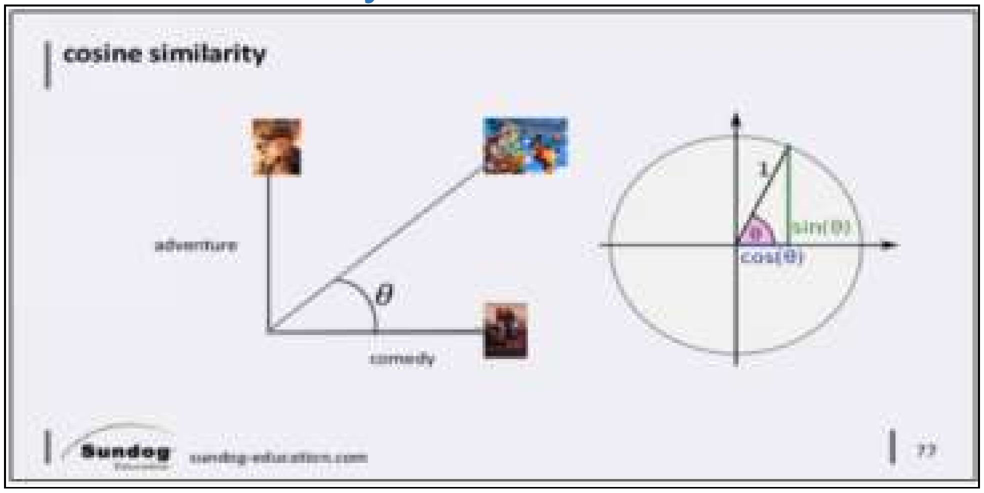
Action* Adventure* Animation* Children's* Comedy* Crime*
Documentary* Drama* Fantasy* Film-Noir* Horror* Musical*
Mystery* Romance* Sci-Fi* Thriller* War* Western

Sundog sundog-education.com 76

Let's start by thinking about how to measure the similarity between two movies just based on their genres. Here's what the raw data from MovieLens looks like – for each movie, we're given a pipe-delimited list of the genres that apply to that movie.

In all, there are 18 different possible genres for every movie. So, we can imagine some sort of similarity measure that looks at how many genres any given pair of movies have in common. So what's a good way of doing that mathematically ?

Cosine Similarity



One approach that works really well in a lot of different situations is the cosine similarity metric. It can be a little tough to wrap your head around, but once you do, it's really simple. And it's not only applicable to content-based filtering, it's applicable to all sorts of other recommendation approaches too.

To make it easier to visualize, imagine we only have two possible genres that every film might belong to: comedy, and adventure. We could then plot every movie on a 2D graph, where we assign a value of 0 if a movie doesn't belong to a genre, and a value of 1 if it does.

So let's start by plotting Toy Story – it's considered both a comedy, and an adventure. So that gives us the coordinates (1,1). Let's plot that.

Next let's look at Grumpier Old Men – according to MovieLens, it's a comedy, but not an adventure. So that gives us the coordinates (1, 0). Here .

Stay with me here. We can think of these positions as vectors, by just drawing a line from the origin of the graph to each movie. Like so.

Now, look at the angle between these vectors – that tells us something about how similar these two movies are in terms of their genres. Let’s call that angle theta. In this case, it’s 45 degrees. But “45 degrees” isn’t a useful similarity metric – ideally we want a measure that’s between 0 and 1, where 0 means “not at all similar,” and 1 means “totally the same thing.”

Well, it turns out the cosine of that angle does just that. Remember the cosine of an angle works like this: as theta approaches 90 degrees, it will become zero, and as it approaches zero degrees, it becomes one. The cosine of 45 degrees is about 0.7, so we could say that the cosine similarity score between Toy Story and Grumpy Old Men is 0.7, based on them both sharing a genre of comedy.

Let’s look at a couple of more examples. Imagine we throw Monty Python and the Holy Grail into the mix, which, like Toy Story, is both an adventure and a comedy. The angle between it and Grumpier Old Men is 45 degrees, just like Toy Story, so its similarity score to Grumpier Old Men is the cosine of 45 degrees, or 0.7. But what is its similarity to Toy Story?

Well, it’s quiz time! What’s the angle theta between the vectors for Toy Story and Monty Python and the Holy Grail?

In this case, the vectors for these two movies is exactly the same. So if you said zero degrees, you’re right. There’s no angle between them at all because they are identical. And the cosine of zero degrees is what?

The cosine of 0 is 1.0 – which is what we’d expect. Since these movies are identical as far as their genres are concerned, they get a perfect similarity score of 1.0 .

What if a movie is completely unlike another? Let’s throw in the movie Indiana Jones, which is an adventure but not a comedy. So it has nothing at all in common with Grumpier Old Men as far as genres are concerned. What's the angle between Indiana Jones and Grumpier Old Men in this case?

It's 90 degrees – and the cosine of 90 degrees is… you guessed it… zero! So we get a cosine similarity score of zero between Indiana Jones and Grumpier Old Men.

Using the cosine has some other nice mathematical properties, since it's nice and smooth as a function. There are other ways to measure similarity too; for example, Euclidean distance just looks at the actual distance between each item as you plot it, and Pearson Correlation is similar to cosine similarity, but it takes the mean values of things into account. In the case of movie genres though, cosine similarity is as good a metric as any.

So how do we scale this up to all 18 genres that we really have? Well, we have to think of each genre as a dimension on that plot. That's right, we need to find the angles between movies in 18-dimensional space, where every dimension represents a movie genre!

There is no way you can actually picture this in your head, so don't even try. But the concept is exactly the same as it was in our 2-genre, 2-dimensional space. We plot every movie's position in 18-dimensional space based on its genres, and figure out the cosine of the angles between each movie as a measure of their genre-based similarity to each other.

convert genres **to** dimensions

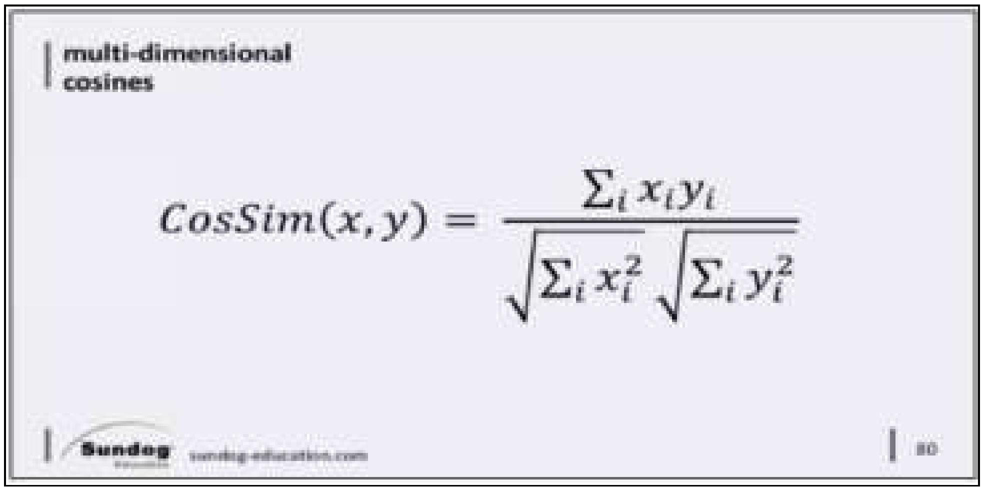
[illegible]

Sundog sundog-education.com

79

So try not think about 18-dimensional space. Just think of every movie being described as a set of 18 coordinates, each of which is 0 if a movie is not of a given genre, and 1 if it is. For example, Toy Story is an adventure, animation, children’s, comedy, and fantasy movie, so if we assign genres to dimensions in the order shown here, Toy Story has a coordinate of 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, and so forth.

That’s not hard, right? We can write code to convert those lists of genres into 18-dimensional coordinates pretty easily.



multi-dimensional cosines

$$CosSim(x,y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

Sundog sundog-education.com

80

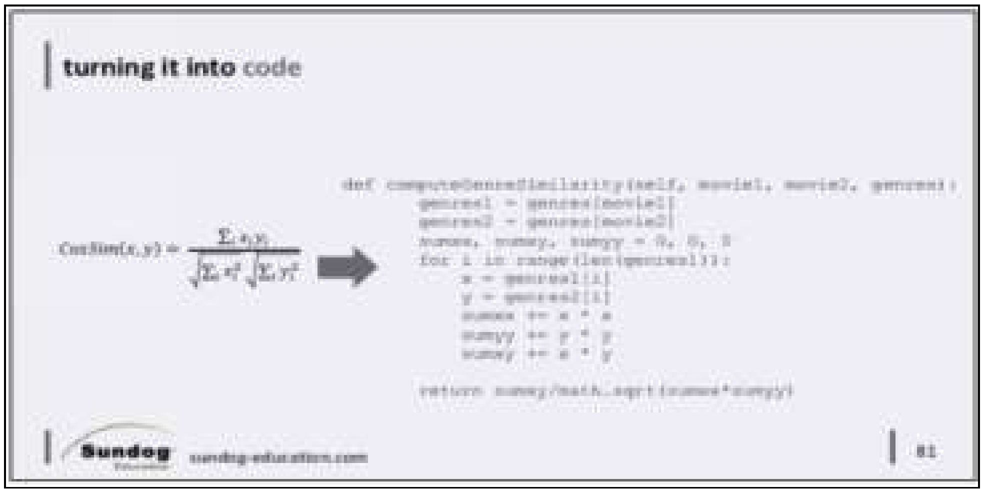
Our next problem is how to compute the cosines between vectors in multidimensional space. It turns out that another nice thing about cosines is that this is pretty simple to do, as well. Here’s the fancy mathematical equation for how to compute the cosine similarity score between any number of dimensions. Let’s break this down.

So to go back to our use case, let’s look at two movies we want to compute the cosine similarity between. We’ll call these two movies X and Y. So, the cosine similarity of x and y is given by this stuff on the right. Don’t let the notation scare you, it’s pretty simple. Let’s start with the top of this fraction. It says to take the sum across i dimensions of the products of x and y in each dimension. So, we go through all 18 dimensions, one at a time, looking only at movies that both have data in this dimension. Say we start with the action dimension, we take the product of movies x and y on the action dimension. Then we add in the product of x and y on the adventure dimension. And so on and so forth .

On the bottom, we sum up all of squares of each genre dimension for the movie X, and we sum up all the squares of each movie dimension of movie Y. For each we take the square root of the value we end up with.

So by taking the summation of the genre products between x and y, and dividing it by the square roots of the summations of the products of each individual movie’s genres, squared, we end up with a cosine similarity score.

This is beautiful, because it’s the sort of algorithm that’s easy to code and that computers are very good at doing.



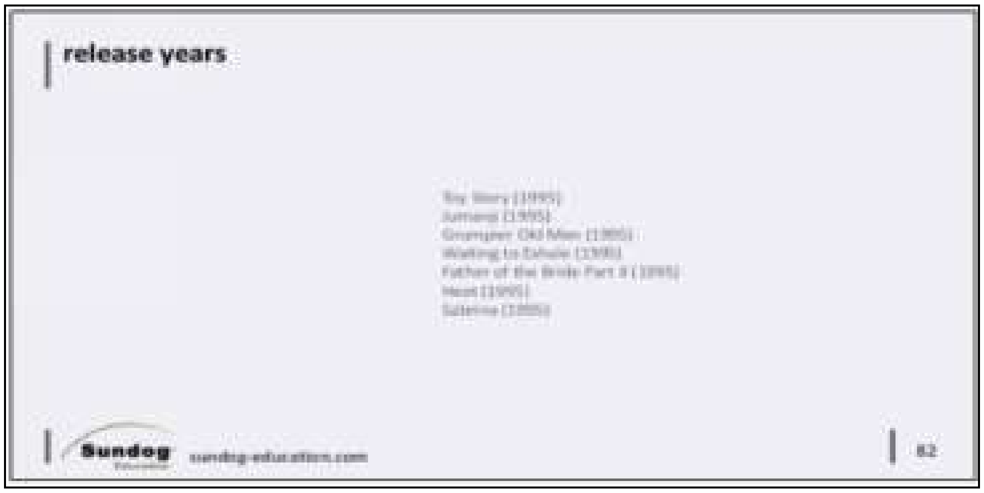
turning it into code

$$CosSim(x,y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

```
def computeGenreSimilarity(self, movie1, movie2, genres):
    genres1 = genres[movie1]
    genres2 = genres[movie2]
    sumxx, sumxy, sumyy = 0, 0, 0
    for i in range(len(genres1)):
        x = genres1[i]
        y = genres2[i]
        sumxx += x * x
        sumyy += y * y
        sumxy += x * y

    return sumxy/math.sqrt(sumxx*sumyy)
```

Sundog Education sundog-education.com 81

In fact, here's the code that actually computes the genre similarity that we'll be working with shortly. This function takes in two movie ID's, and a dictionary that maps movie ID's to their 18-dimensional coordinates in genre space. It just goes through every genre one at a time, extracts the value of the genre for movies x and y, and keeps a running sum of x squared, y squared, and x times y. Then the cosine similarity is just a matter of putting it all together. Note that we simplified the equation in the code a little bit to avoid doing two square roots, since that can be computationally expensive.



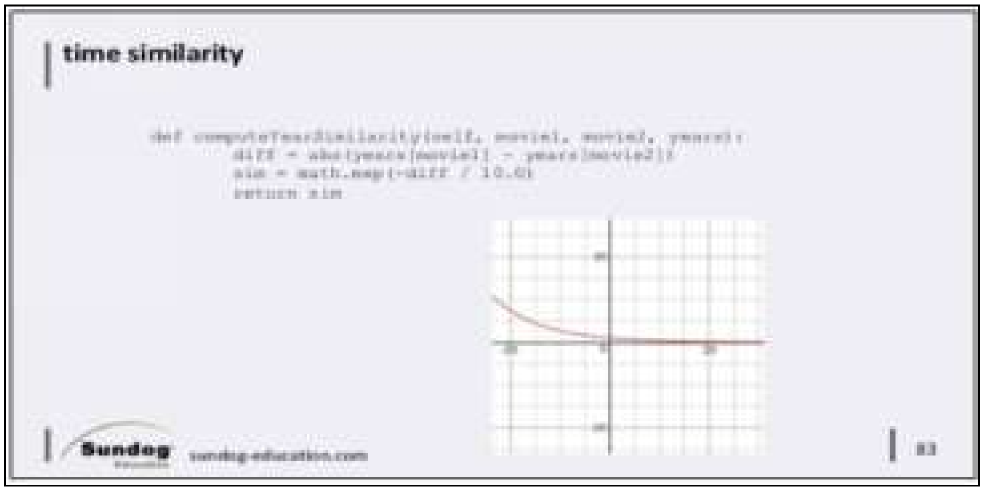
release years

Toy Story (1995)
Jumanji (1995)
Grumpier Old Men (1995)
Waiting to Exhale (1995)
Father of the Bride Part II (1995)
Heat (1995)
Sabrina (1995)

Sundog Education sundog-education.com 82

The other thing we said we wanted to consider was the release years of each movie. Extracting this from the MovieLens data is a little tricky, but the information we want is in the movie titles – they include the year at the end of every title, in parenthesis.

So we just have to do a little bit a string wrangling in our code to extract that data.



time similarity

```
def computeYearSimilarity(self, movie1, movie2, years):
        diff = abs(years[movie1] - years[movie2])
        sim = math.exp(-diff / 10.0)
        return sim
```

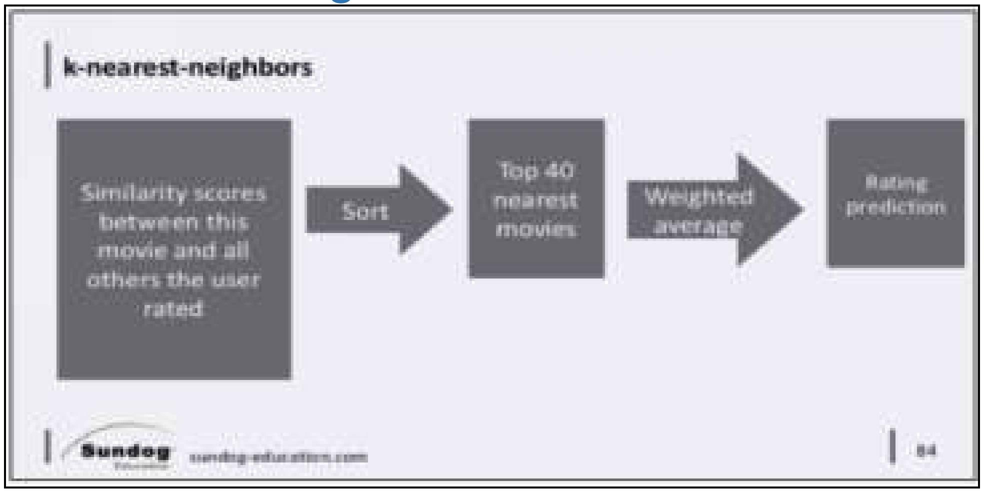
Sundog sundog-education.com 83

How do we assign a similarity score based on release years alone? Well, this is where some of the art of recommender systems comes in. You have to think about the nature of the data you have, and what makes sense. How far apart would two movies have to be for their release date alone to signify they are substantially different? Well, a decade seems like a reasonable starting point. I mean, sci-fi movies in the 70's look pretty different from sci-fi movies in the 80's, for example.

So, we want to start with the difference in release years for two movies – just the absolute value of the difference, it doesn't matter which one came first. Now we need to come up with some sort of mathematical function that smoothly scales that into the range 0 to 1. I chose an exponential decay function – it ends up looking like this. Just look at the right side of the graph, since we're taking the absolute values of the year differences which makes them all positive. At a year difference of 0, we get a similarity score on the Y axis of 1, which is what we want. And this similarity score decays exponentially, getting pretty small at around a difference of 10 years and almost nothing at 20 .

The choice of this function is completely arbitrary, but it seems like a reasonable starting point. In the real world, you'd test many variations of this function to see what really produces the best recommendations with real people.

K-Nearest Neighbors



So how do we turn these similarities between movies based on their attributes into actual rating predictions? Remember, our recommendation algorithms in surpriselib have one job: predict a rating for a given user for a given movie.

One way to do this is through a technique called k nearest neighbors. Quite honestly, it’s a unnecessarily fancy name for a really simple idea. We start by measuring the content-based similarity between everything a given user has rated, and the movie we want to predict a rating for.

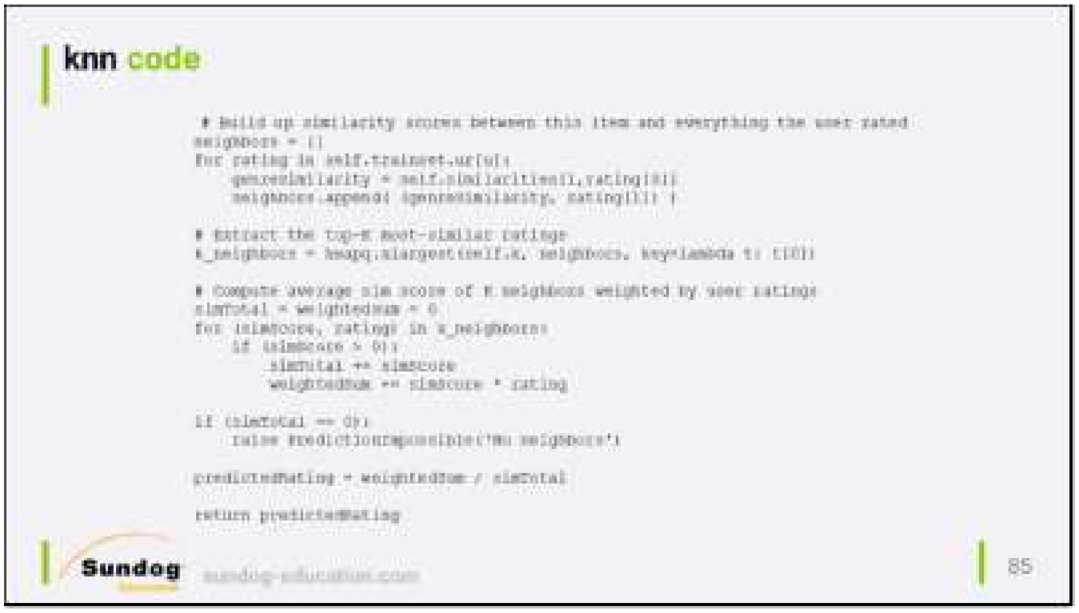
Next, we select some number, call it K, of the nearest neighbors to the movie whose rating we’re trying to predict. You can define “nearest” however you like, so in our case, we’ll say the nearest neighbors are the ones with the highest content-based similarity scores to the movie we’re making a prediction for. So, we could for example select the 40 movies whose genres and release dates most closely match the movie we want to evaluate for this user. That’s it – that’s really all there is to the concept of “k nearest neighbors.” It’s just selecting some number of things that are close to the thing you’re interested in, that is, its “neighbors”, and predicting something about that item based on the properties of its neighbors.

So to turn these top 40 closest movies into an actual rating prediction, we can just take a weighted average of their similarity scores to the movie whose rating we’re trying to predict, weighting them by the rating the user gave them. That’s all there is to it.

let's dive in

Sundog sundog-education.com 86

Coding Activity



knn code

```
 # Build up similarity scores between this item and everything the user rated
neighbors = []
for rating in self.trainset.ur[u]:
    genreSimilarity = self.similarities[i,rating[0]]
    neighbors.append( (genreSimilarity, rating[1]) )

# Extract the top-K most-similar ratings
k_neighbors = heapq.nlargest(self.k, neighbors, key=lambda t: t[0])

# Compute average sim score of K neighbors weighted by user ratings
simTotal = weightedSum = 0
for (simScore, rating) in k_neighbors:
    if (simScore > 0):
        simTotal += simScore
        weightedSum += simScore * rating

if (simTotal == 0):
    raise PredictionImpossible('No neighbors')

predictedRating = weightedSum / simTotal

return predictedRating
```

Sundog sundog-education.com 85

Let's turn that concept into code. This is the meaty part of our prediction function, which takes in a user "u" and an item "I" that we want to predict a rating for. We start with a list called "neighbors", and go through every movie the user has rated, populating it with the content-based similarity score between each movie and the movie we're trying to predict. We pre-computed those similarity scores in the self.similaritie s array.

Next, we use heapq.nlarges t to quickly and easily sort that list into the top-K movies with the highest similarity scores to the movie in question.

After that, it's just a matter of computing the weighted average of the top-K similar movies, weighted by the ratings the user gave them. Assuming we had some data to work with there, we return that as our rating prediction for this user and item.

So let's actually play around and run this thing, and generate some real recommendations just using content-based filtering – we're just going to recommend movies that are similar to movies each user liked, based only on their genre and release date. Let's see how well that actually works.

So, open up Spyder if you haven't already. Let's open up the "ContentBased" folder in the course materials. It contains a copy of the same framework that we developed in the previous section, so there's no need to go through that again. Instead, just open up the ContentRecs.py file, and the ContentKNNAlgorithm.py file.

Let's start by looking at ContentRecs.py. This looks a lot like the "RecsBakeoff" file we wrote while testing out our recommendation evaluation framework; the only real difference is that instead of pitting the SVD algorithm against a random one, we're pitting a new ContentKNNAlgorithm against random recommendations. Already, the work of writing that framework is paying off. In the interests of time, we're not going to compute top-N recommender metrics and only look at accuracy – but we will sample the top-N recommendations for user 85 just to get a feel as to what's going on.

So what's in ContentKNNAlgorithm? Open it up and see.

The main point architecturally is that we're creating ContentKNNAlgorithm as a derived class from surpriselib's AlgoBase class, here on line 15. If you recall, we can implement our own custom recommender algorithms by just implementing our own “estimate” function, and that’s what we’re doing here. But there’s a bit of work we have to do before we can make those predictions.

You can see that we initialize this class by passing in the value of “k” we want to use in k nearest neighbors. By default it’s 40; finding the perfect value of K is just something you have to experiment with in practice .

Next we implement a “fit” function, which gets called by surpriselib when we train an algorithm. What we’re doing here is building up a 2D array that serves as a lookup of the content-based similarity score between any two movies. This takes a while to run, so we print out our progress as we process every 100 movies. The code here is fairly straightforward, you can see we translate user ID’s and item ID’s to their inner ID’s, since that’s what our predict function will be working with, and then we compute genre-based and release-year-based similarity scores for every possible pair and multiply them together to generate a combined content-based similarity score. There’s a little bit of trickery that takes advantage of the symmetry of this 2D array of similarity scores – we know that the score between movies A and B is the same as between movies B and A, so we only compute that score once and copy it into both cases in the array.

Our fit function in turn relies on computeGenreSimilarit y and computeYearSimilarit y , which we’ve implemented here. We already reviewed this code in the slides, but again, the genre similarity is based on a cosine similarity metric between each movie, treating each movies’ genres as coordinates in an 18-dimensional space that represents all possible genres. computeYearSimilarit y is just using an exponential decay function to give more weight to movies that were released at around the same time.

We’ll come back to the mise en scene similarity function later; we’re not using it just yet.

Next we get to the main purpose of our recommender, the estimate() function. This is the same code we looked at earlier, and again, this is where the k nearest neighbors algorithm is happening. We select the k nearest movies a user has rated to the one we're trying to make a prediction for, based on their genres and release years. Then we just compute a weighted average based on their similarity scores and user ratings.

So, let's see if it works! Go back to ContentRecs.py, and hit the play button to run it. Again, you'll want to go get a cup of coffee or something – it will take a few minutes to complete. Hit pause if you're playing along, and resume when it's done.

So, you can see that we definitely outperformed random recommendations – RMSE and MAE are both significantly lower than random recommendations were.

Scrolling down, we can compare the top-10 recommendations for user number 85 using both content based and random recommendations. Qualitatively it's a little hard to say which are better for this particular person, although we do know this user doesn't like movies aimed squarely at children – so "Babe" is probably a questionable result from the random recommender.

One thing that does stick out, however, is that our top predicted ratings for our content-based recommender are significantly lower than those from the random recommender. This is just an artifact of how we chose to compute those predictions; for top-N recommenders, it's the relative order of the predicted ratings that matters, not the predicted ratings themselves – so we shouldn't get too worked up over this. But if you're really striving for rating prediction accuracy, there are ways of normalizing our predicted ratings to get them into the range we want – you can look up log-quantile normalization, for example. But again, in the real world, nobody cares about rating predictions – they care about your top N results.

Still, from a purely qualitative and subjective viewpoint, these recommendations for our test user aren’t that exciting. Personally, I chose user 85 because his or her rating profile reflected some of my own tastes, and these recommendations aren’t ones that I would feel compelled to act on. Can we do better? Let’s see what other sorts of content-based information we might be able to roll into our content-based filtering recommender system.

Bleeding Edge Alert! Mise en Scène Similarities

bleeding edge alert!

Sundog sundog-education.com 88

Time for our first bleeding edge alert! This is where we highlight some new research that looks interesting or promising, but hasn't really made it into the mainstream yet with recommender systems. We want you to have all of the latest and greatest information in this course!

If you're not familiar with the term, we often refer to the current state of the art as "leading edge" – but technology that is still so new that it's unproven in the real world can be risky to work with, and so we call that "bleeding edge."

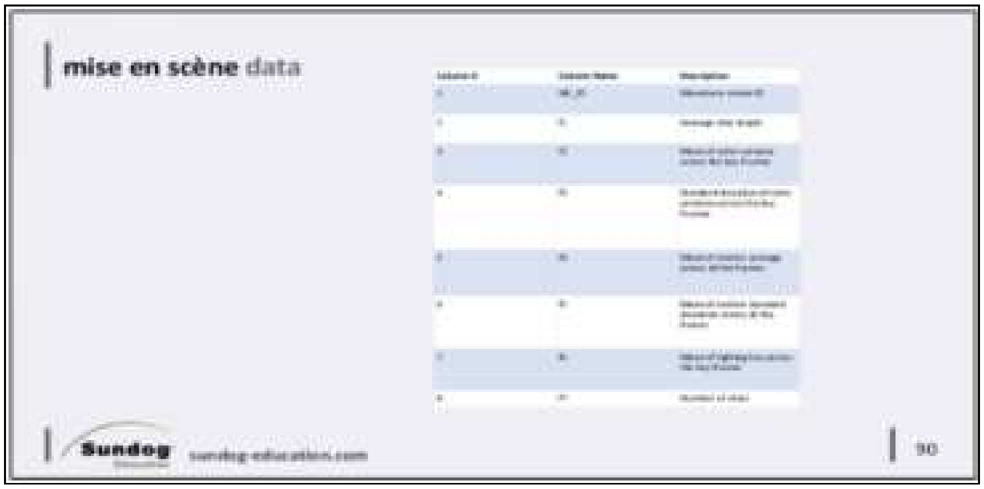
mise en scène

Sundog sundog-education.com 89

Some recent research in content-based filtering has surrounded the use of "mise en scene"data. Technically mise en scene refers to the placement of objects in a scene, but the researchers are using this term a bit more loosely to refer to the properties of the scenes in a movie or movie trailer.

The idea is to extract properties from the film itself that can be quantified and analyzed, and see if we can come up with better movie recommendations by examining the content of the movie itself, scene by scene.

What sort of attributes are we talking about?



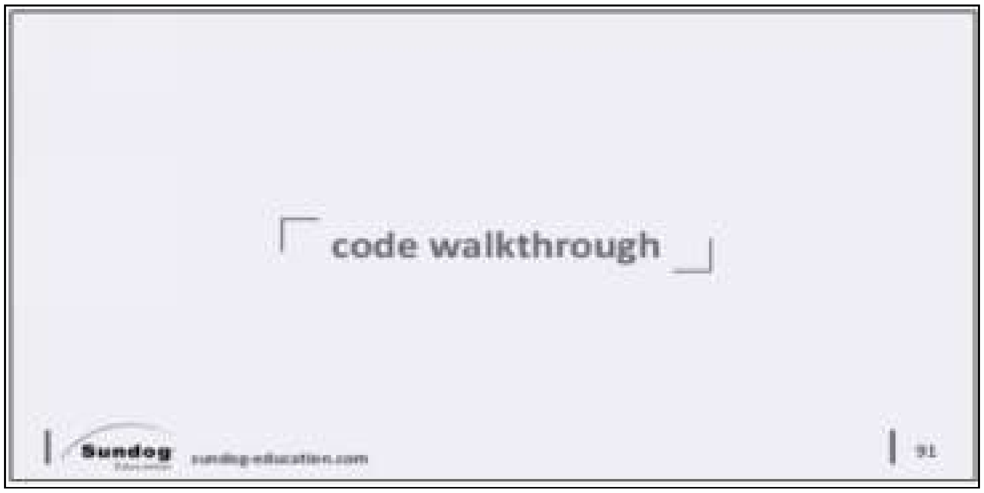
mise en scène data

Column #	Column Name	Description
1	ML_ID	Movielens movie ID
2	[illegible]	Average shot length
3	[illegible]	Mean of color variance across the key frames
4	[illegible]	Standard deviation of color variance across the key frames
5	[illegible]	Mean of motion average across all the frames
6	[illegible]	Mean of motion standard deviation across all the frames
7	[illegible]	Mean of lighting key across the key frames
8	[illegible]	Number of shots

Sundog sundog-education.com 90

Here's the list of attributes that the research team extracted for movie trailers associated with the movielens data set. It includes things like average shot length, color variance, how much motion is in each scene, how scenes are lit, and things like that. So in principle, this should give us a feel as to the pacing and mood of the film, just based on the film itself.

The question is whether this data is more or less useful than the human-generated genre classifications we've already been using for each film. Well, let's find out.



code walkthrough

Sundog sundog-education.com 91

Let’s take another look at ContentKNNAlgorithm.py and have a look at the computeMiseEnSceneSimilarit y function in here. Just like we can compute similarity scores based on genres and release years, we can also compute similarity scores based on this mise en scene data.

How we chose to do that is a bit arbitrary; we’re just taking the absolute difference between these properties for each pair of movies and multiplying them together. There are probably better ways of doing it, but before we spend a bunch of time fine tuning it, let’s just try this out and see if using this data at all does more good than harm.

The code that actually loads and parses the mise en scene data we have is in the MovieLens.py module; it’s not terribly important how that works, but you can go look at it if you’re curious.

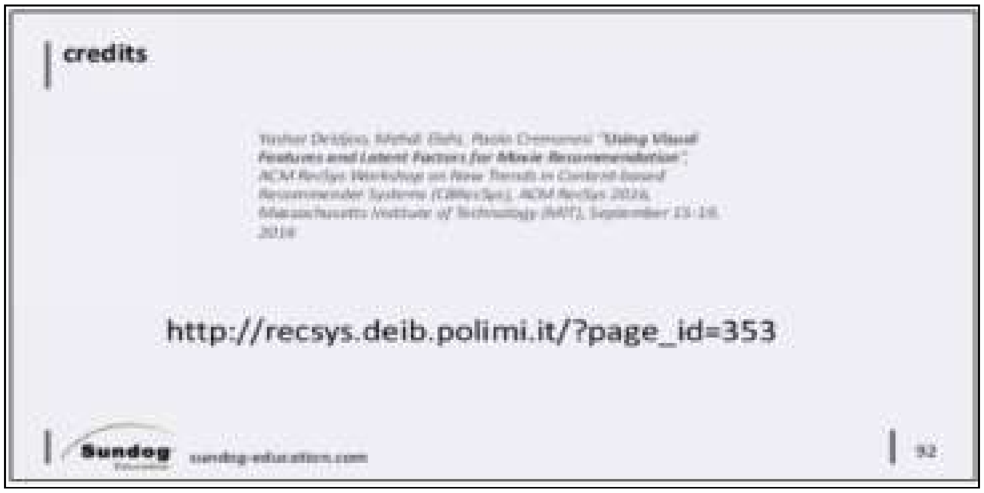
Now, remember our RMSE score from using genres and years was .9375, and our sample top-N recommendations were kind of uninspiring. Let’s put mise en scene data into the mix and see what happens. To do that, just uncomment line 45 to compute that mise en scene similarity score, and multiply it in on line 46, like so.

Let’s run it and see what happens. Save it, select ContentRecs.py, and run it. Again, if you’re playing along, hit pause and resume when it’s done in a few minutes.

OK, so that’s interesting – our RMSE actually got a lot worse, rising to over 1.0. Now, this could again just be an artifact of how we chose to compute mise en scene similarity scores; accuracy isn’t what we’re really concerned with. But do the top-N recommendations look any better from a qualitative standpoint?

I don’t know… it seems like it’s recommending even more obscure stuff, although that new recommendation for “Mission to Mars” seems interesting given this user’s interest in a Star Trek movie. Again, sometimes developing recommendation systems is more an art than a science; you can’t really predict how real people will react to new recommendations they haven’t seen before. Personally I’d be tempted to test this in an A/B test to see how it performs.

If you look at the research literature associated with mise en scene recommendations however, they note that it doesn’t do any favors to accuracy, but does increase diversity. But again, increased diversity isn’t always a good thing when it comes to recommendations – it may just mean you’re recommending random stuff that has no correlation to the user’s actual interests. Still, it was interesting to experiment with it – and it would be even more interesting to experiment with it using real people.


credits

*Yashar Deldjoo, Mehdi Elahi, Paolo Cremonesi: "**Using Visual Features and Latent Factors for Movie Recommendation**", ACM RecSys Workshop on New Trends in Content-based Recommender Systems (CBRecSys), ACM RecSys 2016, Massachusetts Institute of Technology (MIT), September 15-16, 2016*

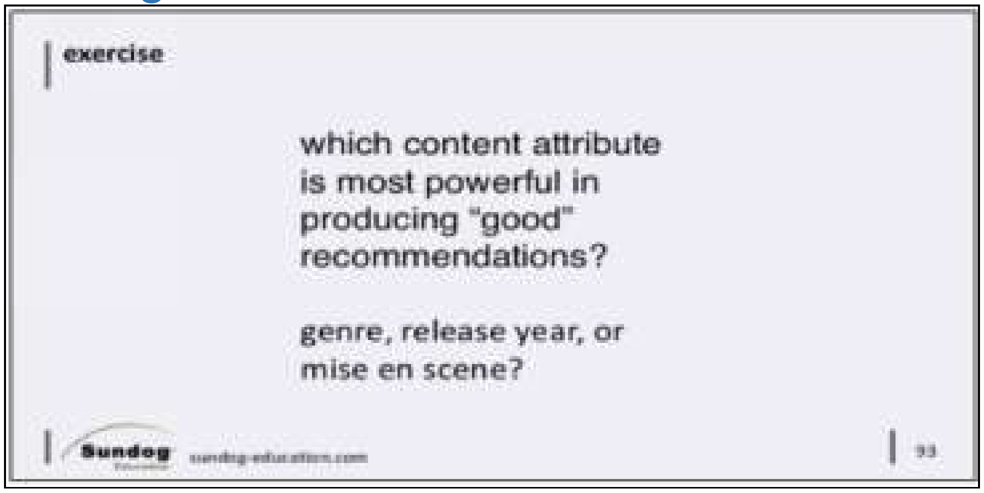
http://recsys.deib.polimi.it/?page_id=353

Sundog sundog-education.com 92

To give the required credit for the mise en scene data we’re using, and to give a pointer for learning more, here’s the reference to the original research paper surrounding it, and a link to where you can explore the data set in more depth.

You can see this is fairly recent research; it’s sort of bleeding edge stuff right now, but I want you to know about the most interesting latest developments in recommender systems as part of this course – this won’t be the last one we explore .

Coding Exercise



exercise

which content attribute is most powerful in producing "good" recommendations?

genre, release year, or mise en scene?

Sundog sundog-education.com 93

As an exercise, try modifying ContentKNNRecommender.py to use genre, release year, and mise en scene data independently. Which produces the best accuracy, and which produces the most satisfying results subjectively to you?

If you have lots of time, you can pass True into the Evaluate function in ContentRecs.py to compute more evaluation metrics to work with.

I’ll go over my results in the next section.

my results

genre	year	mise en scene
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Sundog sundog-education.com

94

So, here are the results I got by trying each content attribute out independently. As far as accuracy goes, genre is the winner, and qualitatively, I think genre is the winner as well.

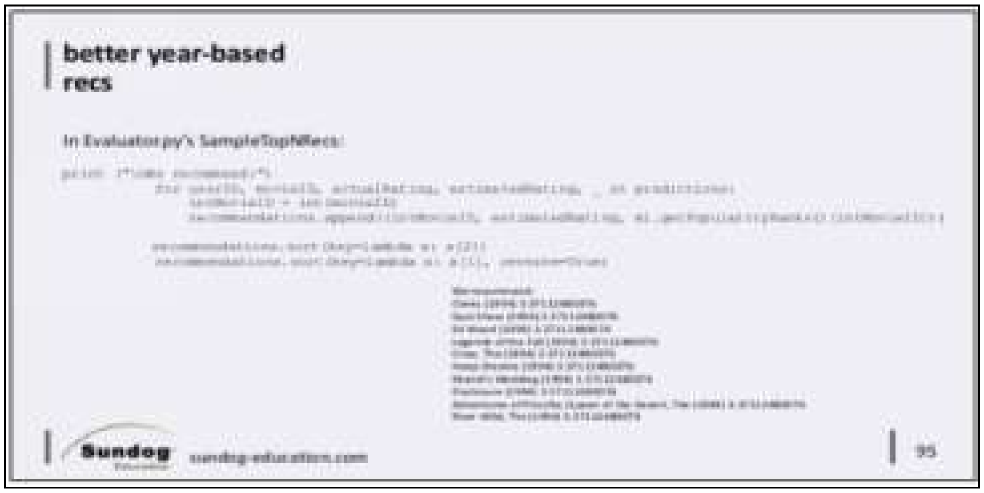
You can see that release year just ended up finding the year this user liked films from the best, and recommended whatever it could find from that year. Since we only have release dates down to the year level, what really happened here is every movie from 1994 was tied for first place, so it’s kind of arbitrary which ones made it into the top 10 recommendations. That doesn’t make for good recommendations.

Mise en scene turned up some really obscure stuff, much of it foreign. It’s possible that this user actually would enjoy foreign films – I only recognize a couple of these, but they are ones I liked personally. Again it’s dangerous to project your own tastes onto somebody else’s recommendation results, but I purposely chose our test user as someone who has similar tastes to my own based on their rating history so I can get a qualitative feel of the quality of the results they are getting .

So, none of these are particularly exciting on their own, although I’d say genre is a clear winner. But combining these attributes together leads to more interesting results, and that’s true in general – the most successful recommender systems are the ones that combine many sources of data, and even many different algorithms, to produce a system that is more than just the sum of its parts.

How would you go about making these results even better based on what you’ve learned so far? Well, we do have popularity ranking data handy – it would be tempting to use popularity at least as a tie-breaker. I bet the release-year-based recommendations would be a lot better if we had a secondary sort based on popularity, for example. If you’re up for more of a challenge, try that out! See if you improve the release year-based recommendations by sorting the “k nearest neighbors” within a given year by popularity. That’s what developing recommender systems is all about; just trying new ideas and seeing what works for your specific users and items that you’re working with. It’s never a one size fits all sort of thing.

Don’t go to the next page, until you’re ready to see a spoiler of how I sorted those year-based recommendations by popularity as a secondary sort.



better year-based recs

In Evaluator.py's SampleTopNRecs:

```
print ("\nWe recommend:")
        for userID, movieID, actualRating, estimatedRating, _ in predictions:
            intMovieID = int(movieID)
            recommendations.append((intMovieID, estimatedRating, ml.getPopularityRanks()[intMovieID]))

        recommendations.sort(key=lambda x: x[2])
        recommendations.sort(key=lambda x: x[1], reverse=True)
```

```
[illegible]
```

sundog-education.com

95

The first thing I did was go back into ContentKNNAlgorithm.py and modify it to only use release years as a criterion, don't forget that or you'll waste a lot of time!

So to have a secondary sort in your top-N results, all we need to do is modify the SampleTopNRecs function in Evaluator.py, which is where we are displaying our top-N recommendations in our test framework.

We already have the MovieLens data available to this function, so we can extract popularity data for each recommendation candidate while we're building up the list of them. You can see that in your recommendations.append call here.

Then to do the secondary sort, we first sort the list by popularity, so the most popular items are at the top of the list. That happens to be in element number 2 of our recommendation tuples. Then we can sort by element 1, which is the estimated rating. So if we have a tie when ranking based on estimated ratings, the original sort by popularity will still be in place from before. There are other ways to do a second sort in Python, but it gets tricky when you need to sort things in different directions, so we'll go with this approach.

The recommendations we end up with are perhaps a little better – honestly, 1994 didn't have many blockbuster movies, and this user already rated most of them. But these results are a little more recognizable than the ones we started with, and some really obscure titles have been dropped from them.

Now, by using popularity data, technically we’re no longer limiting our recommendations to content attributes – popularity is behavior-based data. But the point of this exercise is to not be limited by the buckets different algorithms and approaches present – always be willing to experiment with new ideas, and ways of combining different ideas. That’s where real innovation comes from, and in this case, it made our results a little bit better.

Neighborhood-Based Collaborative Filtering

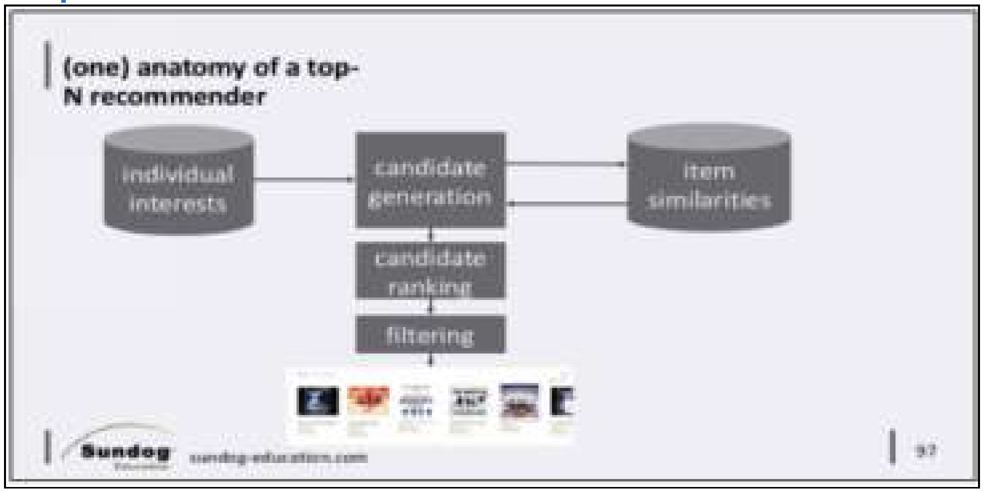
neighborhood-based collaborative filtering

Sundog sundog-education.com

96

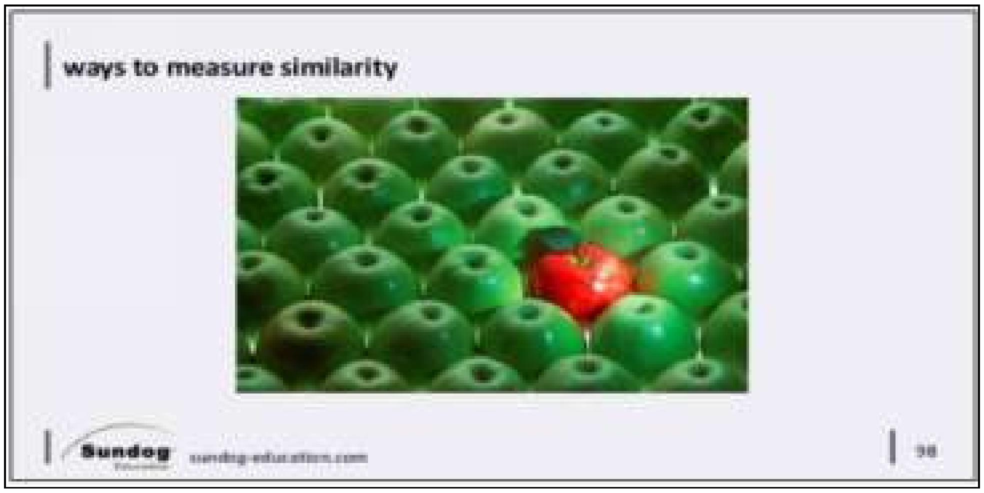
In our next section, we'll cover neighborhood-based collaborative filtering. This is the idea of leveraging the behavior of others to inform what you might enjoy. At a very high level, it means finding other people like you and recommending stuff they liked. Or, it might mean finding other things similar to the things you like; that is, recommending stuff people bought who also bought the stuff you liked. Either way, the idea is taking cues from people like you – your "neighborhood" if you will – and recommending stuff based on things they liked that you haven't tried yet. That's why we call it "collaborative" filtering – it's recommending stuff based on other peoples' collaborative behavior.

Top-N Architectures

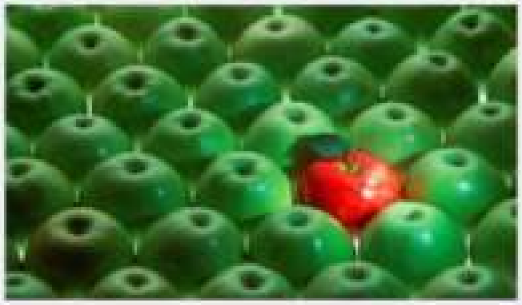


We’re going to start off by revisiting the first architecture for top-N recommender systems we covered earlier in the course. The way we initially did collaborative filtering at Amazon was like this. You start with a data store of some sort that includes ratings information, be they implicit or explicit, from your users. You then generate candidates for things to recommend by looking up other items similar to the items each user liked. Next you score and rank those candidates, filter out stuff the user hasn’t already seen, and produce your final top-N recommendation list.

You can see the heart of this approach is really in that database of item similarities. In other approaches, you might be using a database of similarities between users instead. But either way, producing those lists of similar items or similar people is at the heart of the problem, so let’s spend some time talking about some different ways to measure similarity between people or things.



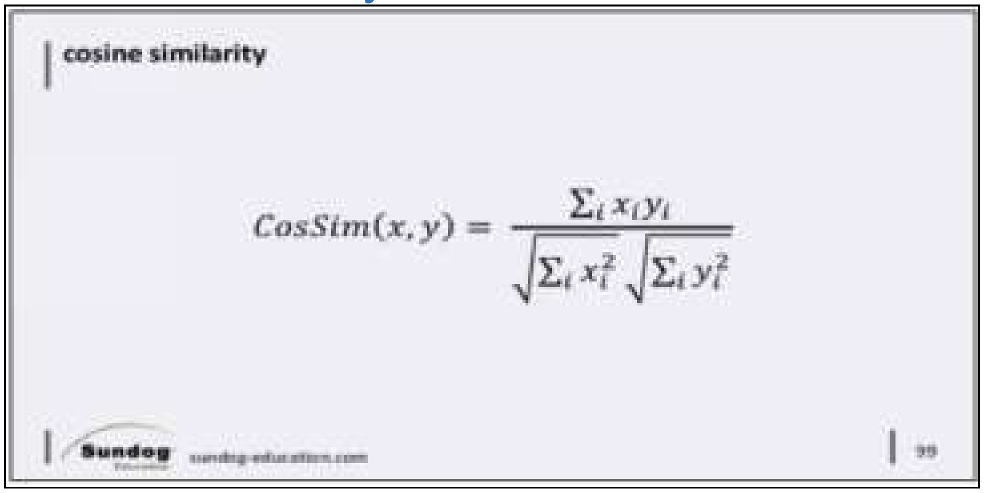
ways to measure similarity



Sundog sundog-education.com 98

The heart of neighborhood-based collaborative filtering is the ability to find people similar to you, or items similar to items you've liked. So just like we measured the similarity between movies with content-based filtering, we can apply the same techniques to measuring similarity based on behavior data. The first step in collaborative filtering is measuring the similarity between things, like these apples, or the similarity between people, so you can find people like you. There are lots of ways to do this.

Cosine Similarity



cosine similarity

$$CosSim(x,y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

Sundog
sundog-education.com
99

The cosine similarity metric works really well in most cases. To recap, if you have a bunch of attributes associated with people or things, you can think of each attribute as a dimension, and think of similarity as the angle between different things when plotted in this multi-dimensional space. Computing the cosine of the angle turns out to be pretty straightforward, even if it's not an easy thing to visualize.

The only difference between this and when we did it with content attributes is that our dimensions will be things like "did this user like this thing", or "was this thing liked by this user." So, every user or every thing might constitute its own dimension. And the dimensions are based on user behavior, instead of content attributes.

The big challenge in measuring these similarities based on behavior data is the sparsity of the data we're working with. There are so many movies in the world, that it's very unlikely that an individual person has seen any specific movie – nobody has seen them all. Well, my old boss at IMDb has, but he's definitely an outlier! For the most part, the data we're working with is very sparse – most movies haven't been watched by a given individual, and conversely, most people haven't seen a given movie generally speaking.

This means it’s tough for collaborative filtering to work well unless you have a LOT of user behavior data to work with. You can’t compute a meaningful cosine similarity between two people when they have nothing in common, or between two items when they have no people in common.

Sparsity

sparsity

	Indiana Jones	Star Wars	Shape of Water	Incredibles	Casablanca
Bob	4				
Ted					
Alice				5	

Sundog sundog-education.com

100

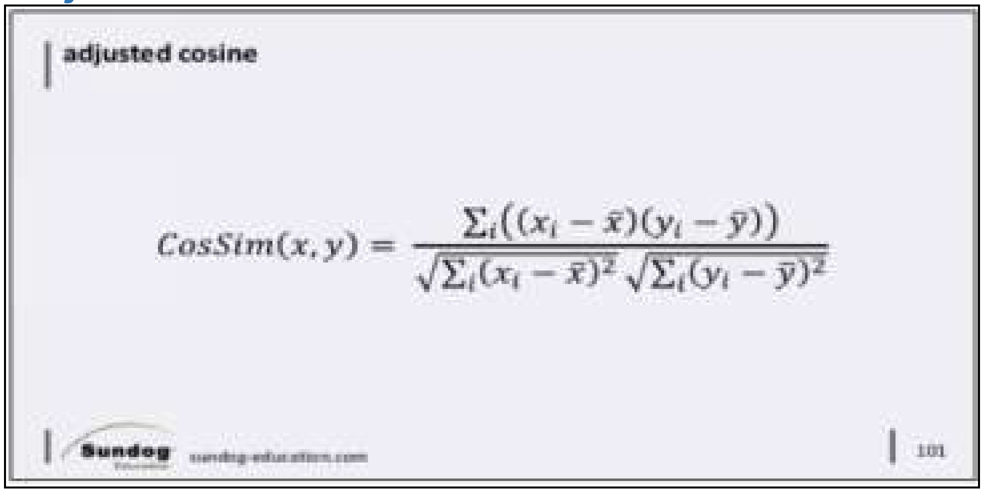
Here's an example of the data a collaborative filtering algorithm might try to be working with. We usually start with a 2D matrix like this of things, people, and the ratings that exist between them. The problem is, that matrix is mostly empty in practice. Of the four movies we listed in this example, Bob has only seen one of them, Alice has only seen one, and Ted hasn't seen any of them. There just isn't enough data to work with, which is why we say the data is sparse. How can I really say anything about how similar Ted is to Alice? All I know is that Alice loved The Incredibles, and Ted hasn't seen it for whatever reason. How can I say anything about the similarity between Star Wars and Casablanca? I literally have no data at all on that. These are the problems sparse data presents.

This is why collaborative filtering works well for big companies like Amazon and Netflix – they have millions of users and so have enough data to generate meaningful relations in spite of the data's sparsity. But even for our examples in this course, the 100,000 ratings we're working with aren't enough to generate really good similarity data.

Sparsity also introduces some computational challenges – you don't want to waste time storing and processing all of that missing data, so under the hood we end up using structures like "sparse arrays" that avoid storing all that empty space in this matrix.

It’s just a reminder that the quantity and quality of the data you have to work with is often way more important than the algorithm you choose. It doesn’t really matter what method you use to measure similarity if you don’t have enough data to begin with. But let’s assume that you do, and talk about some alternative similarity measurements you might use.

Adjusted Cosine



adjusted cosine

$$CosSim(x,y) = \frac{\sum_i((x_i - \bar{x})(y_i - \bar{y}))}{\sqrt{\sum_i(x_i - \bar{x})^2}\sqrt{\sum_i(y_i - \bar{y})^2}}$$

Sundog sundog-education.com 101

So, with our cautionary tale about data sparsity out the way, let’s talk about some other ways to compute similarity .

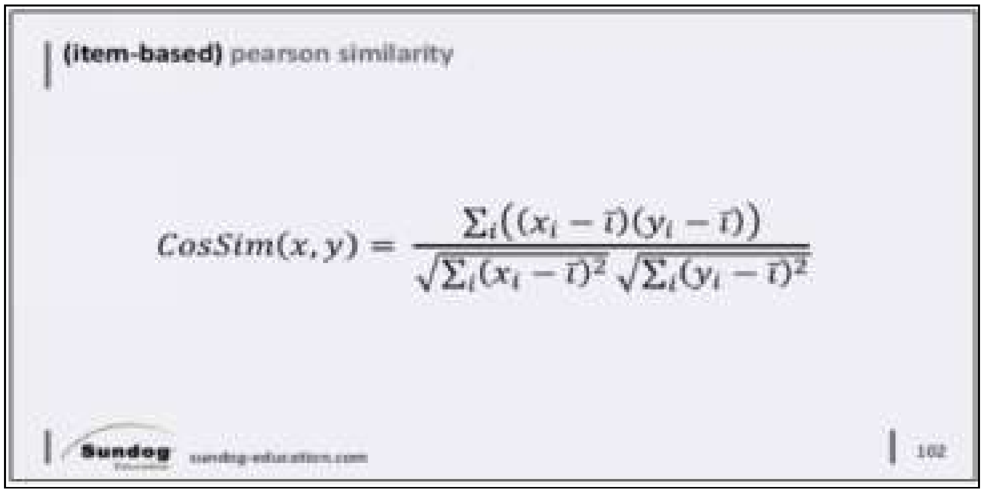
One is the adjusted cosine metric, and it’s applicable mostly to measuring the similarity between users based on their ratings. It’s based on the idea that different people might have different baselines that they are working from – what Bob considers a 3-star movie may be different from what Alice considers a 3-star movie. Maybe Bob is just hesitant to rate things 5 stars unless they are truly amazing, while Alice tries to be nice and rates things 5 stars unless she really didn’t like them. This is a real effect you’ll see not just across different individuals, but across different cultures too. Some countries are more brutal with their ratings than others.

So, adjusted cosine attempts to normalize these differences. Instead of measuring similarities between people based on their raw rating values, we instead measure similarity based on the difference between a users' rating for an item, and their average rating for all items. So if you look at this equation, we've replaced X with Xsub-I minus x-bar, and replaced Y with y-sub-I and y-bar. X-bar means the average of all of users X's ratings, and Y-bar means the average of all of user Y's ratings. So all that's different here from conventional cosine similarity is that we're looking at the variance from the mean of each users' ratings, and not just the raw rating itself.

Now, this sounds good on paper, but in practice data sparsity can really mess you up here. You can only get a meaningful average, or baseline, of an individuals' ratings if they have rated a lot of stuff for you to take the average of in the first place. The MovieLens data set we're working with cheats a little, because they've only included data from people who rated at least 20 movies. In the real world, you might have a lot of users who are new to your system who have only rated one thing – and that data will be totally wasted with the adjusted cosine metric. No matter what they rated that one movie that they told you about, the difference between it and that users' mean will be zero at that point .

So adjusted cosine might be worth experimenting with, but only if you know that most of your users have rated a lot of stuff, implicitly or explicitly. And if you have that much data to begin with, these differences between individuals will start to work themselves out anyway – so you're not likely to see as much of a difference as you might expect when using adjusted cosine.

Pearson Similarity



(item-based) pearson similarity

$$CosSim(x,y) = \frac{\sum_i((x_i - \bar{i})(y_i - \bar{i}))}{\sqrt{\sum_i(x_i - \bar{i})^2}\sqrt{\sum_i(y_i - \bar{i})^2}}$$

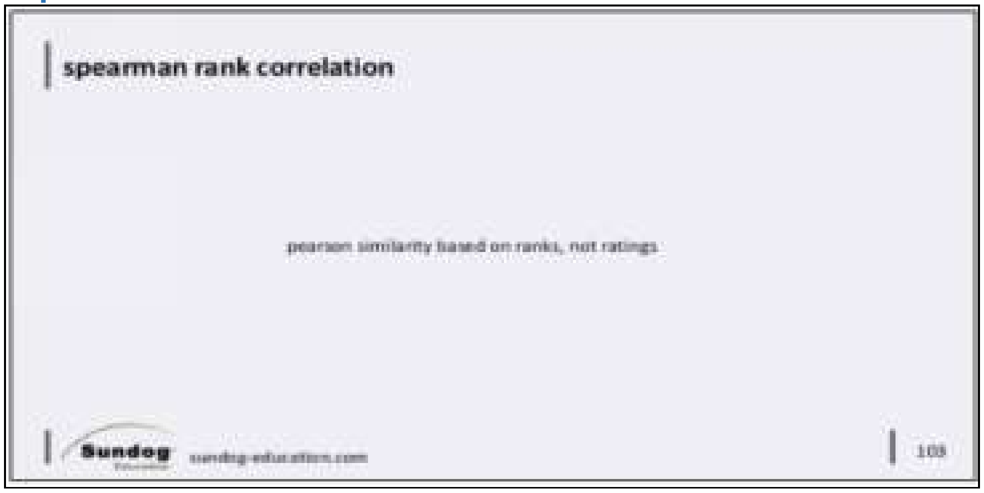
Sundog sundog-education.com 102

A slight twist on adjusted cosine is the Pearson similarity metric. The only difference is that instead of looking at the difference between ratings and a users' average rating, we look at the difference between ratings the average from all users for that given item. So, in this equation we've substituted x-bar and y-bar for i-bar, the average rating of the item in question from across all users.

So, we're no longer trying to account for an individual's personal definitions of specific rating scores, and in a real-world situation with sparse data, that's probably a good thing. But you can think of Pearson similarity as measuring the similarity between people by how much they diverge from the average person's behavior. Imagine a film that most people love, like Star Wars. People who hate Star Wars are going to get a very strong similarity score from Pearson similarity, because they share opinions that are not mainstream.

Note that the only difference between this and adjusted cosine is whether we're talking about users or items. The Surprise library we're using in this course refers to adjusted cosine as "user-based Pearson similarity" because it's basically the same thing, which can be a little confusing. The Pearson similarity we're talking about in this slide is what they call "item-based Pearson similarity." Surprise's documentation also uses slightly different notation than we do, but if you break it down it's the same idea. Making sense of notation is often half the battle of understanding these algorithms!

Spearman Rank Correlation



spearman rank correlation

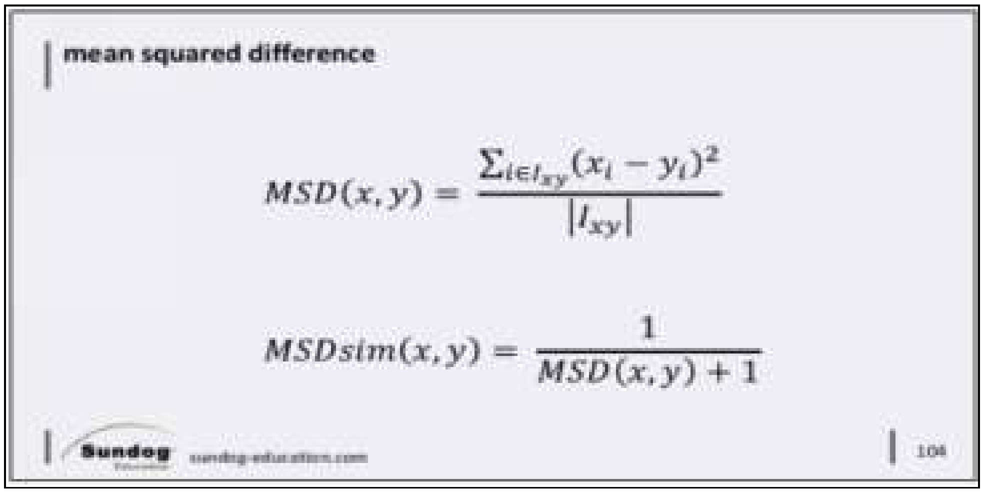
pearson similarity based on ranks, not ratings

Sundog sundog-education.com 103

For completeness, I'm going to briefly mention Spearman rank correlation as well. At a conceptual level, it's the same idea as Pearson similarity, but instead of using rating scores directly we use ranks instead. That is, instead of using an average rating value for a movie, we'd use its rank amongst all movies based on their average ratings. And instead of individual ratings for a movie, we'd rank that movie amongst all that individuals' ratings.

I'm not going into the details on this, because it's going to confuse you. The math gets tricky, it's very computationally intensive, and it's generally not worth it. The main advantage of Spearman is that it can deal with ordinal data effectively, for example if you had a rating scale where the difference in meaning between different rating values were not the same. I've never seen this actually used in real world applications myself, but you may encounter it in the academic literature .

Mean Squared Difference



mean squared difference

$$MSD(x,y) = \frac{\sum_{i \in I_{xy}} (x_i - y_i)^2}{|I_{xy}|}$$

$$MSDsim(x,y) = \frac{1}{MSD(x,y) + 1}$$

Sundog sundog-education.com 104

Another way to measure similarity is the mean squared difference similarity metric, and it's exactly what it sounds like. You look at, say, all of the items that two users have in common in their ratings, and compute the mean of the squared differences between how each user rated each item. It's easier to wrap your head around this, since it doesn't involve angles in multi-dimensional space – you're just directly comparing how two people rated the same set of things. It's very much the same idea of how we measure mean absolute error when measuring the accuracy of a recommender system as a whole.

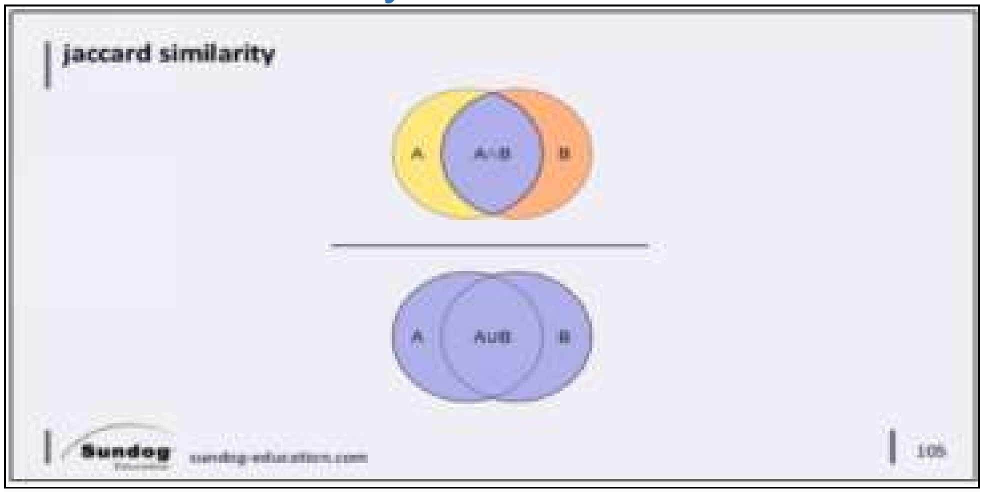
So if we break down that top equation, it says the mean squared difference, or MSD for short, between two users X and Y, is given by the following. On the top of this fraction, we are summing up for every item I that users X and Y have both rated, the difference between the ratings from each user, squared. We then divide by the number of items these users had in common that we summed across, to get the average, or mean .

Now the problem is that we've computed a metric of how different users x and y are, and we want a measure of how similar they are, not how different they are. So to do that, we just take the inverse of MSD, dividing it by one – and we have to stick that "plus one" on the bottom in order to avoid dividing by zero in the case where these two users have identical rating behavior.

You can, by the way, flip everything we just said to apply to items instead of users. So, x and y could refer to two different things instead of two different people, and then we'd be looking at the differences in ratings from the people these items have in common, instead of the items people have in common. As we'll see shortly, there are two different ways of doing collaborative filtering – item-based, and user-based, and it's important to remember that most of these similarity metrics can apply to either approach.

So that's MSD. It's easier to comprehend than cosine similarity metrics, but in practice you'll usually find that cosine works better.

Jaccard Similarity


jaccard similarity

Sundog sundog-education.com 105

The last similarity measure we'll cover is the Jaccard similarity metric. Fortunately it's real easy to wrap your head around. Let's imagine two users A and B. The Jaccard similarity between these two users is simply the size of the intersection between these two users' ratings, divided by the union of those two users' ratings. So we would count up all of the movies for example that both users indicated some sort of interest in, and divide that by the total number of movies that either user indicated interest in.

Since we're just counting things up, we're not looking at actual rating values at all here – which is some pretty important information to be throwing away, you would think. But if you're dealing with implicit ratings, for example, just the fact that somebody watched something or bought something, you might not have real rating values to work with anyhow – you just have the fact that they either did something to express interest in an item, or they didn't. And in that case, Jaccard can be a reasonable choice that's very fast to compute.

recap

cosine

adjusted cosine

pearson

spearman

msd

jaccard

Sundog sundog-education.com

106

So a quick recap on different similarity measures before we move on.

Cosine similarity is a good jack-of-all-trades. It’s almost always a reasonable thing to start with.

Adjusted cosine and Pearson are two different terms for basically the same thing – it’s mean-centered cosine similarities at the end of the day. So it works in average rating behavior across all of a user’s item ratings, or the average ratings of an item across all users, depending on which way you flip it. The idea is to deal with unusual rating behavior that deviates from the mean, but in practice it can sometimes to more harm than good.

Spearman rank correlation is the same idea as Pearson, using rankings instead of raw ratings, and it’s not something you’re likely to be using in practice.

MSD is mean squared difference, which is just an easier similarity metric to wrap your head around than cosine similarity, but in practice it usually doesn’t perform better. Finally we talked about Jaccard similarity, which is just looking at how many items two users have in common, or how many users two items have in common, divided by how many items or users they have between both of them. It’s really simple and well suited to implicit ratings, like binary actions such as purchasing or viewing something, where you either do something or you don’t. But you can also apply cosine similarity to implicit ratings, too – so at the end of the day, cosine similarity remains my default “go-to” similarity metric.

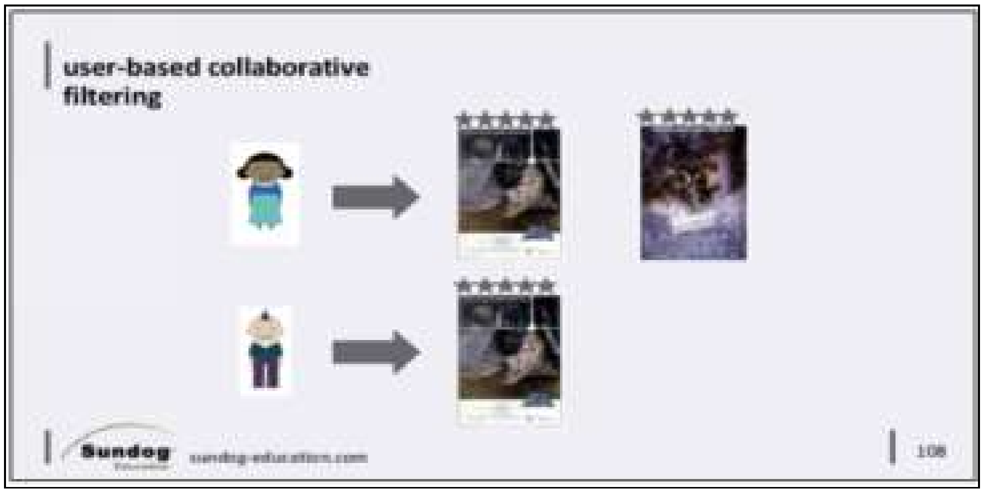
User-based Collaborative Filtering

user-based collaborative filtering

Sundog sundog-education.com

107

Let's talk about one specific implementation of neighborhood-based collaborative filtering - user-based collaborative filtering. It's the easiest one to wrap your head around, so it seems like a good place to start.



user-based collaborative filtering

Sundog sundog-education.com

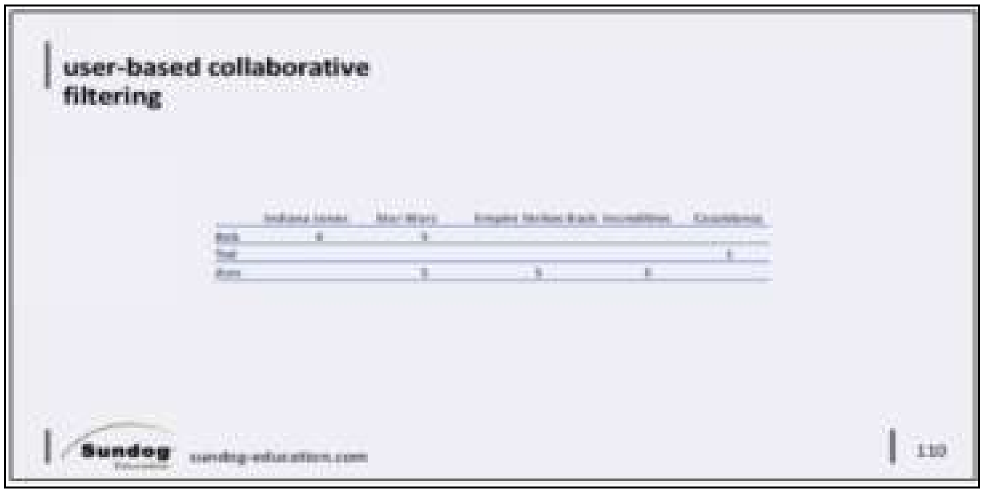
108

The idea behind user-based collaborative filtering is pretty simple. Start by finding other users similar to yourself based on their ratings history, and then recommend stuff they liked that you haven't seen yet. So let's say that Ann and Bob here both loved the first Star Wars movie. Ann also watched the next Star Wars movie, The Empire Strikes Back – but Bob lives under a rock, and hasn't seen that yet.

user-based collaborative filtering

Sundog Education sundog-education.com 109

So, it seems kind of obvious to recommend The Empire Strikes back to Bob. If Ann loved the same movies Bob loves, and Ann loves The Empire Strikes Back, there's a good chance Bob will love it too.



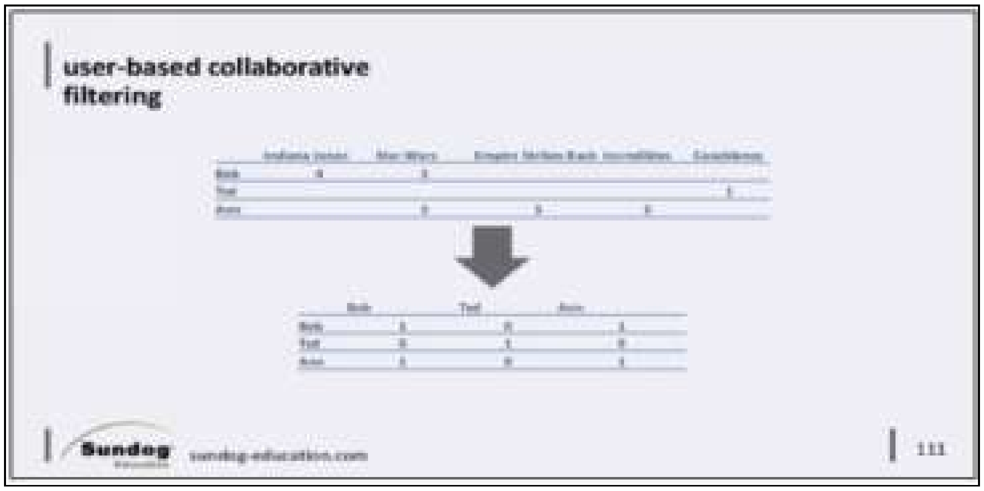
user-based collaborative filtering

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5			
Ted					1
Ann		5	5	4	

Sundog Education sundog-education.com 110

So how do we do this? Well the first step is collect the data we need – that is a table of all of the ratings for everyone in our system. We can think of this as a 2D array with movies on one axis, users on the other, and ratings values in each cell. So we can see here that Ann Loved both of the first Star Wars films, Bob only saw the first one and loved it, and they each saw other stuff as well that they don't have in common. Ann might enjoy Indiana Jones, from the looks of it, since she has that connection to Bob through Star Wars.

So now, it's not too much of a stretch to imagine this data as describing vectors for each user in item space. If these were the only five movies in existence, we could describe Bob with a 5-dimensional vector of (4, 5, 0, 0, 0) and Ann with (0, 5, 5, 5, 0), where 0 indicates a missing value. So that gives us what we need in order to compute the cosine similarity score between any two users. Of course, we can experiment with different similarity metrics too, but let's start with cosine.



user-based collaborative filtering

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5			
Ted					1
Ann		5	5	5	

	Bob	Ted	Ann
Bob	1	0	1
Ted	0	1	0
Ann	1	0	1

Sundog sundog-education.com

111

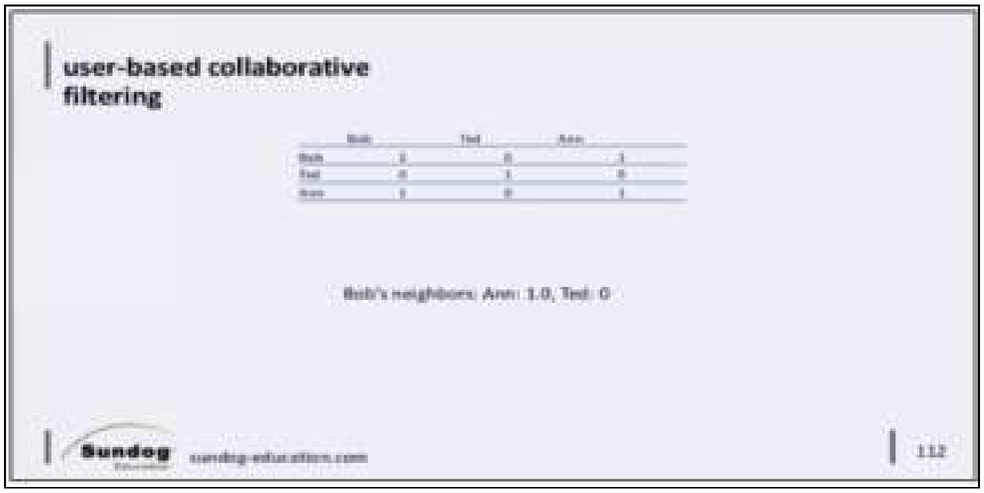
So now we can build up another 2D matrix – one that maps the cosine similarity score between any two users in our system. If our only three users are Bob, Ted, and Ann, it might look like this. Take a close look so you can understand what's going on here.

This will all just fall out of the algorithm for measuring cosine similarity, but you can see that it makes intuitive sense looking at the results. Everyone is 100% similar to themselves, so if we look up Bob's similarity to Bob for example, we get 1.0, and that's true for anyone who has rated at least one thing.

Bob and Ted have no movies in common at all that they both rated, so they end up with a similarity score of 0. Note there is some symmetry here – the combination of Bob and Ted can be looked up separately from Ted and Bob, but it's the same score either way. We can exploit that symmetry to avoid computing half of the values in this table.

Bob and Ann are the more interesting example – while they have different sets of movies that they rated, when we measure similarity we only look at the movies they have in common. In this case, they only have one movie in common – Star Wars – and since they both gave it the same rating of 5 stars, they too get a similarity score of 1.0, or 100%.

I'd like to point out that saying two users are 100% similar doesn't necessarily mean they love the same things – it can also mean they hate the same things. If Bob and Ann both rated Star Wars 1 star, they would still be 100% similar. In fact, the math behind cosine similarity works out such that if you only have one movie in common, you end up with 100% similarity no matter what – even if Bob loved Star Wars and Ann hated it, in a sparse data situation they'd both end up 100% similar. I'm just reinforcing that sparse data is a huge problem with collaborative filtering in general, and it can lead to weird results, and sometimes you need to enforce a minimum threshold on how many movies users have in common before you consider them to avoid weird stuff like that.


user-based collaborative filtering

	Bob	Ted	Ann
Bob	1	0	1
Ted	0	1	0
Ann	1	0	1

Bob's neighbors: Ann: 1.0, Ted: 0

Sundog sundog-education.com 112

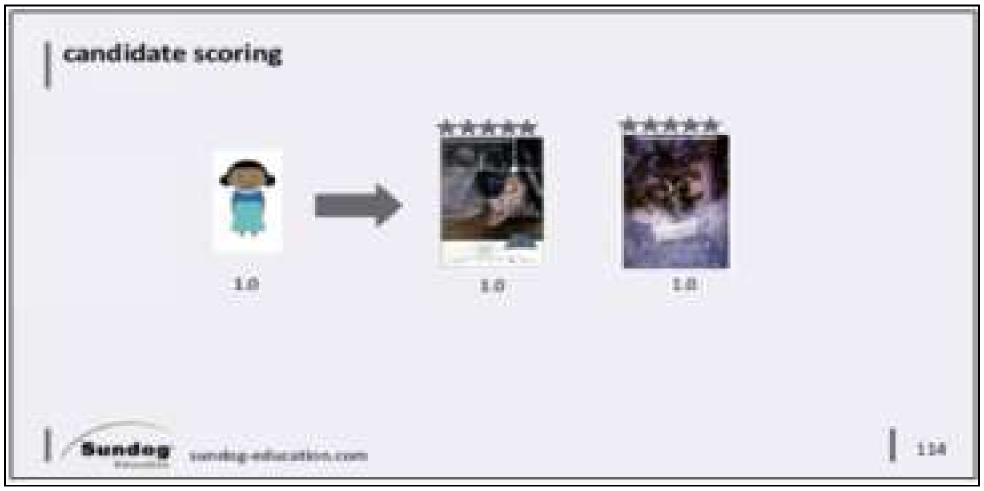
So let's say we want to generate recommendations for Bob now. We can use this handy matrix that we pre-computed to quickly look up how similar he is to everybody else.

Then we can sort that list by similarity scores, and pick off the top N neighbors from that list. In this case we end up with Ann at the top of the list, and we'd probably discard Ted by enforcing a minimum similarity score threshold at this stage. We also skip over the case of comparing Bob with himself. In a less sparse data situation, you'd have more good results to work with besides just Ann of course.

candidate generation

Sundog sundog-education.com 113

So now we can pull all of the movies Ann, and everyone else from the Bob's top-N user similarity results, and consider these recommendation candidates. Still with me? In this simple example, Ann is the only user who is similar to Bob, but if there were other users that we found, we'd throw their ratings in this pile as well.



candidate scoring

Next we need to figure out which of these recommendation candidates are the best ones to present to Bob, so we need to score them somehow.

There are different ways to do this, but it seems reasonable to take the rating assigned to each candidate as part of it. We want to recommend things that similar users loved, not things that similar users hated. So one component could be some sort of normalized rating score here – maybe we translate a 5-star rating to 1.0 just to keep everything on the same scale.

We should probably also take the similarity with the user who generated these ratings into account, so maybe we can multiply that rating score of 1.0 by Ann's similarity score to Bob, which is also 1.0. So we end up assigning a candidate score of 1.0 to both of these items.

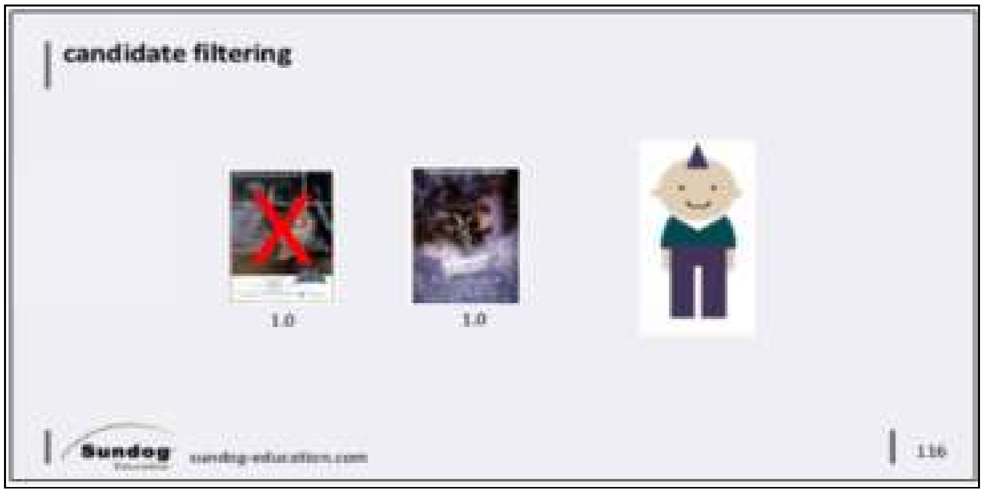
You might also encounter the same movie more than once, if more than one similar user rated it. In this case you probably want to strengthen the relationship to that movie by adding it in again to the final score for that movie.

Again, there are variants on how to do this. Maybe instead of normalizing ratings scores to a 0-1 scale, you can actually assign negative scores to things rated 1 or 2 stars to actively push them down in the final results. There are a lot of different ways to score recommendation candidates, and there’s no standard way of doing it. It’s another case of where you need to experiment to see what works best for the data you have.

candidate sorting

Sundog sundog-education.com 115

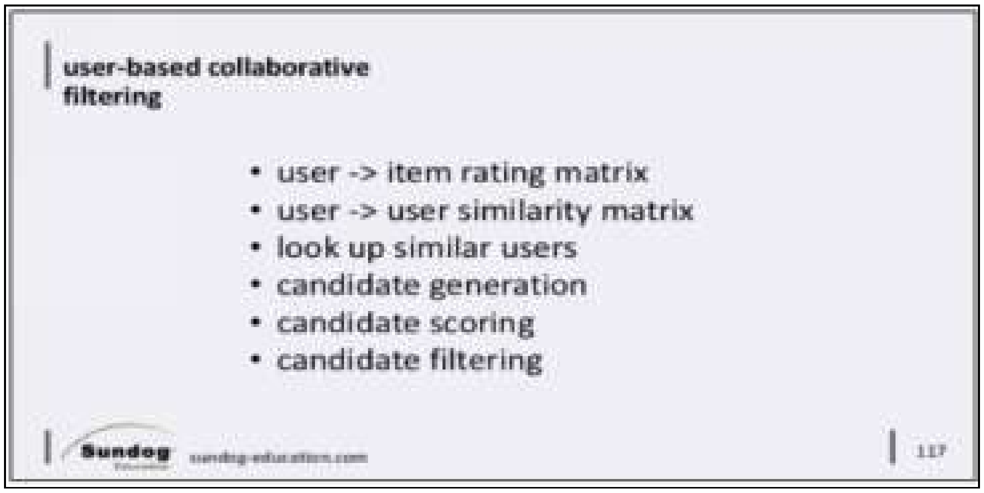
Next, we just sort the recommendation candidates by their final scores. In our simple example, we have a two-way tie of 1.0.


candidate filtering

Sundog sundog-education.com 116

The last step is to filter out stuff Bob already rated, since there's no point in recommending movies he's already seen. You might filter things out in other ways, too – maybe there is some minimum candidate score below which you won't present a result to a user no matter what, or maybe you want to filter out results that might be offensive based on their content.

That leaves us with The Empire Strikes Back, which hopefully makes Bob a happy user of our user-based collaborative filtering system!



user-based collaborative filtering

- user -> item rating matrix
- user -> user similarity matrix
- look up similar users
- candidate generation
- candidate scoring
- candidate filtering

Sundog sundog-education.com 117

So to recap, the steps involve in user-based collaborative filtering are these:

Start by building up a lookup table of users to all of the items they rated, and those rating values.

Then build up another 2D matrix of similarity scores between every pair of users. At this point you've done all of the computationally intensive stuff, and can just re-use these tables to generate recommendations quickly for anybody.

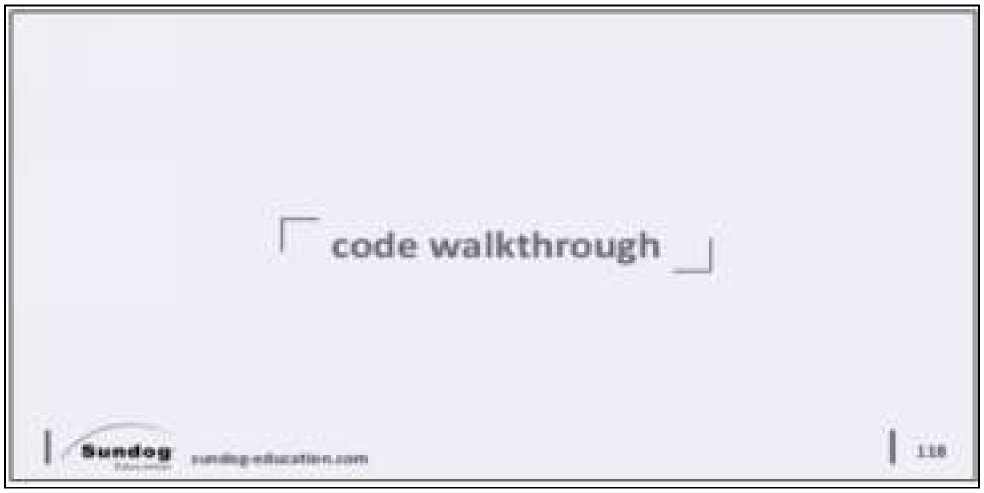
When we want recommendations for a specific user, we can look up all the top similar users to that user .

Next, we generate recommendation candidates by pooling together all of the stuff those similar users rated.

We then score all of those candidates by looking at how the similar users rated them, how similar the user rating them was to you, and whatever else you want.

Finally we filter out stuff the user has already seen, and we're done!

Coding Activity


code walkthrough

Sundog sundog-education.com 118

So let’s make all this talk concrete, and run some real code to perform user-based collaborative filtering on the MovieLens data set. Open up Spyder, and close out any old stuff we don’t need anymore. Now open up the CollaborativeFiltering folder in the course materials, and load up everything in there.

The file we’re interested in right now is SimpleUserCF.py, so click on that tab and let’s walk through it .

It’s surprisingly small, right? Oh, see what I did there? Surprisingly? It’s because we’re using the Surprise library to do a lot of the heavy lifting here.

So this script just walks through the steps we talked about in the slides. It’s not really integrated into the evaluation framework we developed earlier, because user-based collaborative filtering as we described is strictly for generating top-N recommendations; at no point did we attempt to predict user ratings, so we can’t really shoehorn this algorithm into the Surprise framework because it revolves around making rating predictions. That’s OK, we’ll get back to our framework really soon, and we’re going to still use some components of Surprise.

Step one was to load up our data, and that’s the first thing this script does – loads up the MovieLens ratings data set, and builds up a complete training set. We don’t need to split out a test set since again, we’re not measuring accuracy here, we’re just making top-N recommendations and seeing what they look like.

This next chunk of code builds up the similarity matrix between every possible user pair. To do this quickly and easily, we’re using SurpriseLib’s KNNBasic algorithm, which as part of what it does builds up a similarity matrix that we can use instead for our own purposes. Pay particular attention to the sim_option s parameters here – we’re specifying that we want to use cosine for our similarity metric. You could also specify msd or pearson here, and surpriselib would just do that for you. We’re also specifying that we want user based similarities, so we get a matrix mapping user to user similarity scores.

Then we just tell SurpriseLib to go and build that user similarity matrix, and hang on to a copy of it for ourselves .

Next on lines 33-34, we extract all of the similar users to our test user, user number 85 who we will grow to know and love. We then convert that to a list of tuples containing the inner user ID’s and user similarity scores to user 85, being careful to skip over the similarity to ourself as that’s not useful data. We then use the heapq.nlarges t function to quickly sort all of the users by their similarity to user 85, and pluck off the top k results to get our “neighborhood” of similar users.

Next, we build up our list of recommendation candidates by extracting everything these similar users rated, and as we go we score them based on their ratings by the similar users, and by their similarity to our user. One thing that’s worth calling out is that we’re actually adding in the score for a given movie if we encounter it more than once, giving more weight to items that more than one similar user rated. Our use of Python’s defaultdic t is how we ensure we get a default value of 0 for each item to start with.

Next, we build up a dictionary for use in the final filtering stage, so we can quickly look up whether a recommendation has already been rated by the user.

The last step is to sort the recommendation candidates by their final scores, and go through the top N of those, skipping over results that we've already watched. As we go, we look up the movie names and print them out in a human readable format.

So, let's run it and see what happens!

Whoah, that was really quick, right? It's because we didn't have to predict the rating for every possible item by every possible user as part of trying to measure accuracy. The only time consuming part here is building up that user-to-user similarity matrix, and after that making recommendations for individuals is really, really fast. That's why collaborative filtering is something big companies like Amazon can use easily with their massive data set and massive transaction rates – efficiency matters as much as anything in those sorts of large-scale settings.

And the results look really exciting, too! I chose user 85 because his or her rating profile seemed similar to my own interests, and every single one of the movies recommended here are ones that I have both seen and enjoyed. Even though 100,000 ratings is considered relatively sparse, we still ended up with some very promising results – much better than what we saw when using content attributes alone.

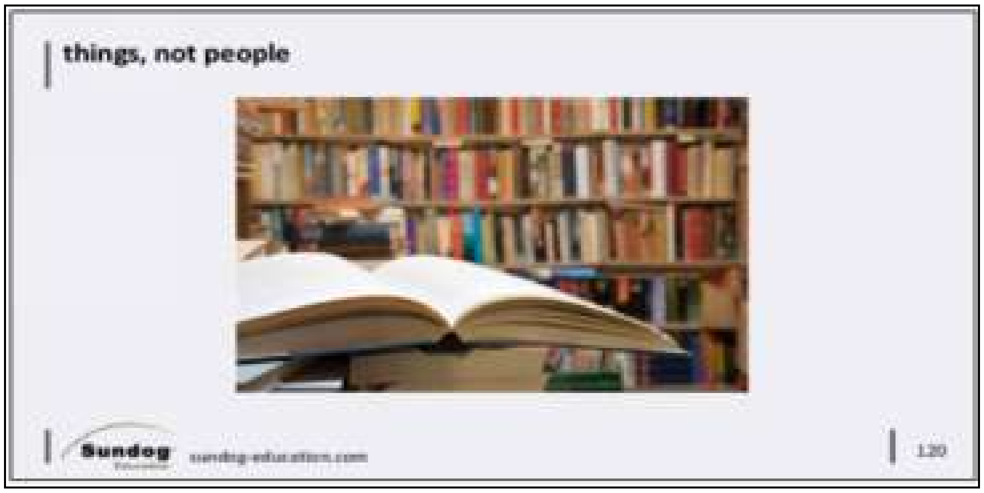
So finally, we're getting some results that we can really be happy about here! You could probably stop taking this course right now and have what you need to make a successful real world system. But, we want to squeeze every dollar we can out of our recommender system, and there are a few more tricks to show you on how to make the results even better – so stick with me here.

Item-Based Collaborative Filtering

item-based collaborative filtering

Sundog
sundog-education.com
119

Another way to do collaborative filtering is by flipping the problem on its head – instead of looking for other people similar to you and recommending stuff they liked, look at the things you liked, and recommend stuff that's similar to those things. We call this item-based collaborative filtering, instead of user-based.



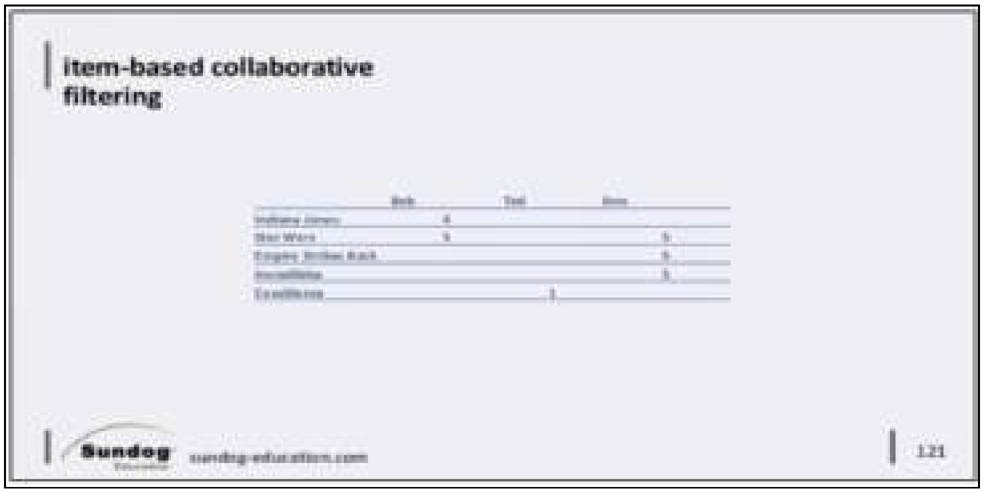
things, not people

There are a few reason why using similarities between items could be better than similarities between people.

One is that items tend to be of a more permanent nature than people. A math book will always be a math book, but an individual's tastes may change very quickly over the span of their lives. So, focusing on the similarities between unchanging objects can produce better results than looking at similarities between people who may have liked something last week, and something totally different this week. Your math book will always be similar to other math books, but a person who liked a math book might be bored with math a few months from now. As such, you can get away with computing an item similarity matrix less often than user similarities, because it won’t change very quickly.

Another very important advantage to building item similarities is that you usually have far fewer items to deal with than people. Whatever company you’re developing a recommender system for probably has a relatively small product catalog compared to the number of customers they have – there are way more people than there are things to recommend to them in most cases. This means that a 2D matrix mapping similarity scores between every item in your catalog will be much smaller than a 2D matrix mapping similarities between every user that visits your site. Not only does that make it simpler to store that matrix, it makes it faster to compute as well. And when you’re dealing with massive systems like Amazon or Netflix, computational efficiency is very important. Not only does it require fewer resources, it means you can regenerate your similarities between items more often, making your system more responsive when new items are introduced.

Using item similarities also makes for a better experience for new users. When a new user comes to your web site, as soon as they have indicated interest in one thing, you can recommend items similar to that thing to them. With user-based collaborative filtering, you wouldn’t have any recommendations for a new user until they make it into the next build of your user similarity matrix.



item-based collaborative filtering

	Bob	Ted	Ann
Indiana Jones	4		
Star Wars	5		5
Empire Strikes Back			5
Incredibles			5
Casablanca		1	

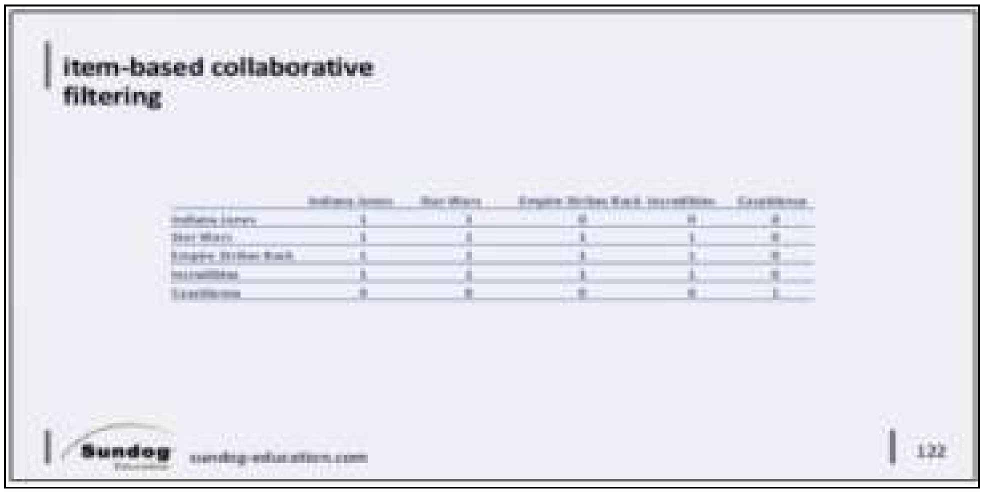
Sundog sundog-education.com 121

So the general process for item-based collaborative filtering is really the same as user-based; the only difference is that we flip the use of users and items.

So instead of starting with a matrix that has users as rows and items as columns, we're starting with items as the rows and users as the columns. This lets us look up all the user ratings for a given item quickly, so we can measure similarity between items based on the users that rated them.

Before, we thought of users existing in a space where every item was a dimension, and found the cosine similarity between these user vectors. But now, we're just flipping that – we're thinking of items existing in a space where every user is a dimension, and finding the cosine similarity between these item vectors .

Of course, cosine similarity is just one of many ways to measure the similarity between items – and there may be other ways of scoring these item pairs that aren't based on similarity at all. But I can't get into that without violating NDA's!



item-based collaborative filtering

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Indiana Jones	1	1	0	0	0
Star Wars	1	1	1	1	0
Empire Strikes Back	1	1	1	1	0
Incredibles	1	1	1	1	0
Casablanca	0	0	0	0	1

Sundog Education sundog-education.com 122

So now we can compute the cosine similarity scores between every possible item pair, based on the users each item has in common who rated them.

Again in this example all of our scores come out to 1 or 0 because we have so little data to work with, but in the real world you'd see more interesting and meaningful numbers here. You might also notice that we ended up with a larger matrix than when we looked at user similarities, but again that's just a quirk of our example data – normally you would have more users than items, not the other way around, and that would mean your item similarity matrix will be smaller than a user similarity matrix would be.

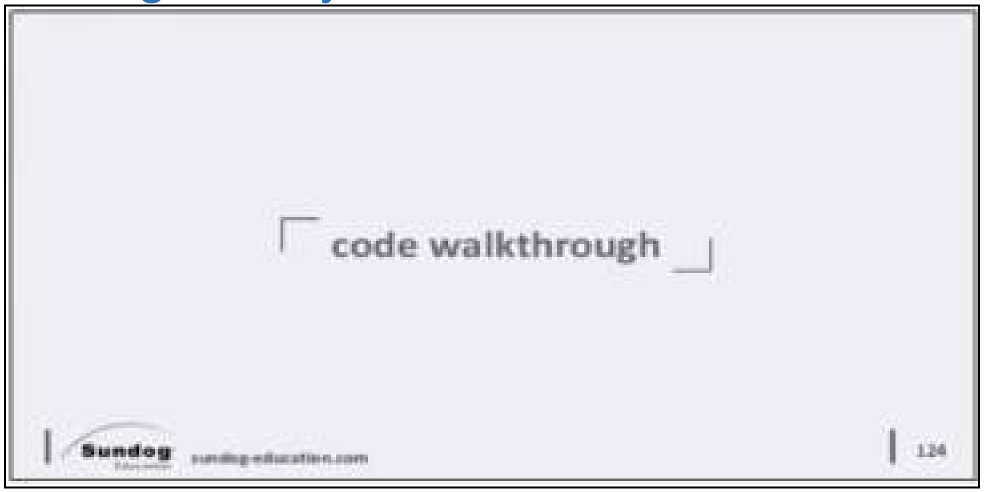
item-based collaborative filtering

Sundog Education sundog-education.com 123

So let's go back to our friend Bob. All we know about Bob is that he liked Star Wars – maybe he's a new user and that's the only thing we know about him so far.

We can now consult our item-to-item similarity matrix to look up other movies similar to Star Wars based on the ratings of other users who watched Star Wars in the past. Interestingly, this technique not only picks up The Empire Strikes Back, but also picks up Indiana Jones and The Incredibles because Ann liked those too. So even in this tiny example data set, we're getting more interesting results from item-based collaborative filtering than we did with user-based collaborative filtering .

Coding Activity


code walkthrough

Sundog sundog-education.com 124

So let's play around with item-based collaborative filtering. Open Spyder back up, and take a look at SimpleItemCF.py.

As you might expect, it looks a lot like SimpleUserCF.py, because the general approach is the same as user-based collaborative filtering – we're just focusing on relationships between items instead of users.

The first difference you'll see is that when we set up the sim_option s , we're passing false for "user based." This tells the surprise library to generate an item-to-item similarity matrix, using cosine as its similarity metric, instead of a user-to-user similarity matrix.

Next we pull of the top-K highest rated items for our test user, and then we look up all of the items similar to each of those items. These become our recommendation candidates, and as we build up our list of candidates, we score them by their similarity score to the item our user rated, and by the rating our user gave that item .

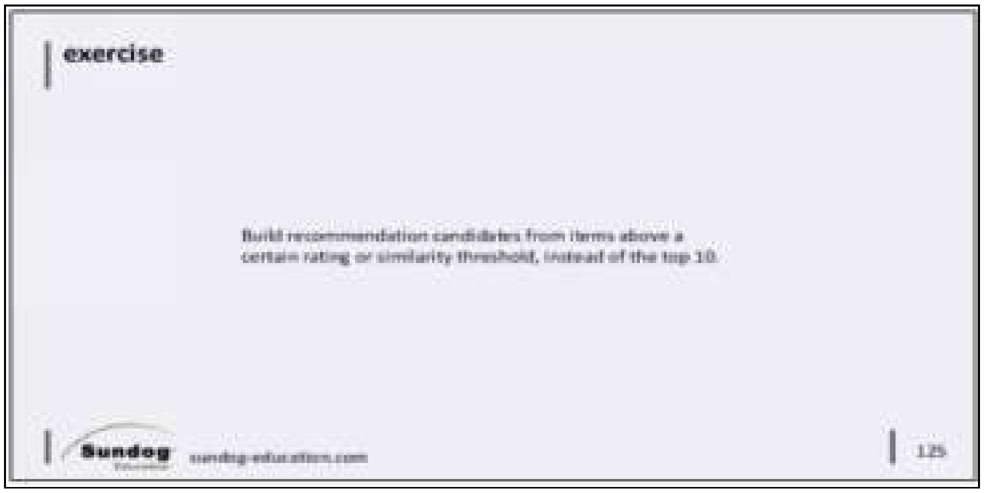
The rest of the code is the same – we build up a dictionary of stuff our user has already seen, and while we're printing out the top-scoring results, we filter those out from the final results.

If you're new to Python, it's probably a good idea to pause here for a bit and study this code a little more closely.

But once it's sunk in - let's run it and see what happens!

This too didn't take too long – but the results are interesting. We're getting much more obscure recommendations than we did with user-based collaborative filtering. However, these were probably really good recommendations for user number 85, whose ratings all date back to the late 1990's. Because all we know is the movies he liked from 20 years or more ago, we're picking up items similar to those older movies – and those tend to be older movies, as well. So there's a bit of bias in the data here that's coloring our results to some extent.

It's a good cautionary tale – you can use tools like this to sample the recommendations an algorithm will give you, but you have to be careful when judging these results subjectively. As much as I may have similar tastes to user number 85, I'm not him, and I'm not even in the same time he was in when he left his ratings. Remember item-based collaborative filtering is what Amazon used with outstanding success, so if you were to prematurely discard this algorithm because its results on one test user didn't resonate with you, that would be a big loss. You need to test it out on several real people if possible, and then move to a large scale A/B test to see if this algorithm really is better or worse than whatever you might have today.


exercise

Build recommendation candidates from items above a certain rating or similarity threshold, instead of the top 10.

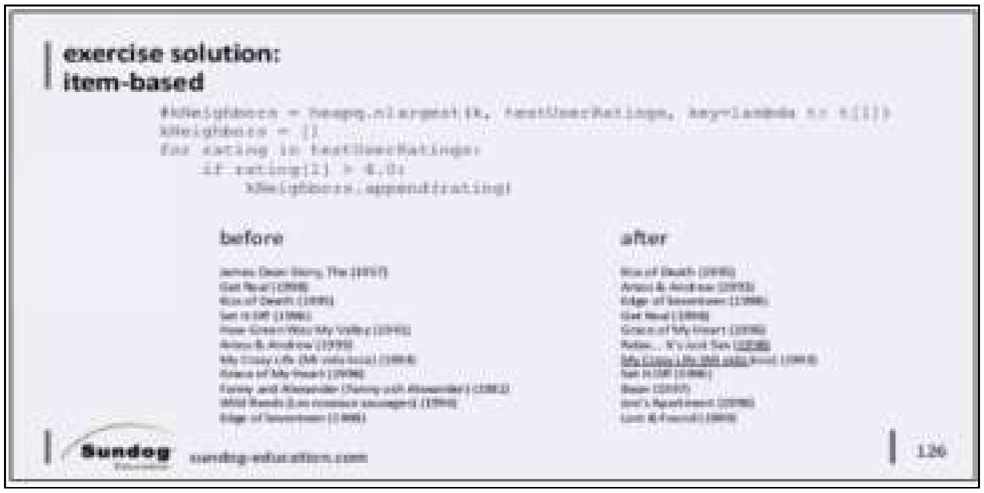
Sundog Education sundog-education.com 125

Let's fiddle with our results a bit, because as we've said there are many ways to implement user based and item based collaborative filtering.

One thing we're doing that's kind of arbitrary is pulling off the top-10 highest rated items for a user when generating item-based recommendations, or the top-10 most similar users when finding user-based recommendations. That seems like kind of an arbitrary cutoff. Maybe it would be better if instead of taking the top-K sources for recommendation candidates, we just used any source above some given quality threshold.

For example, maybe any movie a user rated higher than 4 stars should generate item-based recommendation candidates, no matter how many or how few of them there may be. Or any user that has a cosine similarity greater than 0.95 should be used to generate candidates in user-based recommendations.

This is a pretty easy change to make, but if you're new to Python it's a good learning experience. Go give it a try, and in the next slides I'll show you how I did it.



exercise solution: item-based

```
#kNeighbors = heapq.nlargest(k, testUserRatings, key=lambda t: t[1])
kNeighbors = []
for rating in testUserRatings:
    if rating[1] > 4.0:
        kNeighbors.append(rating)
```

before

James Dean Story, The (1957)
Get Real (1998)
Kiss of Death (1995)
Set It Off (1996)
How Green Was My Valley (1941)
Amos & Andrew (1993)
My Crazy Life (Mi vida loca) (1993)
Grace of My Heart (1996)
Fanny and Alexander (Fanny och Alexander) (1982)
Wild Reeds (Les roseaux sauvages) (1994)
Edge of Seventeen (1998)

after

Kiss of Death (1995)
Amos & Andrew (1993)
Edge of Seventeen (1998)
Get Real (1998)
Grace of My Heart (1996)
Relax... It's Just Sex (1998)
My Crazy Life (Mi vida loca) (1993)
Set It Off (1996)
Bean (1997)
Joe's Apartment (1996)
Lost & Found (1999)

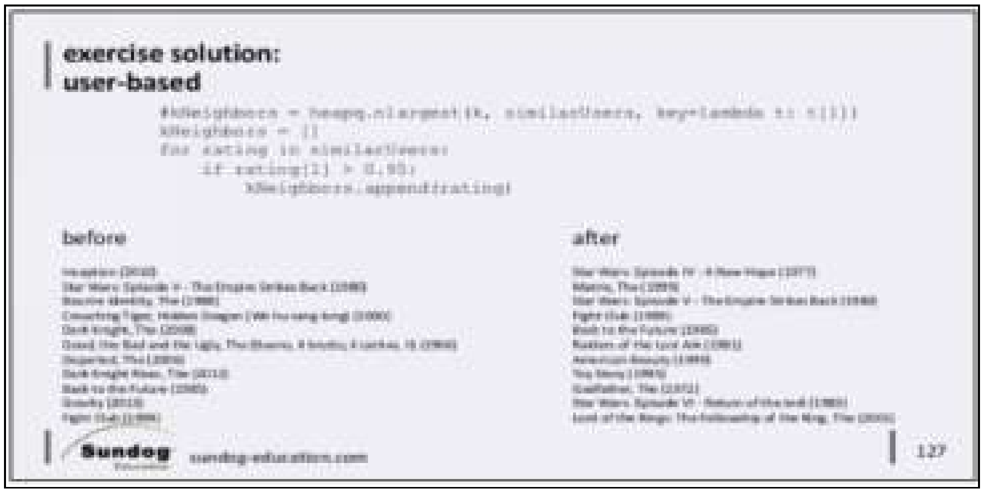
Sundog Education sundog-education.com 126

So here’s how I went about this – I just commented out the existing line in SimpleItemCF.py that generated the top K highest rated movies from our test user, and replaced it with some simple code that goes through and adds any rating above 4 stars to the list of movies that generate recommendation candidates.

Here’s the list of movie recommendations that item-based similarities produced before making this change, where we used a top-K approach. And here’s the list after this change.

It’s hard to say which set of results is “better,” but what’s remarkable is how different these lists are. We do see some of the same movies bubbling up to the top, such as “Get Real” and “Kiss of Death,” but our top pick on the left is totally gone on the right! With this change, we displaced some of the original results with different titles, such as “Bean” and “Joe’s Apartment.” These replaced titles on the left including “Wild Reeds” and “Fanny and Alexander.” It’s interesting that I’ve actually heard of the new titles and never heard of the ones that got dropped, so subjectively it seems like this might be a promising change to test on real users .

Let’s have a look at how this same idea changes our results with user-based recommendations.



exercise solution: user-based

```
#kNeighbors = heapq.nlargest(k, similarUsers, key=lambda t: t[1])
kNeighbors = []
for rating in similarUsers:
    if rating[1] > 0.95:
        kNeighbors.append(rating)
```

before

Inception (2010)
Star Wars: Episode V - The Empire Strikes Back (1980)
Bourne Identity, The (1988)
Crouching Tiger, Hidden Dragon (Wo hu cang long) (2000)
Dark Knight, The (2008)
Good, the Bad and the Ugly, The (Buono, il brutto, il cattivo, Il) (1966)
Departed, The (2006)
Dark Knight Rises, The (2012)
Back to the Future (1985)
Gravity (2013)
Fight Club (1999)

after

Star Wars: Episode IV - A New Hope (1977)
Matrix, The (1999)
Star Wars: Episode V - The Empire Strikes Back (1980)
Fight Club (1999)
Back to the Future (1985)
Raiders of the Lost Ark (1981)
American Beauty (1999)
Toy Story (1995)
Godfather, The (1972)
Star Wars: Episode VI - Return of the Jedi (1983)
Lord of the Rings: The Fellowship of the Ring, The (2001)

Sundog sundog-education.com 127

Here’s how I implemented the same idea on user-based collaborative filtering. I commented out the line that selected the top K users most similar to our test user, and replaced it with code that builds up a list of arbitrary length of any users with a cosine similarity score of 0.95 or higher.

The results we started with look pretty compelling, but let’s have a look at the results we get when making this change. Again, it’s hard to say which is better without testing it on real people at large scale – but the thing that strikes me the most is that we seem to have lost recommendations for newer movies by making this change. Again, that may just be a side effect of the time bias in our data set, because we end up considering many more users than we did previously. Maybe 0.95 isn’t a high enough threshold – try a higher threshold, like 0.98, and you’ll find that newer movies such as the Dark Knight and Inception return to the results .

So again, we’ve found a relatively small change that can make a huge difference in the results. Often you don’t just want to test different recommendation algorithms, you want to test different variations and parameters on those algorithms. In this case, I’d not only want to test the idea of using a threshold instead of a top-K approach, I’d also want to test many different threshold values to find the best one. In the real world, you’ll find that your biggest problem is just not having time to run all of the different experiments you want to run to make your recommendations better!

Now, although we can't measure accuracy with user based or item based collaborative filtering because they don't make rating predictions, we can still measure hit rate, because it is still just a top-N recommender.

So, let's go back to Spyder and have a look at our EvaluateUserCF.py file. This looks a lot like our SimpleUserCF file, but we've set things up to do leave-one-out cross-validation, and we're using our RecommenderMetrics package to measure hit rate on it. There's not a lot of new code to talk about here – we just generate the "leave one out" test set up top here, and use that when evaluating things at the end. And instead of just printing out the results, we store them so we can measure them – and we also generate up to 40 recommendations per user instead of 10. Also, we're generating recommendations for everyone, and not just a single test user.

Let's run it out of curiosity….

That was surprisingly fast! Even though we're generating recommendations for everybody, and not just one person. As we've mentioned, one really nice property of collaborative filtering is how quickly it can generate recommendations for any given individual once we've built up the similarity matrix we need.

And the end result of 5.5% is pretty darn good – this validates our intuition just looking at the user-based collaborative filtering results subjectively that it looked promising.

exercise

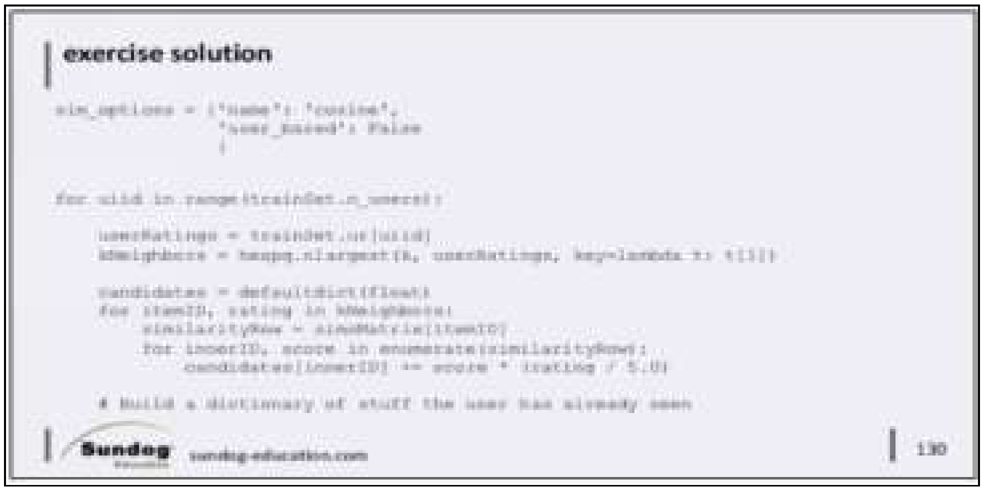
measure the hit-rate of item-based collaborative filtering

Sundog Education sundog-education.com 129

So, now I’m going to give you a slightly more challenging exercise.

Our intuition was that for our particular data set, the user-based results looked more exciting than the item-based results, which seemed to be focused on older, more obscure film titles. But we weren’t sure if they were really worse recommendations, or if they were actually better recommendations back at the time when our test user left his ratings.

So let’s measure it. Try modifying the EvaluateUserCF.py function to measure item-based collaborative filtering instead of user-based, and see how the hit rate compares. By looking at the SimpleItemCF.py file we used earlier, you should be able to pull this off. So give that a try if you’re up for it, and I’ll show you how I did it in the next slides.



exercise solution

```
sim_options = {'name': 'cosine',
               'user_based': False
               }

for uiid in range(trainSet.n_users):

    userRatings = trainSet.ur[uiid]
    kNeighbors = heapq.nlargest(k, userRatings, key=lambda t: t[1])

    candidates = defaultdict(float)
    for itemID, rating in kNeighbors:
        similarityRow = simsMatrix[itemID]
        for innerID, score in enumerate(similarityRow):
            candidates[innerID] += score * (rating / 5.0)

    # Build a dictionary of stuff the user has already seen
```

Sundog Education sundog-education.com 130

So, it should be pretty easy to adapt our code to measure user-based collaborative filtering to item-based. There are just two chunks of code that need to be changed.

The first is telling Surprise to generate item-to-item similarities by passing “False” for user-based in the sim_options structure.

Next, we just adapt the meat of the code from SimpleItemCF.py to work within our loop that iterates through every user, and not just a single test user. The rest of the code remains the same.

When we run it, the results are a bit shocking – our hit rate from item-based collaborative filtering is only 0.5%, compared to 5.5% for user-based! Why is this? We’ve convinced ourselves that item-based should be a superior approach, and that’s been proven in industry!

It’s hard to prove, but I think it’s just a quirk of the data set we are using. The MovieLens data set was initially created about 20 years ago, and it has more data from back then than it does for current movies. We’ve seen that item-based seems to focus on those older movies as a result, and this is probably leading to bad hit rates when we’re looking at more modern users in the data set. It’s just a hypothesis, but the moral of the story again is that it’s hard to really evaluate recommender systems offline, especially when you’re using historical, smaller data sets to test with. If we were to decide on an approach based on the results of this last exercise, we would conclude that user based collaborative filtering is far superior and reject the item-based approach – but if we were to test both algorithms on real-world people using real-world data in an A/B test, the results could end up being very different.

It’s also true that collaborative filtering in general is sensitive to sparse and noisy data, and we are using a relatively small data set here, which isn’t helping matters.

But if there’s one theme in this course, it’s that you shouldn’t put too much faith in offline accuracy measurements of recommender systems. Real users of your system don’t care one lick about how well you can predict their ratings for movies they’ve already seen, and that’s what all of these offline metrics measure.

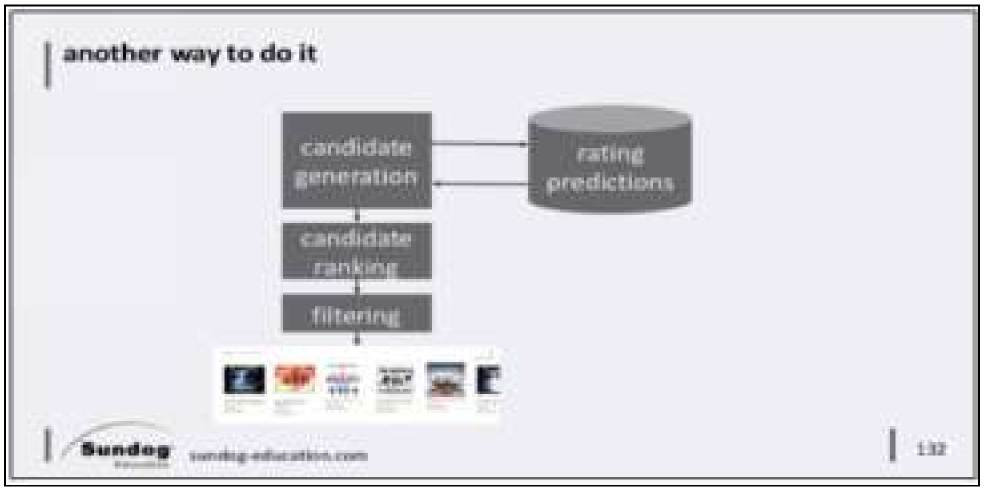
KNN Recommenders

k-nearest-neighbors (knn) recommenders

Sundog sundog-education.com 131

As we've seen, it's very difficult to evaluate collaborative filtering without running expensive experiments on real world users. Since they aren't based on making rating predictions, we can't really measure their accuracy offline.

So, the concept of collaborative filtering has been applied to recommender systems that do make rating predictions, and these are generally referred to in the literature as "KNN recommenders." Let's have a look at how they work.

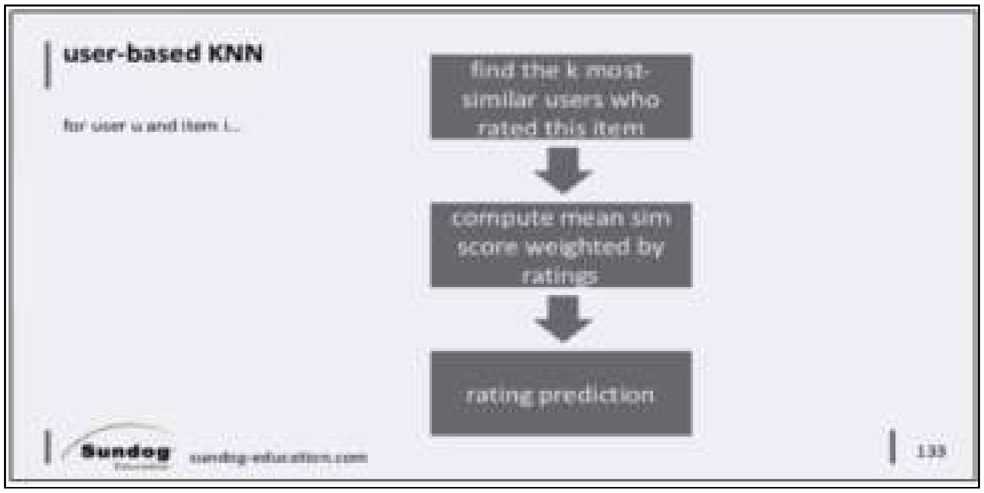

another way to do it

Sundog sundog-education.com 132

Let's revisit the architecture of a recommender system based on rating predictions. In this sort of system, we generate recommendation candidates by predicting the ratings of everything a user hasn’t already rated, and selecting the top K items with the highest predicted ratings. From there, everything else works more or less the same way.

This obviously isn’t a terribly efficient approach, but since we’re predicting rating values, we can measure the offline accuracy the system using train/test or cross-validation, which is useful in the research world.

Our challenge, then, is to somehow predict ratings using our data on item or user similarities, so we can still call this collaborative filtering. One way is to base those predictions on the top K items or users with the best similarity scores, which is why we call this approach “K nearest neighbors.” Let’s dive into how that works.



user-based KNN

for user u and item i...

find the k most-similar users who rated this item
compute mean sim score weighted by ratings
rating prediction

Sundog Education sundog-education.com

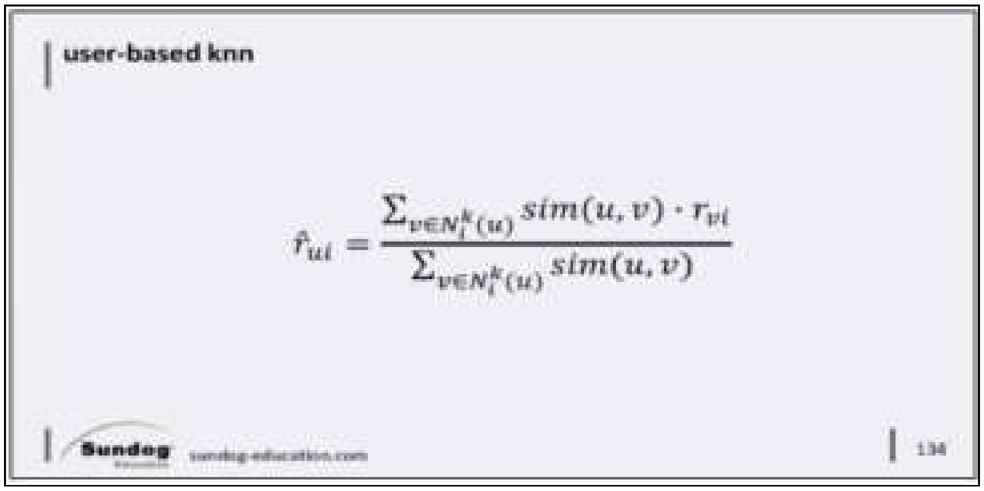
133

So for the user-based KNN approach, here’s how it works. Since we’re trying to predict ratings for a given user and item, we start by finding similar users to the user in question – but we need to restrict this to users who have also rated the item in question. We restrict this to some number K of similar users. So, we end up with a set of ratings from K similar users for this item – these are our “K nearest neighbors”, together with a measure of how similar the user is to the user we’re making a prediction for.

Next we do a weighted average of the user similarity scores for each associated rating, weighted by the ratings themselves.

This gives us a reasonable prediction of how this user might rate this item, based on similar users who also rated this item.

You can see it's similar in spirit to the user-based collaborative filtering approach we described earlier, but it's completely different in how it works. It still uses a user-to-user similarity matrix at its core, but we're not just recommending stuff that people similar to you liked. Instead, we're reaching deeper into the data to try and make predictions about ratings for every possible item for every possible user. It's a more complex approach, and usually that's a bad thing.



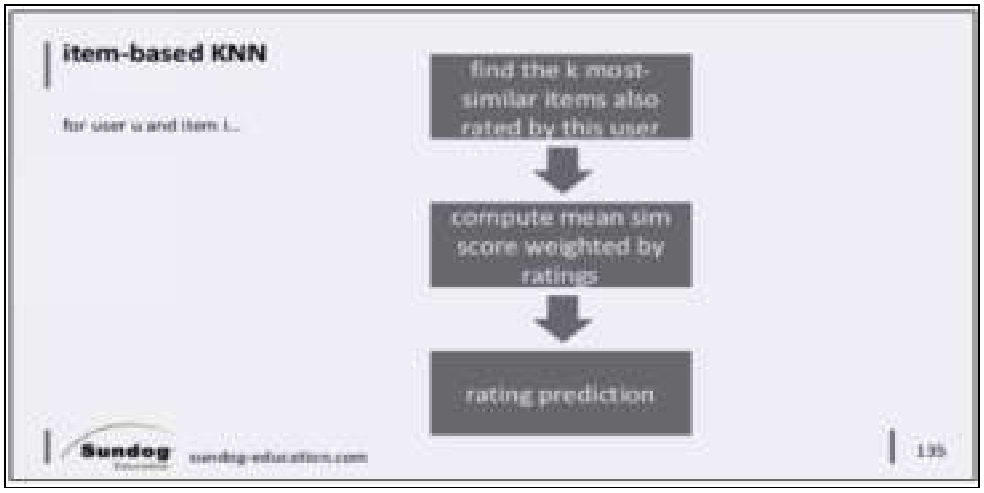
user-based knn

$$\hat{r}_{ui} = \frac{\sum_{v \in N_i^k(u)} sim(u,v) \cdot r_{vi}}{\sum_{v \in N_i^k(u)} sim(u,v)}$$

Sundog Education sundog-education.com

134

Here's the fancy math for what we just said, using the same notation as the SurpriseLib documentation. It says that the predicted rating for user u and item I is equal to the sum of the k nearest users, where we call each of these k-nearest users v. We sum up the products of the similarity scores between users u and v as we go, multiplied by the rating given to this item by user v. Then we divide by just the sum of those user similarity scores, to end up with a weighted average that can be used as a rating prediction.



item-based KNN

for user u and item i...

find the k most-similar items also rated by this user

compute mean sim score weighted by ratings

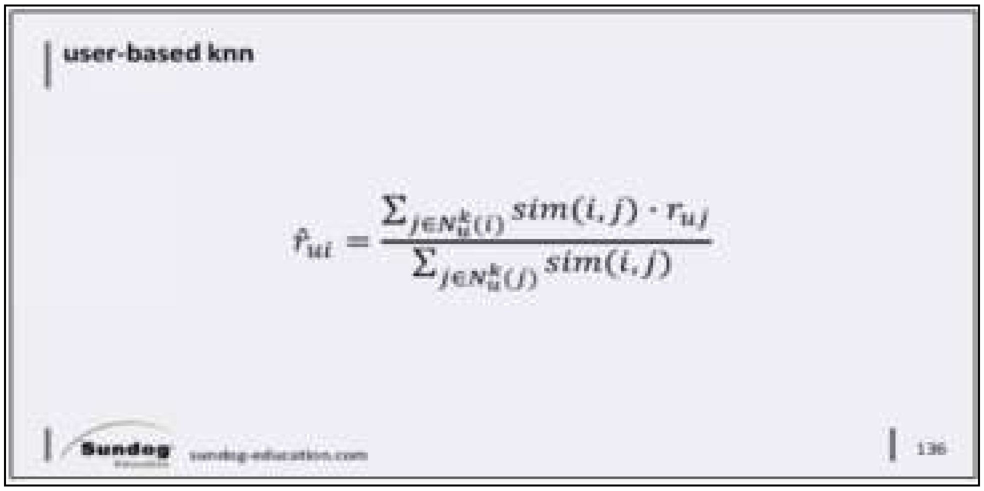
rating prediction

Sundog Education sundog-education.com

135

Item-based KNN works exactly the same, we just flip the words “user” and “item”. So to predict the rating of a user u for an item I, we start with the set of k items also rated by this user that are most similar to item i.

From there, it’s the same deal – compute the weighted average of the sims cores by rating, and you have yourself a reasonable rating prediction to work with.



user-based knn

$$\hat{r}_{ui} = \frac{\sum_{j \in N_u^k(i)} sim(i,j) \cdot r_{uj}}{\sum_{j \in N_u^k(j)} sim(i,j)}$$

Sundog Education sundog-education.com

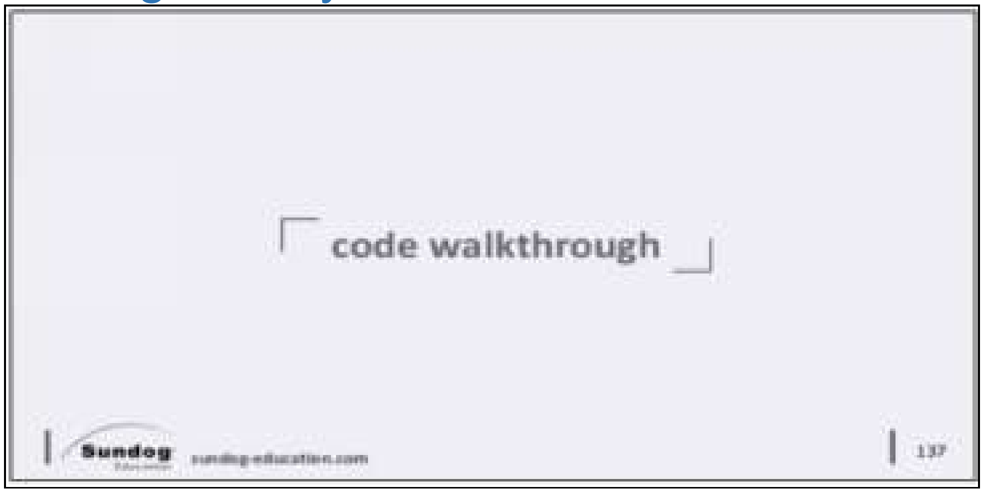
136

The math is identical, except we’re looking at similarities between items I and j instead of users u and v here. So again, the predicted rating is based on the set of the K most-similar items also rated by this user to the item in question. We sum up all the item similarity scores weighted by their ratings, and divide by the sum of the similarity scores.

It’s kind of shoe-horning the concept of collaborative filtering into a framework built around rating predictions, but it’s not that bad of a rating predictor as we’ll see. If your goal however is create top-N recommendations, you’re probably better off with the simpler, more direct approach we described earlier.

Fortunately KNN is an algorithm that’s built into SurpriseLib, so we can experiment with it without having to write much code at all. Let’s take a look .

Coding Activity


code walkthrough
Sundog
sundog-education.com
137

So let’s head back to Spyder, and this time have a look at the KNNBakeOff.py file inside our CollaborativeFiltering folder.

As I said, SurpriseLib will do most of the work for us here – we just have to tell it what we want it to do. We’re importing the KNNBasic package, because that’s what implements both user-based and item-based KNN recommendations as we’ve described them.

All we do is create one instance with user-based set to true, another to false, and pit them against each other, and also the random recommender as a baseline. We’ll print out accuracy metrics, but in the interest of time we’ll just sample the top-N recommender results. You can certainly pass true to the Evaluate function if you want to get hit rate metrics as well, but it will take a lot longer to run. Let’s kick it off, and we’ll come back in a few minutes when it’s done.

So let’s take a look at the accuracy results. This is interesting – both user based and item based had about the same accuracy score here. User based came out slightly better, but it’s so slight I don’t think we can really read anything into that. Both outperformed random recommendations substantially, so that’s a good thing – it’s doing something useful, for sure.

So just based on those accuracy scores, we might conclude that KNN recommendations are a pretty good idea – certainly one worth trying out in production.

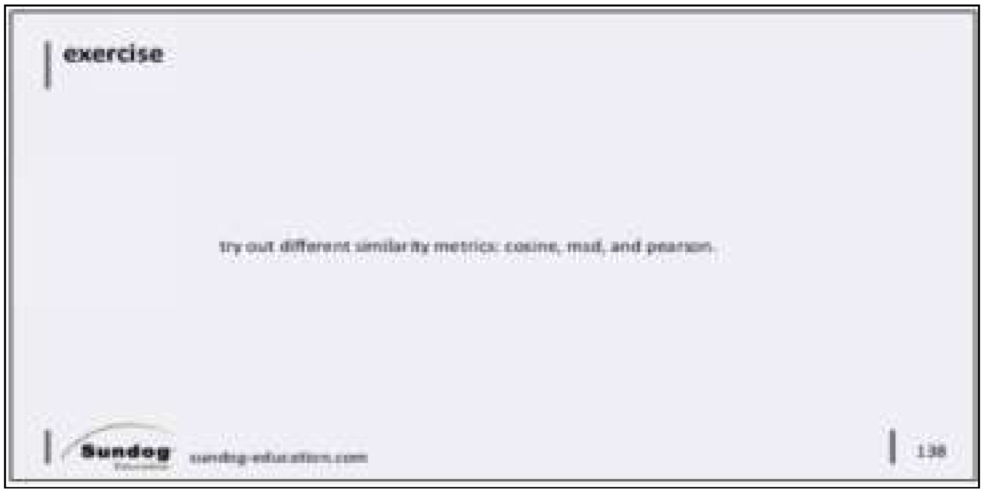
But let’s look at the actual recommendations we get back for our test user, our friend user number 85.

I don’t know about you, but I haven’t heard of a single one of these recommended movies in the user-based results. It’s hard to believe these movies are such great recommendations that they warrant a 5-star prediction from someone with similar interests to myself, when I’ve never even heard of any of them. Maybe they’re great and I would love them if I watched them, but it seems suspect.

The item-based results aren’t any better, in terms of recognizability. I’ve never heard of these, either. Even if these are great recommendations, it’s not really generating any user trust in the recommender system if all you recommend are obscure items.

The random recommendations actually look a lot better from a subjective standpoint when we look at the top-N results.

So, on the surface it looks like we may have made a system that’s pretty good at predicting ratings of movies people have already seen, but might not be very good at producing top-N recommendations. This is yet another example of how focusing on offline accuracy metrics can lead you to the wrong conclusions.



exercise

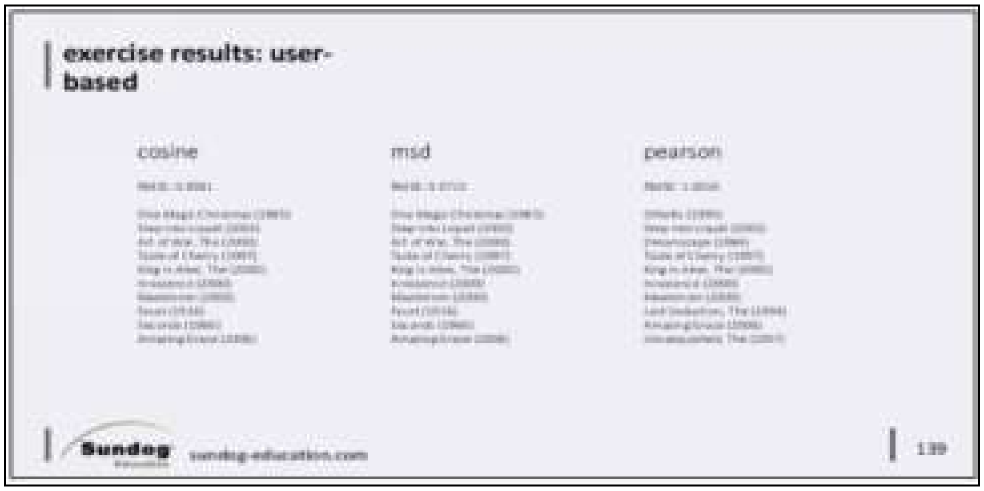
try out different similarity metrics: cosine, msd, and pearson.

Sundog sundog-education.com

138

Maybe we can improve on these results somehow by tweaking the parameters of the algorithm. As an exercise, try out the different similarity measures SurpriseLib offers. Right now we're using cosine, but how about msd and pearson? Give it a go, and compare the results you get from each. Are they substantially different?

After you've experimented with these different metrics, continue to the next slide and I'll show you my results.



exercise results: user-based

cosine	msd	pearson
RMSE: [illegible]	RMSE: [illegible]	RMSE: [illegible]
One Magic Christmas (1985)	One Magic Christmas (1985)	Othello (1995)
Step Into Liquid (2003)	Step Into Liquid (2003)	Step Into Liquid (2003)
Art of War, The (2000)	Art of War, The (2000)	Dreamscape (1984)
Taste of Cherry (1997)	Taste of Cherry (1997)	Taste of Cherry (1997)
King Is Alive, The (2000)	King Is Alive, The (2000)	King Is Alive, The (2000)
Innocence (2000)	Innocence (2000)	Innocence (2000)
Maelström (2000)	Maelström (2000)	Maelström (2000)
Faust (1926)	Faust (1926)	Last Seduction, The (1994)
Seconds (1966)	Seconds (1966)	Amazing Grace (2006)
Amazing Grace (2006)	Amazing Grace (2006)	Unvanquished, The (1957)

Sundog sundog-education.com

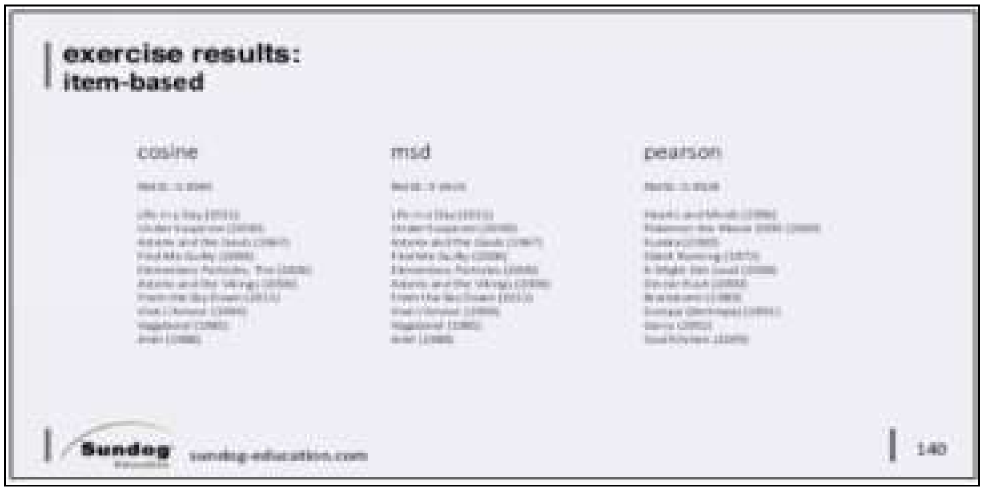
139

So I tabulated my results in using different similarity metrics with KNN for both the user-based and item-based cases. Let's start with user-based.

If we look at the RMSE error scores, we might conclude that mean squared distance, or msd, significantly outperforms the cosine metric

– it's .97 as opposed to .99. But look at the actual top N recommendations for our test user – they are exactly the same! This means the math behind MSD leads to more accurate rating predictions, but the ranking of those predictions is more or less the same. So from a practical standpoint, that increased accuracy doesn't matter. I just can't say enough that focusing on accuracy alone is not a good idea – you need to look at real-world outputs of the system together with it.

Pearson at least delivers some slightly different results, but are they better? Well not according to the RMSE score, although in theory Pearson is supposed to normalize things in such a way that should be beneficial. Is "Othello" a better top recommendation than "One Magic Christmas?" Well for me, it certainly would be. So again, what the accuracy metric is telling you might run counter to what real-world users might respond to.

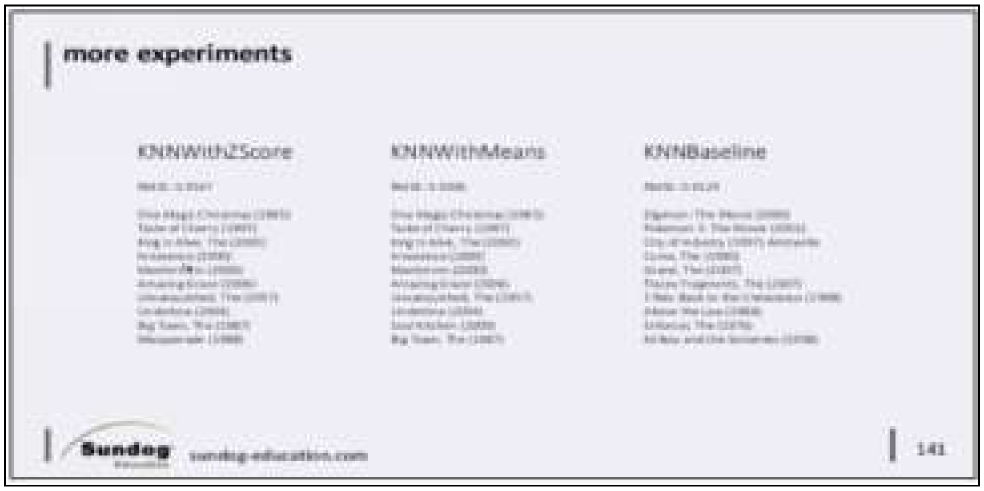


exercise results: item-based

cosine	msd	pearson
RMSE: [illegible]	RMSE: [illegible]	RMSE: [illegible]
Life in a Day (2011)	Life in a Day (2011)	Hearts and Minds (1996)
Under Suspicion (2000)	Under Suspicion (2000)	Pokemon the Movie 2000 (2000)
Asterix and the Gauls (1967)	Asterix and the Gauls (1967)	[illegible] (2005)
Find Me Guilty (2006)	Find Me Guilty (2006)	Silent Running (1972)
Elementary Particles, The (2006)	Elementary Particles (2006)	It Might Get Loud (2008)
Asterix and the Vikings (2006)	Asterix and the Vikings (2006)	[illegible] (2003)
From the Sky Down (2011)	From the Sky Down (2011)	Brainstorm (1983)
Vive L'Amour (1994)	Vive L'Amour (1994)	Europa (Zentropa) (1991)
Vagabond (1985)	Vagabond (1985)	Gerry (2002)
Ariel (1988)	Ariel (1988)	Soul Kitchen (2009)

Sundog sundog-education.com 140

It's a similar story looking at item-based KNN. No matter what the similarity metric, we end up with a lot of obscure movies, many of them foreign. And there's an even bigger difference between cosine and msd's RMSE scores, yet they produce exactly the same results. Pearson fares better on RMSE in the item-based case, and it also generates a set of recommendations that is substantially different. Subjectively, I recognize at least a couple of these recommendations as ones I might enjoy – so this tells me that in a real world setting, experimenting with Pearson similarity on item-based collaborative filtering may be a worthwhile experiment.



more experiments

KNNWithZScore	KNNWithMeans	KNNBaseline
RMSE: [illegible]	RMSE: [illegible]	RMSE: [illegible]
One Magic Christmas (1985)	One Magic Christmas (1985)	Digimon: The Movie (2000)
Taste of Cherry (1997)	Taste of Cherry (1997)	Pokemon 3: The Movie (2001)
King Is Alive, The (2000)	King Is Alive, The (2000)	City of Industry (1997) [illegible]
Innocence (2000)	Innocence (2000)	[illegible]
Maelström (2000)	Maelström (2000)	[illegible]
Amazing Grace (2006)	Amazing Grace (2006)	[illegible]
Unvanquished, The (1957)	Unvanquished, The (1957)	[illegible]
Undertow (2004)	Undertow (2004)	Above the Law (1988)
Big Town, The (1987)	Soul Kitchen (2009)	[illegible]
Masquerade (1988)	Big Town, The (1987)	[illegible]

Sundog sundog-education.com 141

Surpriselib also offers some variants on KNN recommendations we can try as well. I evaluated them on the item-based case just to see if any of them produce markedly better results.

The short answer is no – although on paper they appear to have much better accuracy results, the top-N results are still pretty obscure and random.

This should be expected, because each of these algorithms attempt to somehow “baseline” their ratings predictions to be more consistent with how the real ratings are distributed in our training data. Without getting too far into the details, the “baseline” variant predicts ratings relative to some “baseline” average rating score across the data set. And the “means” and “z score” variants attempt to account for differences in the baselines for individual users, like we described when we talked about the adjusted cosine metric. In every case, you can see we did in fact achieve better accuracy scores. But the end results don’t care about accuracy; they only care about the order of the predicted ratings we end up with. So, we’re left with recommendations that don’t seem all that inspiring again – even if the accuracy metrics look exciting.

why is knn so bad?

Sundog sundog-education.com

142

So why does KNN have so much trouble presenting compelling items to our users? Well, it’s actually pretty well known that KNN doesn’t work well in practice, and unfortunately some people conclude that collaborative filtering in general is some naïve approach that should be replaced with completely different techniques. But as we’ve seen, collaborative filtering isn’t the problem, it’s forcing collaborative filtering to make rating predictions that’s the problem. Remember back when we started this section? We had some pretty exciting results when we just focused on making top-N recommendations and completely forgot about optimizing for rating accuracy.

And, it turns out that’s what’s at the heart of the problem. Ratings are not continuous in nature, and KNN treats them as though they are continuous values that can be predicted on a continuous scale. If you really wanted to go with KNN, it would be more appropriate to treat it as a rating classification problem than as a rating prediction problem. KNN is also very sensitive to sparse data, so in all fairness you might get better results if you didn’t just have 100,000 ratings to play with .

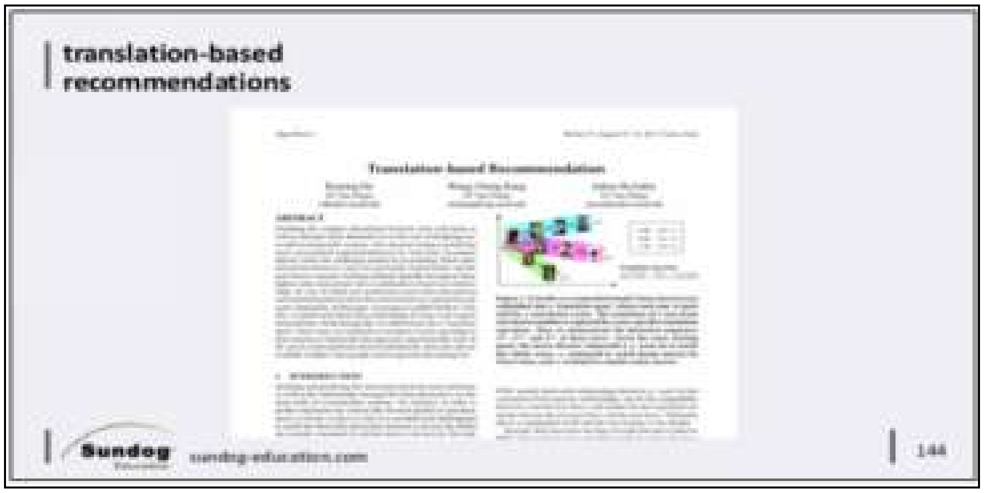
But the most fundamental thing is that accuracy isn’t everything! The main reason KNN produces underwhelming results is because it’s trying to solve the wrong problem.

Bleeding Edge Alert! Translation-Based Recommendations

bleeding edge alert!

Sundog sundog-education.com 143

It’s time for another bleeding edge alert! This is where we talk about some recent research that has promising results, but hasn’t yet made it into the mainstream with recommender systems yet.


translation-based recommendations

Translation-based Recommendation

Sundog sundog-education.com 144

I want to mention an interesting paper from the 2017 Conference on Recommender Systems. It’s called Translation-based recommendations, and it comes from a team at the University of California at San Diego.

translation-based recommendations

https://sites.google.com/view/ruining-he/publications

Sundog sundog-education.com 145

If you want to check out the original paper, as well as the data and code behind it, you can find it at this link .

The idea behind it is that users are modeled as vectors moving from one item to another in a multi-dimensional space, and you can predict sequences of events – like which movie a user is likely to watch next – by modeling these vectors.

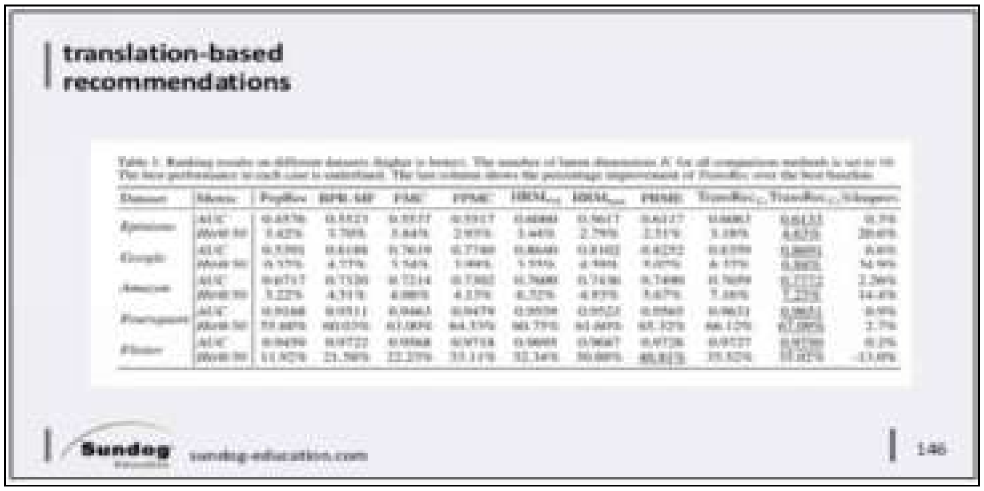
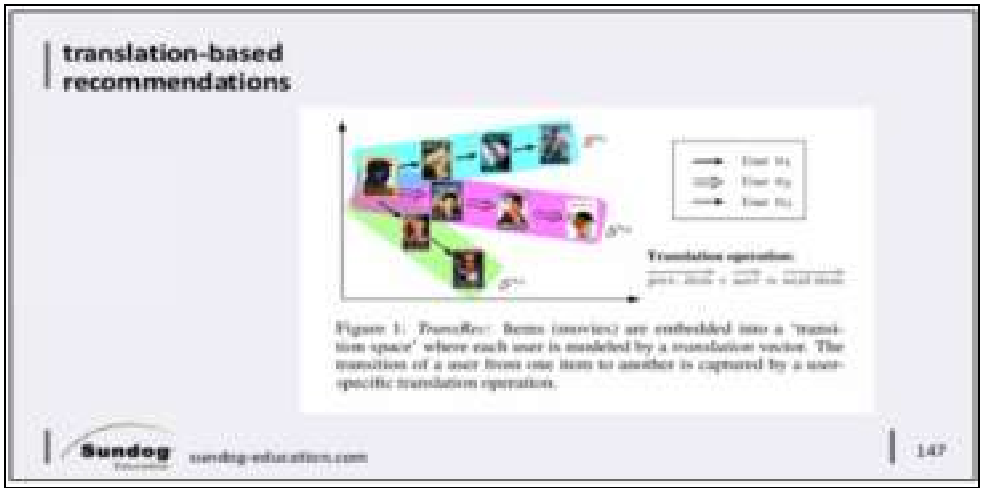

translation-based recommendations

Table 3: Ranking results on different datasets (higher is better). The number of latent dimensions K for all comparison methods is set to 10. The best performance in each case is underlined. The last column shows the percentage improvement of TransRec over the best baseline.

Dataset	Metric	PopRec	BPR-MF	FMC	FPMC	HRM$_{avg}$	HRM$_{max}$	PRME	TransRec$_{L_1}$	TransRec$_{L_2}$	%Improv.
Epinions	AUC	0.4576	0.5523	0.5537	0.5517	0.6060	0.5617	0.6117	0.6063	0.6133	0.3%
	Hit@50	3.42%	3.70%	3.84%	2.93%	3.44%	2.79%	2.51%	3.18%	4.63%	20.6%
Google	AUC	0.5391	0.6188	0.7619	0.7740	0.8640	0.8102	0.8252	0.8359	0.8691	0.6%
	Hit@50	0.57%	4.77%	3.54%	3.99%	3.55%	4.59%	5.07%	6.37%	6.84%	34.9%
Amazon	AUC	0.6717	0.7320	0.7214	0.7302	0.7680	0.7438	0.7480	0.7609	0.7772	2.36%
	Hit@50	3.22%	4.31%	4.08%	4.23%	6.52%	4.95%	5.67%	7.18%	7.23%	14.4%
Foursquare	AUC	0.9168	0.9511	0.9463	0.9479	0.9559	0.9523	0.9565	0.9631	0.9651	0.9%
	Hit@50	55.60%	60.03%	63.00%	64.53%	60.75%	61.60%	65.32%	66.12%	67.09%	2.7%
Flixster	AUC	0.9459	0.9722	0.9568	0.9718	0.9695	0.9687	0.9728	0.9727	0.9750	0.2%
	Hit@50	11.92%	21.58%	22.23%	33.11%	32.34%	30.88%	40.81%	35.52%	35.02%	-13.6%

Sundog sundog-education.com 146

The reason this paper is exciting is because it out-performed all of the best existing methods for recommending sequences of events in all but one case in one data set. And I also like that they measured their results based on hit rate instead of accuracy; they are focused on the right thing.



translation-based recommendations

Figure 1: TransRec: Items (movies) are embedded into a 'transition space' where each user is modeled by a translation vector. The transition of a user from one item to another is captured by a user-specific translation operation.

Sundog sundog-education.com 147

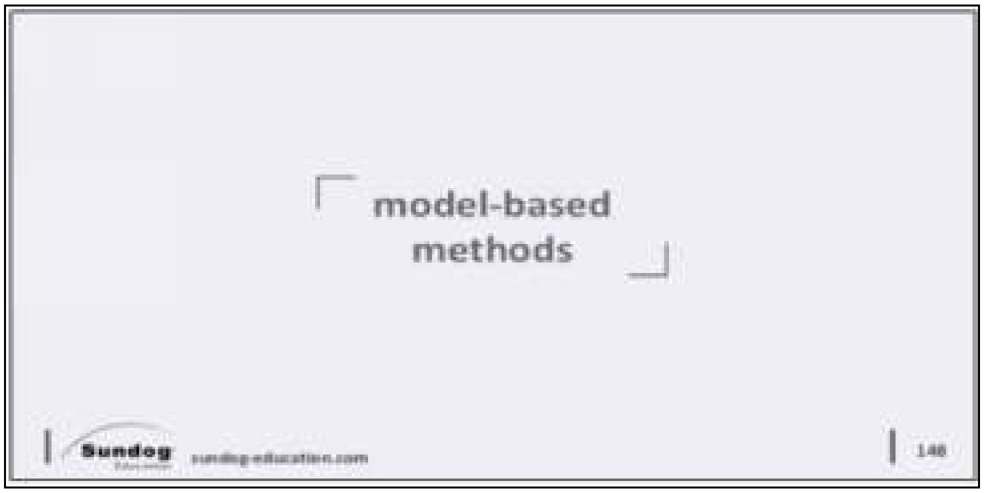
This image from the paper illustrates the basic idea. You position individual items, like movies, in a “transition space” where neighborhoods within this space represent similarity between items, so items close together in this space are similar to each other. The dimensions correspond to complex transition relationships between items. Since this technique depends on arranging items together into local, similar “neighborhoods,” I still classify it as a neighborhood-based method.

In this space, we can learn the vectors associated with individual users. Perhaps a user who watches a Tom Cruise movie is likely to move along to the next Tom Cruise movie, for example, and that transition would be represented by a vector in this space. We can then predict the next movie a user is likely to watch by extrapolating along the vector we’ve associated with that user.

The paper doesn’t go into a lot of detail on how it’s done, but they do provide the code and data they used, so if you can read C++ code, you can reverse engineer what they did .

It’s all very advanced stuff, but it seems to work. So if you find yourself in a situation where you need to predict a sequence of events, like which movies or videos a person is likely to watch next given their past history, you might want to do a search for translation-based recommendations and see how it’s coming along in the real world. Mainly, I just want you to remember the term “translation based recommendations” so you can remember to check up on it if you need to recommend sequences someday. Right now, it’s bleeding edge – but it looks promising.

Model-Based Methods



model-based methods

Sundog sundog-education.com 148

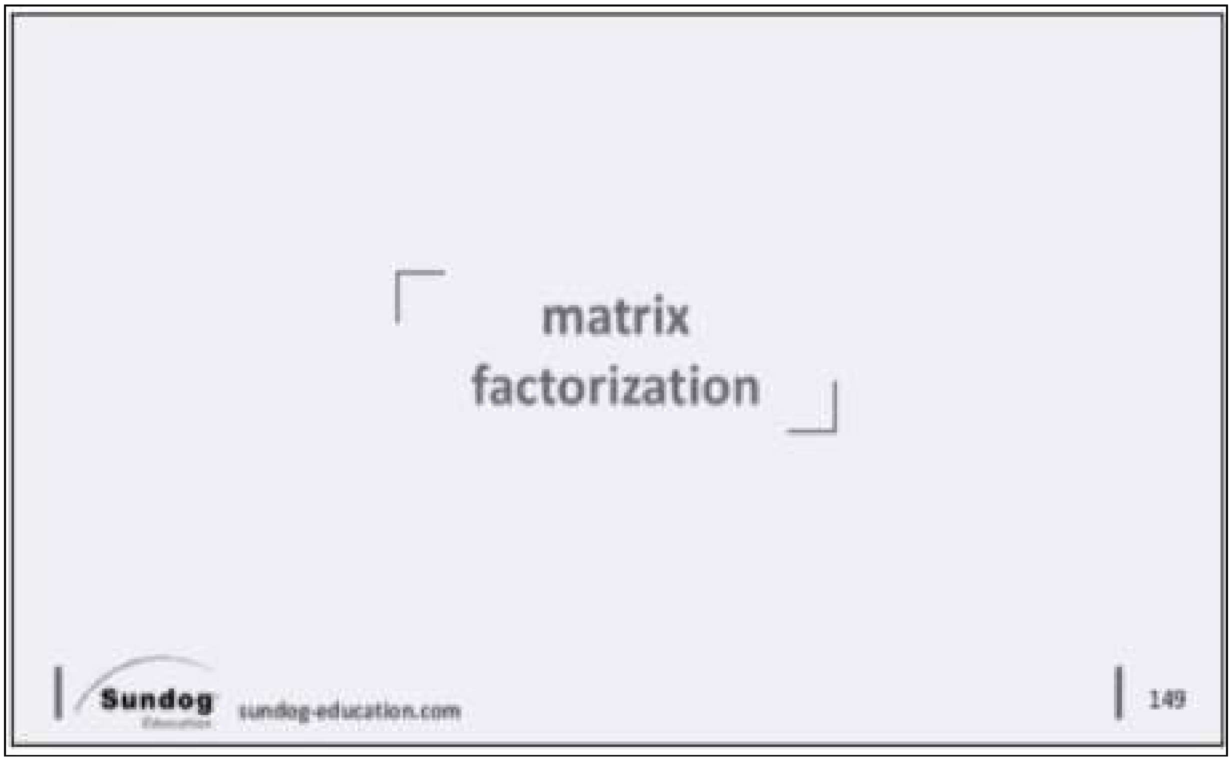
So as we've seen, collaborative filtering can produce great results that have been shown to work really well in real, large-scale situations. So why would we look for something even better?

Collaborative filtering has been criticized as having limited scalability, since computing similarity matrices on very large sets of items or users can take a lot of computing horsepower. I don't really buy this, however – as we've seen, using item-based collaborative filtering reduces the complexity of that substantially, to the point where you can compute an entire similarity matrix for extremely large product catalogs on a single machine. And even if you couldn't, technologies such as Apache Spark allow you distribute the construction of this matrix across a cluster if you need to.

A legitimate problem with collaborative filtering, though, is that it is sensitive to noisy data and sparse data. You'll only get really good results if you have a large data set to work with that's nice and clean .

So, let's explore some other ways to make recommendations. We'll group these together under the label "model-based methods," since instead of trying to find items or users that are similar to each other, we'll instead apply data science and machine learning techniques to extract predictions from our ratings data. Machine learning is all about training models to make predictions, so we’ll treat the problem of making recommendations the same way – we will train models with our user ratings data, and use those models to predict the ratings of new items by our users.

This puts us squarely in the space of the rating prediction architecture that SurpriseLib is built around, and although we’ve shown it’s not always the most efficient approach, it allows us to repurpose machine learning algorithms to build recommender systems – some of which are very good at predicting ratings. Whether they make for good top-N recommendations is something we’ll need to find out.



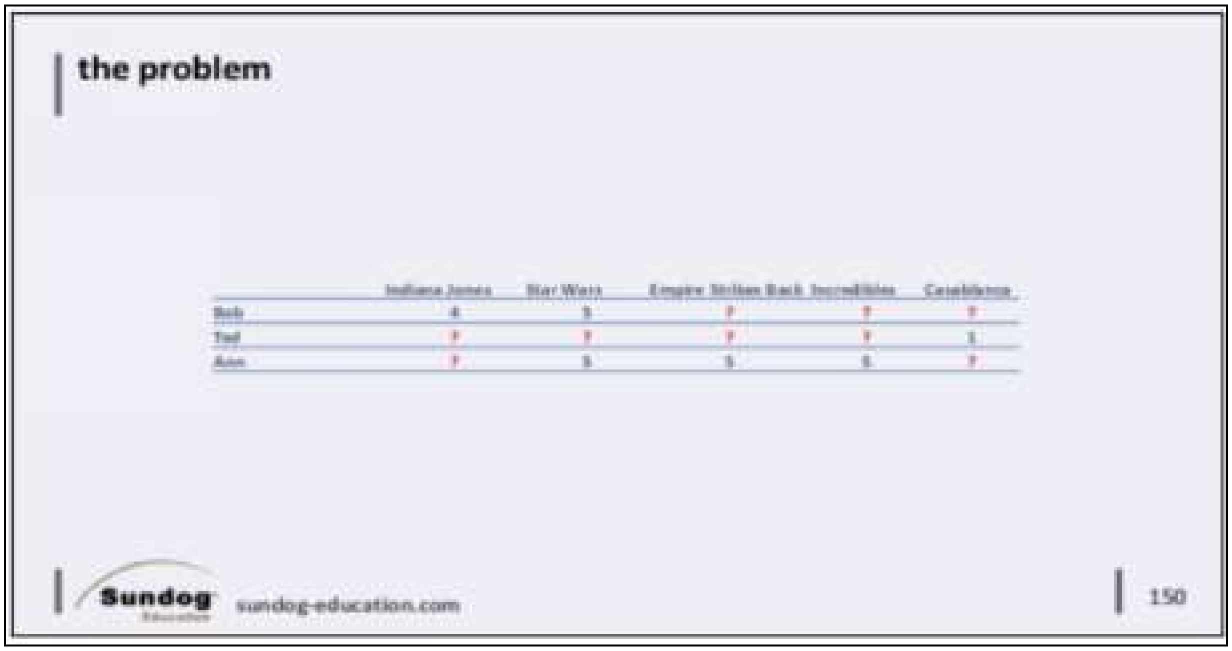
matrix factorization

Sundog Education sundog-education.com

149

Matrix Factorization

There are a wide variety of techniques that fall under the category of matrix factorization. These algorithms can get a little bit creepy – they manage to find broader features of users and items on their own, like “action movies” or “romantic” – although the math doesn’t know what to call them, they are just described by matrices that describe whatever attributes fall out of the data.



the problem

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5	?	?	?
Ted	?	?	?	?	1
Ann	?	5	5	5	?

Sundog Education sundog-education.com

150

The general idea is to describe users and movies as combinations of different amounts of each feature. For example, maybe Bob is defined as being 80% an action fan and 20% a comedy fan – we'd then know to match him up with movies that are a blend of about 80% action and 20% comedy. That's the general idea; let's dig into how it works.

Working backwards from what we want to achieve is usually a good approach, so what are we trying to do when predicting ratings for users? Well, you can think of it as a filling in a really sparse matrix.

We can think of all of our ratings as existing in a 2D matrix, with rows representing users, and columns representing items – in our case, movies. The problem is most of the cells in this matrix are unknown; our challenge is to fill those unknown cells in with predictions.

Well, if we're trying to come up with a matrix at the end of the day, maybe there are some machine learning techniques that work on matrices we can look at. Once such technique is called principal component analysis, or PCA.

principal component analysis

Sundog sundog-education.com

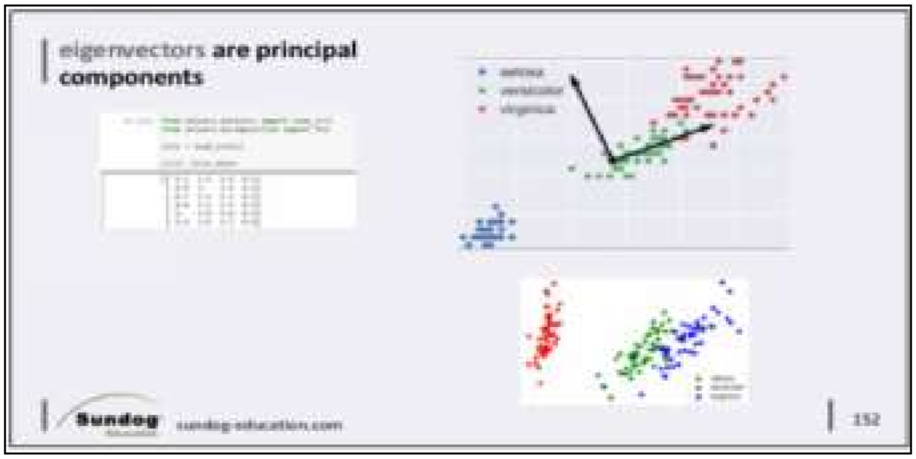
151

Principal Component Analysis

Principal component analysis is usually described as a "dimensionality reduction" problem. That is, we want to take data that exists in many dimensions, like all of the movies a user might rate, into a smaller set of dimensions that can accurately describe a movie, such as its genres.

So why is there a picture of a flower on the screen? Well, it's a little hard to imagine how this works on movies at first – but there's a data set on Iris flowers that makes for a good example of PCA in action.

Look closely at an iris, and you'll see it has a bunch of large petals on the outside, and some smaller petals on the inside. Those inner petals aren't called petals at all, they're called sepals. So one way to describe the shape of a specific Iris flower is by the length and width of its petals, and the length and width of its sepals.


eigenvectors are principal components

Sundog sundog-education.com

152

So if we look at the data we're working with in the Iris data set, we have length and width of petals and sepals for each Iris we've measured – so that's a total of four dimensions of data.

Our feeble human brains can't picture a plot of 4D data, so let's just think about the petal length and width at first. That's plotted here for all of the irises in our data set.

Without getting into the mathematical details of how it works, those black arrows are what's called the "eigenvectors" of this data. Basically, it's the vector that can best describe the variance in the data, and the vector orthogonal to it. Together, they can define a new vector space, or basis, that better fits the data.

These eigenvectors are our "principal components" of the data. That's why it's called principal component analysis – we are trying to find these principal components that describe our data, which are given by these eigenvectors .

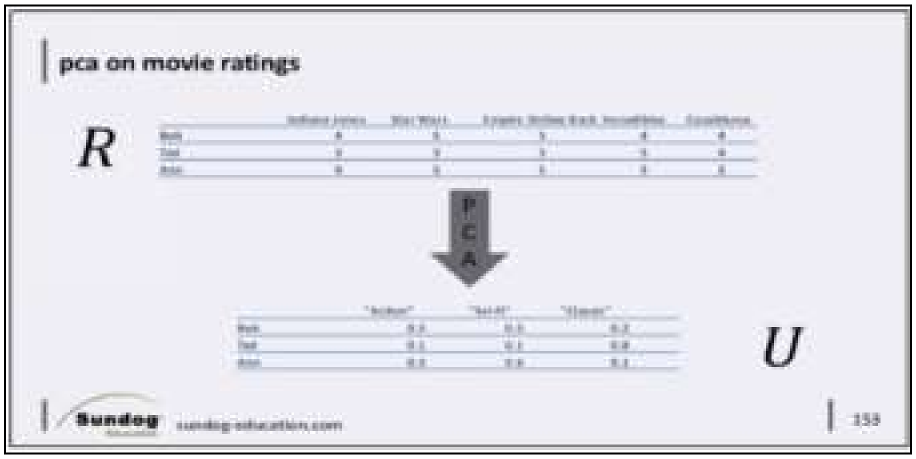
Let's think about why these principal components are useful. First of all, we can just look at the variance of the data from that eigenvector. That distance from the eigenvector is a single number; a single dimension that is pretty good at reconstructing the two-dimensional data we started with here for petal length and petal width. So you can see how identifying these principal components can let us represent data using fewer dimensions than we started with.

Also, these eigenvectors have a way of finding interesting features that are inherent in the data. In this case, we're basically discovering that the overall size of an iris's petals are what's important for classifying which species of iris it is, and the ratio of width to length is more or less constant. So the distance along this eigenvector is basically measuring the "bigness" of the flower. The math has no idea what "bigness" means, but it found the vector that defines some hidden feature in the data that seems to be important in describing it – and it happens to correspond to what we call "bigness."

So, we can also think of PCA as a feature extraction tool. We call the features it discovers latent features.

So here's what the final result of PCA looks like on our complete 4-dimensional Iris data set – we've used PCA to identify the two dimensions within that 4-D space that best represents the data, and plotted it in those two dimensions. PCA by itself will give you back as many dimensions as you started with, but you can choose to discard the dimensions that contain the least amount of information. So by discarding the two dimensions from PCA that told us the least about our data, we could go from 4 dimensions down to 2.

We can't really say what these two dimensions represent. What does the X axis in this plot mean? What does the Y axis mean? All we know for sure is that they represent some sort of latent factors, or features, that PCA extracted from the data. They mean something, but only by examining the data can we attempt to put some sort of human label on them.



pca on movie ratings

R

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5	5	4	4
Ted	3	3	3	5	4
Ann	4	5	5	3	3

	"Action"	"Sci-Fi"	"Classic"
Bob	0.3	0.5	0.2
Ted	0.1	0.1	0.8
Ann	0.3	0.6	0.1

U

Sundog sundog-education.com 153

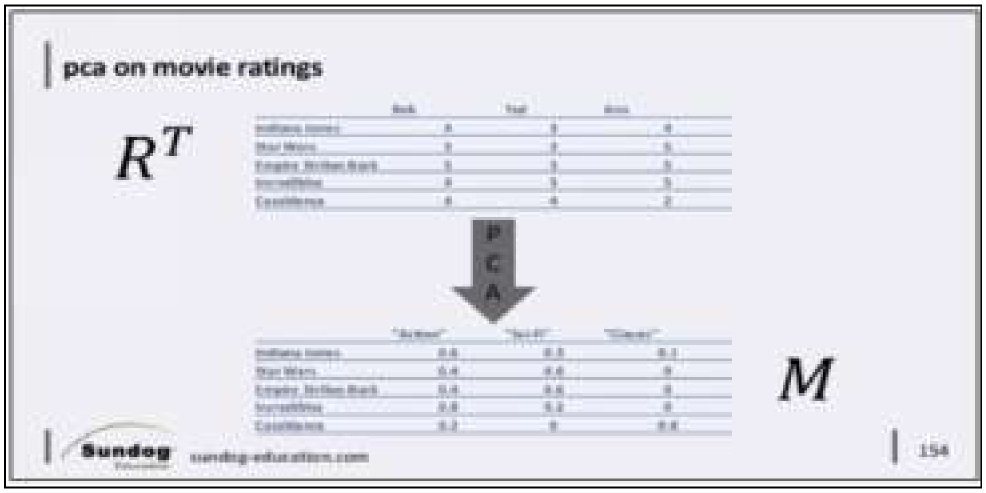
So just like we can run PCA on a 4-dimensional Iris data set, we can also run it on our multidimensional movie rating data set, where every dimension is a movie. We'll call this ratings matrix that has users as rows R.

Just like it did with our Iris data set, PCA can boil this down to a much smaller number of dimensions that best describe the variance in the data. And often, the dimensions it finds correspond to features humans have learned to associate with movies as well, for example, how "action-y" is a movie? How romantic is it? How funny is it? Whatever it is about movies that causes individuals to rate them differently, PCA will find those latent features and extract them from the data. PCA doesn't know what they mean, but it finds them nonetheless.

So, we could ask PCA to distill things down to say 3 dimensions in this example, and it would boil our ratings down to 3 latent features it identified. PCA won't know what to call them, but let's say they end up being measures of each person's interest in action, sci-fi, and classic genres. So for example, we might think of Bob as being defined as 30% action, 50% sci-fi, and 20% classic in terms of his interests.

Now, take a look at the columns in this new matrix. Each column is a description of users that make up that feature. "Action" can be described as 30% of Bob plus 10% of Ted plus 30% of Ann.

Let’s call this matrix U – its columns describe typical users for each latent feature we produced.



pca on movie ratings

R^T

	Bob	Ted	Ann
Indiana Jones	4	5	4
Star Wars	5	5	5
Empire Strikes Back	5	5	5
Incredibles	4	5	5
Casablanca	4	4	2

	"Action"	"Sci-Fi"	"Classic"
Indiana Jones	0.6	0.3	0.1
Star Wars	0.4	0.6	0
Empire Strikes Back	0.4	0.6	0
Incredibles	0.8	0.2	0
Casablanca	0.2	0	0.8

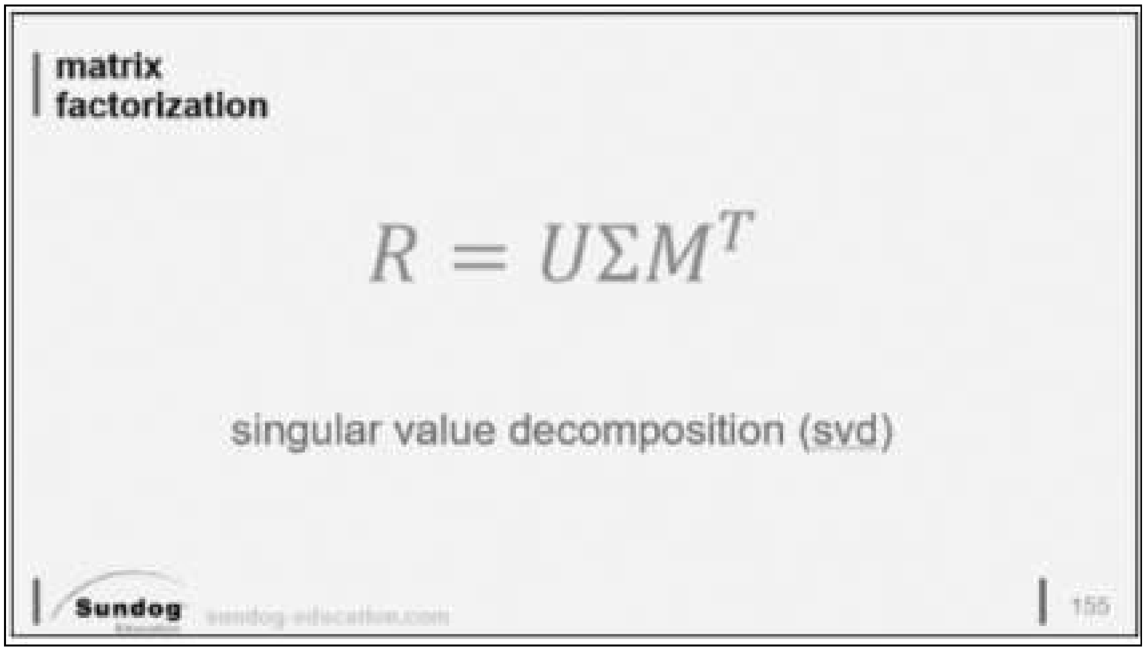
M

Sundog sundog-education.com 154

Just like we can run PCA on our user ratings data to find profiles of typical kinds of users, we can flip things around and run PCA to find profiles of typical kinds of movies.

If we re-arrange our input data so that movies are rows and users are columns, it would look like this. We call this the transpose of the original ratings matrix, or R-T for short .

Now if we run PCA on that, it will identify the latent features again, and describe each individual movie as some combination of them. Each column is now a description of some typical movie that exhibits some latent feature. Again, these have no inherent meaning, but in practice it might fall along movie genre lines – in reality though, it’s more complex.



matrix factorization

$$R = U\Sigma M^T$$

singular value decomposition (svd)

Sundog sundog-education.com 155

Let’s call this resulting matrix that describes typical movies M.

So how do these matrices that describe typical users and typical movies help us to predict ratings?

Well, it turns out that the typical movie matrix and the typical user matrix’s transpose are both factors of the original rating matrix we started with. So if we have M and we have U, we can reconstruct R. And if R is missing some ratings, if we have M and U we can fill in all of those blanks in R! That’s why we call this matrix factorization – we describe our training data in terms of smaller matrices that are factors of the ratings we want to predict.

There is also that sigma matrix in the middle that we haven’t talked about that we need – it’s just a simple diagonal matrix that only serves to scale the values we end up with into the proper scale. You could just multiply that scaling matrix into M or U, and still think of R as just the product of two matrices if that makes it easier to wrap your head around it.

So, you could reconstruct R all at once by multiplying these factors together, and get ratings for every combination of users and items. Once you have these factors though, you can also just predict a rating for a specific user and item by taking the dot product of the associated row in U for the user, and the associated column in M-T for the item. That’s just how matrix multiplication works.

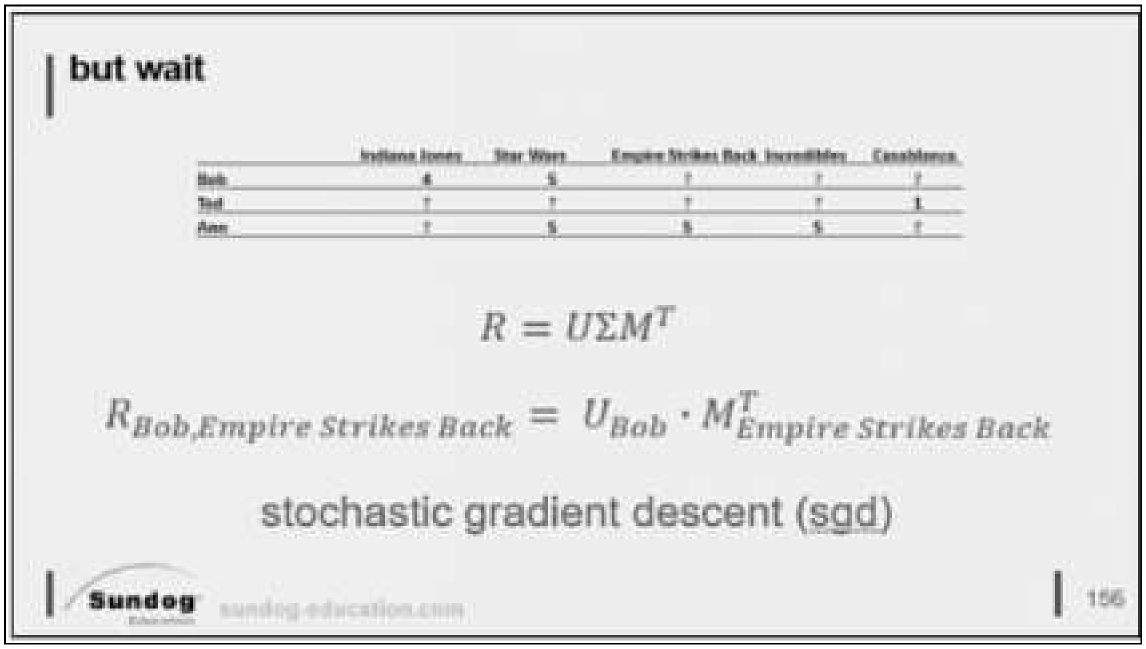
Finally, we’re going to tie it all together. A few times in this course we’ve used a built-in recommender called SVD, and it’s known to produce very accurate results – it was used widely during the Netflix Prize amongst the leaders in the competition. SVD stands for *singular value decomposition.

Know what singular value decomposition does? It’s a way of computing U, Sigma, and M-T together all at once very efficiently. So all SVD is doing is running PCA on both the users and the items, and giving us back the matrices we need that are factors of the ratings matrix we want. SVD is just way to get all three of those factors in one shot.

So, you finally know what SVD is and what it’s doing! Phew.

But wait – how do we compute U and M-T when our original matrix R is missing most of its values? You can’t run PCA on a matrix where most of it is missing – it must be a complete matrix.

You could just fill in the missing values with some sort of reasonable default values, like mean values of some sort – and this is what people did at first.



but wait

	Indiana Jones	Star Wars	Empire Strikes Back	Incredibles	Casablanca
Bob	4	5	?	?	?
Ted	?	?	?	?	1
Ann	?	5	5	5	?

$$R = U \Sigma M^T$$

$$R_{Bob,Empire\ Strikes\ Back} = U_{Bob} \cdot M^T_{Empire\ Strikes\ Back}$$

stochastic gradient descent (sgd)

Sundog sundog-education.com 156

But, there’s a better way. Remember every rating can be described as the *dot product of some row in the matrix U and some column in the matrix M-T. For example, if we want to predict Bob’s rating for The Empire Strikes Back, *we can compute that as the dot product of the Bob row in U and the Empire Strikes Back column in M-T.

So, let’s assume we have at least some known ratings for any given row and column in U and M-T. We can treat this as a minimization problem, where we try to find the values of those complete rows and columns that best minimize the errors in the known ratings in R.

There are lots of machine learning techniques that can do that sort of thing, such as *Stochastic Gradient Descent, or SGD for short. Basically it just keeps iterating at some given learning rate until it arrives at a minimum error value. We could talk about SGD in more depth, but I think I’ve made your head hurt enough for now – and we’ll return to SGD when we talk about neural networks, because it comes into play there as well. And again, SGD is just one way to do it – Apache Spark for example uses a different technique called alternating least squares, or ALS.

You might be confused here because all of a sudden we're talking about learning the values in the matrices M and U, and not computing them directly which is what SVD does. And, you're right to be confused – when we say we're doing SVD recommendations, it's not really SVD, because you can't do real SVD with missing data. It's an SVD-inspired algorithm that was invented for the Netflix Prize, but it's not really pure SVD.

The winner of the Netflix prize was a combination of a specific variant of SVD called SVD++, and another technique called Restricted Boltzmann Machines – but that's a story for another section!

I realize this is all pretty tough to wrap your head around, and you might want to watch this section again to help it sink in. But the important points are this: you can think of all ratings for a set of users and items as a matrix R, and that matrix R can be factored into smaller matrices that describe general categories of users and items that can be multiplied together. A quick way to get those matrices is a technique called SVD (singular value decomposition), and once you have those factored matrices, you can predict the rating of any item by any user by just taking a dot product from each matrix. Techniques such as SGD (stochastic gradient descent) and ALS (alternating least squares) can be used to learn the best values of those factored matrices when you have missing data.

That's definitely enough theory – let's see SVD in action and recommend some movies with it!

Coding Activity: SVD

code walkthrough

Sundog sundog-education.com 158

So open up Spyder, and close out any existing windows so we don't get confused (control-shift-w) .

Open up everything inside the MatrixFactorization folder inside your course materials.

Let's start by looking at the SVDBakeOff.py file. Surpriselib includes a couple of different SVD implementations; one is SVD more or less as we've described it, and the other is called SVD++. It's a slight variant of SVD that ended up being part of the winning system for the Netflix Prize. The difference is fairly subtle; it has to do with the actual loss function that is used while running stochastic gradient descent. In SVD++, this loss function takes into account the idea that merely rating an item at all is some form of implicit interest in the item, no matter what the rating was. But that was enough to make SVD good enough to win.

There's not much to talk about in the code itself; the actual implementation of SVD and SVD++ is handled by the library. You can look at SurpriseLib in GitHub if you want to see how the code works under the hood, but it's way too complex to walk through right now. When you start dealing with algorithms as complex as SVD, it's almost never a good idea to write it from scratch yourself. Use third party, open source libraries that have been used and validated by others instead – the odds are just too great that you'll mess up some tiny thing that ruins your results otherwise.

So let's go ahead and kick this off and see what sort of results we get. It will take several minutes to run, so we're going to just pause and come back when it's done. SVD++ in particular takes some time.

These accuracy results are the best we've seen so far, and as expected SVD++ is a little better – actually getting below 0.9 RMSE. That's pretty impressive, as accuracy goes. But what do the actual top-N recommendations look like? Let's scroll down and take a look at SVD first .

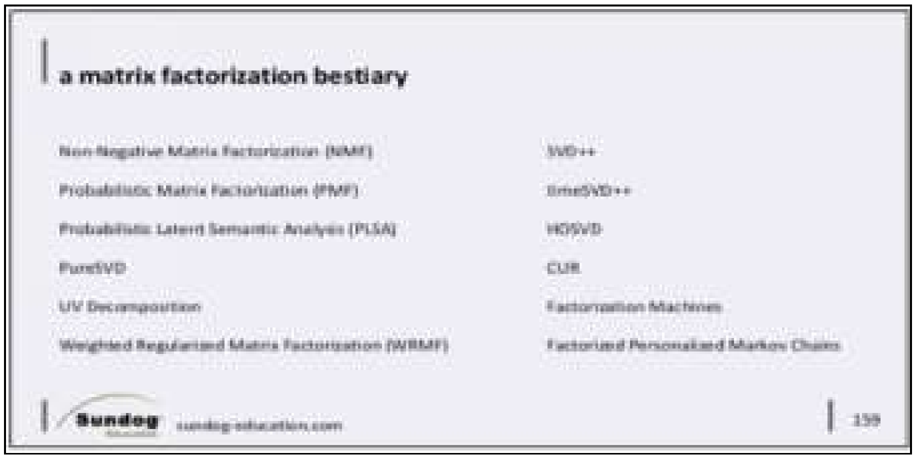
OK, these don't look too bad. Gladiator, Moon, Wallace and Gromit – those are all movies I enjoyed, and recall that I selected our test user as someone who has similar tastes to myself. Many of the other results are movies that are literally on my list of movies I want to watch, so on the surface it seems that the SVD recommender did a pretty good job of recommending a mix of movies I know about that give me trust in the system, and recommending somewhat more obscure titles that I'm genuinely likely to enjoy. This seems like something worth testing on real people for sure.

The difference in accuracy with SVD++ was subtle, so I wouldn't expect to see any big changes in its results, but let's see.

Hm, well we do see a few of the same movies that SVD came up with, but we see a few new ones too – like Harry Potter and Indiana Jones. If user 85 really hasn't seen those movies yet, I'm kind of amazed that they haven't come up before. So all things being equal, SVD++ seems like the one I'd experiment with first. I like these results; it seems like something good is happening here.

That’s the thing with algorithms that extract latent factors – they can be a little bit spooky in how well they distill users down to their fundamental interests, and how well they distill movies into what makes them unique. Matching the two up with SVD can be a powerful idea. That was a lot of theory to wrap your head around, but we got a pretty good payoff for it here !

Flavors of Matrix Factorization



a matrix factorization bestiary

Non-Negative Matrix Factorization (NMF)	SVD++
Probabilistic Matrix Factorization (PMF)	timeSVD++
Probabilistic Latent Semantic Analysis (PLSA)	HOSVD
PureSVD	CUR
UV Decomposition	Factorization Machines
Weighted Regularized Matrix Factorization (WRMF)	Factorized Personalized Markov Chains

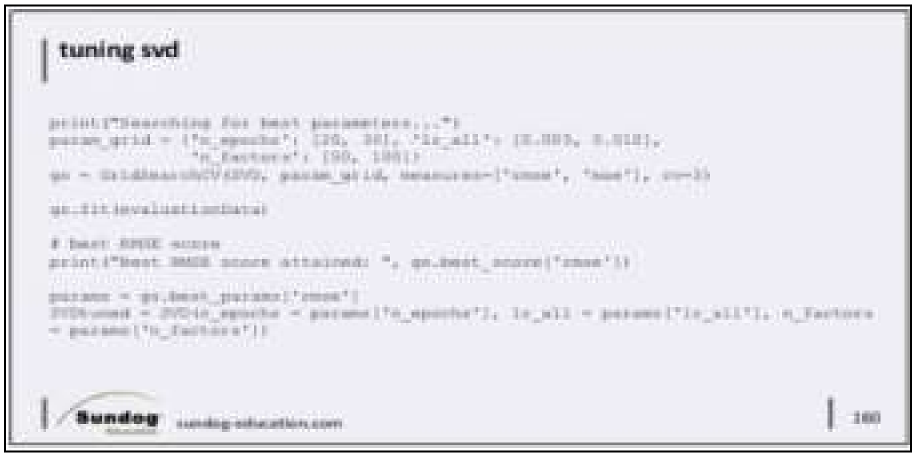
Sundog sundog-education.com 159

Given the success of SVD with the Netflix Prize, it’s not surprising that a lot of research since then has focused on building upon SVD for other specific tasks, or trying to improve it further. You can find recent papers on all of these variants out there. I’ll call out a few that are more interesting.

Factorization Machines are worth a look, as they are well suited to predicting ratings, or predicting clicks in recommender systems. It’s the same general idea as SVD, but a bit more general purpose I’d say – it can handle sparse data without trying to shoehorn itself into the problem, like SVD does. I’m calling attention to it because Amazon’s SageMaker service in AWS offers Factorization Machines as a built-in algorithm, so it’s something that’s easy to experiment with on very large data sets in the cloud. The only real downside is that it only works with categorical data, so you have to work a little bit harder at preparing your data to work with it. We’ll revisit factorization machines a couple of times later in the course.

There are also variants that are specifically for recommending series of events, like the next things you're likely to watch or click on given your immediate history. In neighborhood-based methods we talked about translation-based recs doing this, but here in the model-based section we have tools such as timeSVD++ and Factorized Personalized Markov Chains that can tackle the same sorts of problems.

Probabilistic Latent Semantic Analysis, or PLSA, is something a team I once ran experimented with once, and the early results were promising. You can use it to extract latent features from the content itself; for example, you could apply PLSA to movie titles or descriptions and match them up with users in much the same way PCA works. Content-based methods like this aren't likely to do very well on their own, but combining this with models built on user behavior data could be a good idea.



tuning svd

```
print("Searching for best parameters...")
param_grid = {'n_epochs': [20, 30], 'lr_all': [0.005, 0.010],
              'n_factors': [50, 100]}
gs = GridSearchCV(SVD, param_grid, measures=['rmse', 'mae'], cv=3)

gs.fit(evaluationData)

# best RMSE score
print("Best RMSE score attained: ", gs.best_score['rmse'])

params = gs.best_params['rmse']
SVDtuned = SVD(n_epochs = params['n_epochs'], lr_all = params['lr_all'], n_factors
= params['n_factors'])
```

Sundog sundog-education.com 160

As we've discussed before though, once you start getting into complex algorithms, often just tuning the parameters of the algorithm can produce markedly better results. And it turns out that SVD has several such parameters .

This gets into what we call hyperparameter tuning; it's a pretty big problem in machine learning in general. A lot of algorithms are very sensitive to parameters such as learning rates, and often different settings for these parameters make sense for different data sets.

For example, with SVD, we can adjust how many latent factors we try to extract – how many dimensions we want boil things down to. There's no right answer for that; it depends on the nature of the data you're dealing with. In the surpriselib implementation of SVD, this value is passed into the constructor of the SVD model as a parameter named "n_factors", and you can set it to whatever you want. Similarly, you can set your own learning rate for the SGD phase with "lr_all," and how many epochs, or steps, you want SGD to take with the "n_epochs" parameter.

Hyperparameter tuning is usually a matter of just trying different values and iterating until you find the best one. Generally it makes sense to start with the default setting, whatever that is, and then start guessing. Try doubling it – did that make it worse? Try halving it – did that make it better? Well then at least we know it should be lower. Is halving too much? Does ¾ work better? Basically you just keep narrowing down until you stop seeing significant gains in the quality of your results.

Fortunately, surpriselib contains a GridSearchCV package that helps you with hyperparameter tuning. It allows you to define a grid of different parameters you want to try out different values for, and it will automatically try out every possible combination of those parameters and tell you which ones performed the best.

Let's look at this code snippet. If you look at the param_grid dictionary we're setting up, you'll see it maps parameter names to lists of values we want to try. Remember we have to run the algorithm for every possible combination, so you don't want to try too many values at once. We then set up a GridSearchCV with the algorithm we want to test, the dictionary of parameters we want to try out, how we're going to measure success (in this case, by both RMSE and MAE,) and how many folds of cross-validation we want to run each time.

Then we run "fit" on the GridSearchCV with our training data, and it does the rest to find the combination of parameters that work best with that particular set of training data. When it's done, the best RMSE and MAE scores will be in the best_score member of the GridSearchCV, and the actual parameters that won will be in the best_params dictionary. You can look up the best_params for whatever accuracy metric you want to focus on; RMSE or MAE.

So, then we can create a new SVD model using the best parameters and do more interesting things with it.

Coding Exercise

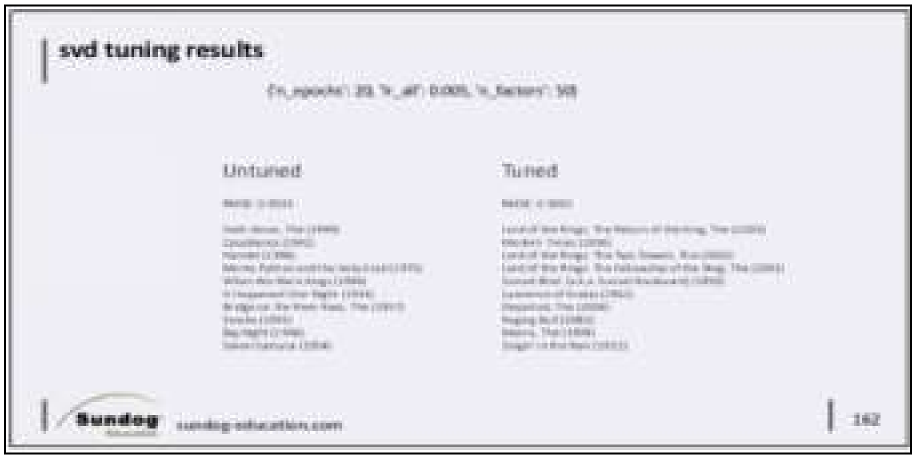
exercise

tune the hyperparameters for SVD with the MovieLens data set.

Sundog sundog-education.com 181

You can probably guess what your next exercise is… go ahead and try this out! Modify the SVDBakeOff script such that it searches for the best hyperparameters to use with SVD. Then, generate top-N recommendations with it and see how they look.

Oh, and if you noticed the SVDTuning.py file included in the course materials – yeah, that's the solution to this exercise. Resist the temptation to look at it unless you really get stuck. We'll review that file when we come back.



svd tuning results

{'n_epochs': 20, 'lr_all': 0.005, 'n_factors': 50}

Untuned	Tuned
RMSE: [illegible]	RMSE: [illegible]
[illegible]	Lord of the Rings: The Return of the King, The (2003)
Casablanca (1942)	Modern Times (1936)
Hamlet (1996)	Lord of the Rings: The Two Towers, The (2002)
Monty Python and the Holy Grail (1975)	Lord of the Rings: The Fellowship of the Ring, The (2001)
When We Were Kings (1996)	Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)
It Happened One Night (1934)	Lawrence of Arabia (1962)
Bridge on the River Kwai, The (1957)	Departed, The (2006)
[illegible]	Raging Bull (1980)
Big Night (1996)	Matrix, The (1999)
Seven Samurai (1954)	Singin' in the Rain (1952)

Sundog sundog-education.com 162

So we can briefly take a look at our SVDTuning.py file here to see how I went about doing hyperparameter tuning on SVD. You'll see it's mainly the same code we covered in the slides to set up the GridSearchCV object and use it to find the best parameters to set on SVD.

We then set up a little bake-off between SVD using the default parameters, and the tuned parameters we learned from GridSearchCV .

Here we see the results. I settled on 20 epochs, a learning rate of 0.005, and 50 factors. You may have gotten even better results depending on how persistent you were on iterating upon the ideal numbers instead of just estimating them, as I did.

But, we can see we did indeed get a better RMSE score with these tuned results. The difference is small, but it did result in very different top-N recommendation results, which is interesting. I would consider both of these results good, but they are completely different! For example, Lord of the Rings doesn't turn up at all when using the default parameters of SVD, but all three movies of the trilogy bubbled up when we tuned it. While the algorithm is probably correct in predicting that user number 85 would love all three of these movies, this is a good example of where results aren't diverse enough. Often, high diversity is a sign of random recommendations that aren't very good – but diversity that's too low can also be a bad thing. It would be better to only recommend the first movie in the series in a case like this, and free up those other two slots for different titles. Sometimes simple little rules like that – don’t recommend more than one movie in the same series – can make a difference.

Bleeding Edge Alert! Sparse Linear Methods

bleeding edge alert!

Sundog sundog-education.com

163

It's time for another bleeding edge alert! This is where we talk about some recent research that has promising results, but hasn't yet made it into the mainstream with recommender systems yet.

sparse linear methods (SLIM)

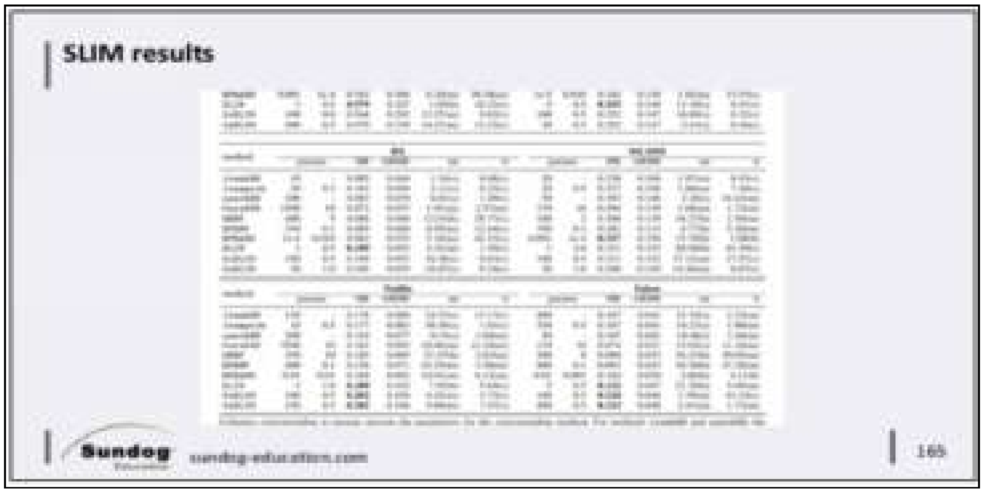
SLIM: Sparse Linear Methods for Top-N Recommender Systems

[illegible]

Sundog sundog-education.com

164

In this bleeding edge alert, we're actually going back to the year 2011 to look at sparse linear methods, or SLIM for short. This came out of the University of Minnesota, which has been a pioneer in the field recommender systems from the very beginning. The results from SLIM were very exciting, and I don't know why we don't hear about it being used in industry more. It's something definitely worth watching and keep an ear out for.



SLIM results

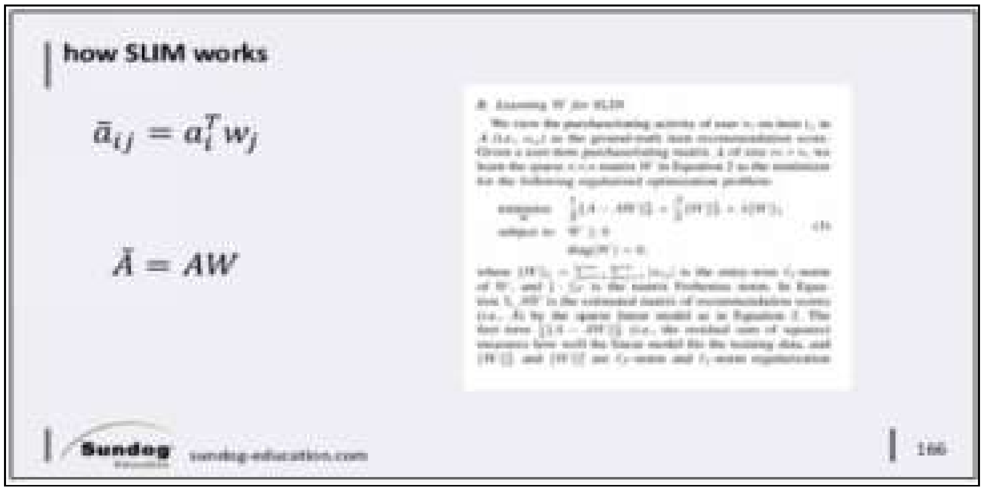
[illegible]

Sundog Education sundog-education.com

165

What caught my attention with this paper is how consistently SLIM outperformed a wide variety of competing algorithms, on a wide variety of data sets. Although, they didn't include SVD++ in their comparisons – but they did compare against many similar algorithms. Also, they measured their success on hit rate, so their focus is very much on top-N recommendations instead of rating prediction accuracy, which again is the right thing to focus on.

But not only did SLIM beat the pants off of everything else on the Netflix data set, it also wins with data set from a book store, from Yahoo Music, from credit card purchasing data, and a couple of mail-order retailer data sets. The only data set where it didn't come out on top was, ironically, MovieLens – but it placed a close second.



how SLIM works

$$\tilde{a}_{ij} = a_i^T w_j$$

$$\tilde{A} = AW$$

B. Learning W for SLIM

[illegible]

Sundog Education sundog-education.com

166

The idea behind SLIM is to generate recommendation scores for a given user and item by a sparse aggregation of the other things that user has rated, multiplied by a sparse aggregation of weights, which is where the magic lives. Here's what the notation in the paper looks like – tilde a sub i-j represents the unknown score for a given user for a given item, and that's equal to the entire row of stuff that user has rated, here indicated by a sub I T, multiplied by some precomputed weights associated with that row. Because the user has only rated some items and weights only exist for these items, that's where the word sparse comes from in this method.

So if you extend this idea to the entire user/item rating matrix, it looks like this:

Pretty simple, right? To predict ratings, we just multiply the rating we know about by some magical weight matrix called W. Both A and W are sparse matrices, meaning it contains incomplete data – just the ratings that actually exist.

So the secret sauce is in how W is computed, and this is where it gets complicated. So complicated that a lot of people have trouble implementing it, which might explain why it's not more widely used. This is just the first couple of paragraphs from the paper that describes how to do it. At its heart, it's an optimization problem just like we used stochastic gradient descent to learn the matrices in SVD. But you seriously need a higher degree in mathematics to understand what's going on here. If phrases like "matrix Frobenius norm" and "l1-norm regularization" make sense to you, then knock yourself out. But otherwise, I'll be snooping around on Github for people who have already implemented this successfully and shared their code for it. I did find one, but the guy who wrote it had trouble getting it to work at scale. It's not easy stuff at all.

Still, you can't argue with SLIM's results, so if you see it offered as part of a machine learning library you're working with, it's definitely worth your time to experiment with it. The research community has taken note, and some extensions on SLIM have emerged, such as contextual SLIM and higher-order SLIM, or HOSLIM. Even in the original paper, an extension called fsSLIM, or feature selection SLIM, is described, which restricts SLIM to use only columns most similar to what you want – this improves hit rate ever so slightly.

So if a SLIM implementation comes your way, give it a second look. It’s promising stuff.

Recommendations with Deep Learning

recommendations with deep learning

Sundog sundog-education.com

167

So we all know that artificial intelligence and “deep learning” with artificial neural networks are all the rage now, so it’s not surprising that there is a lot of research around applying deep learning to recommender systems.

Know something, though? Matrix factorization can be implemented with a neural network, so we’ve already achieved a lot of the same results you can get through deep learning – it’s just another way to implement the same thing. SVD seems spooky in how good it is, in the same way that deep learning often seems spooky. And there’s a very good reason for that – both techniques can have the same results.

But, deep learning also opens up entirely new approaches to making recommendations that are worth exploring. A lot of this is still bleeding-edge stuff, but it’s where things are headed for the time being – so it’s important to understand .

Introduction to Deep Learning

intro to deep learning

Sundog Education sundog-education.com

168

But before we can understand how to apply deep learning and artificial neural networks to recommender systems, first we need to understand how deep learning and artificial neural networks work. If you're already familiar with deep learning and Tensorflow in particular, you can probably get away with skipping this next section. But if Tensorflow and neural networks are new to you, or you're just rusty, I encourage you to go through the next section so you understand how it works before we try to apply these tools to recommender systems.

This section is adapted from a portion of a larger, free course that I offer on YouTube on deep learning and neural networks, so if you want to dive deeper just search for Sundog Education on YouTube and you'll find it. We're not going to talk about recommender systems in this section; we're going to cover the concepts of deep neural networks in more general terms. Then, in subsequent sections, we'll apply what we've learned to the problem of recommender systems .

Deep Learning Pre-requisites

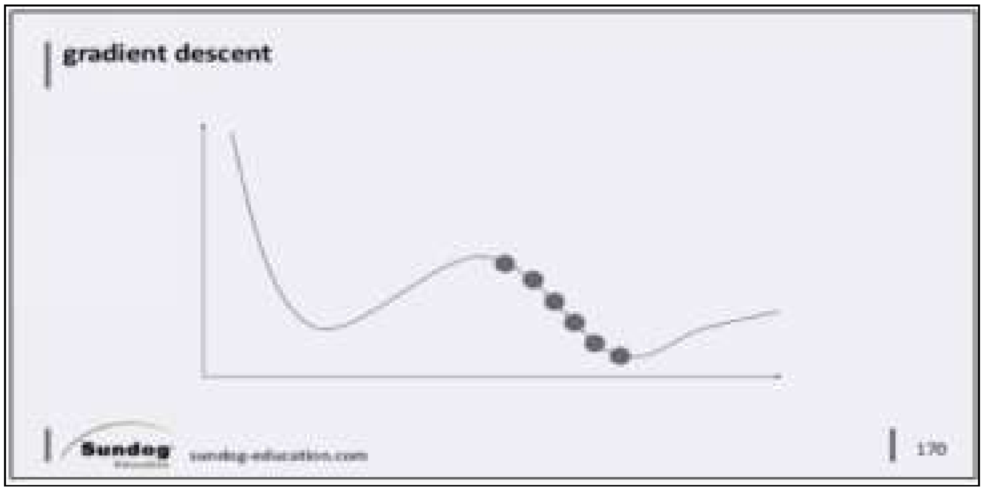
deep learning pre-requisites

Sundog sundog-education.com 169

Before we dive into deep learning, there are a few algorithms you need to understand first, as they are important components of how deep learning works.

One of these pre-requisites is stochastic gradient descent, or SGD – that's the same SGD we talked about when we covered matrix factorization. As I mentioned, it's possible to implement matrix factorization using neural networks, so it shouldn't come as any surprise that you can train both approaches using the same technique.

We're also going to talk about autodiff, which is a mathematical trick that SGD depends on with neural networks. And we'll cover softmax, which is used at the output of neural networks to translate their raw outputs into more useful rating classifications.



gradient descent

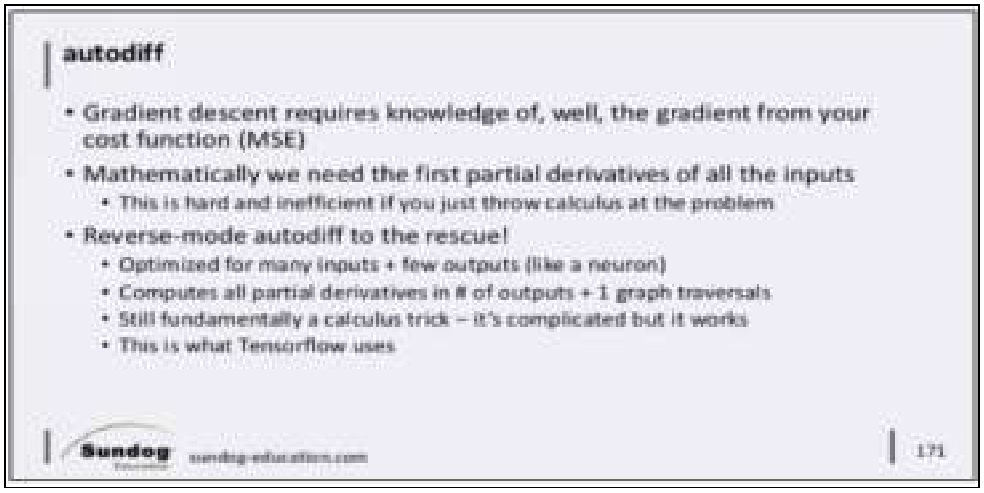
Sundog sundog-education.com 170

The first thing we want to talk about is Gradient Descent. This is basically a Machine Learning optimization technique for trying to find the most optimal set of parameters for a given problem. So what we're plotting here basically is some sort of cost function, some measurement of the error of your learning system - this applies to machine learning in general. You're going to have some sort of function that defines how close your predicted values are to the actual observed values. So in the context of supervised learning, we will be feeding our algorithm, or model, a group of parameters. These parameters define ways that we have tuned the model, and we need to identify different values of those parameters that produce the optimal results.

So the idea with gradient descent is that you just pick some point at random, and each one of these dots represents some set of parameters to your model. Maybe it's the various parameters for some model we've talked about before, or maybe it's the exact weights within your neural network. Whatever it is, we're going to try some set of parameters to start with, and we will then measure whatever the error that produces on our system. Then, we move on down the curve here. So we'll try a different set of parameters here, again, just like moving in a given direction with different parameter values, and we then measure the error that we get from that, and in this case we actually achieved less error by trying this new set of parameters. So we say, "OK, I think we're heading in the right direction here. Let's change them even more in the same way," and we just keep on doing this at different steps until finally we hit the bottom of a curve here and our error starts to increase after that point. So at that point we'll know that we actually hit the bottom of this gradient – and that's the nature of the term, "gradient descent." Basically we're picking some point at random with a given set of parameters that we measure the error for, and we keep on pushing those parameters in a given direction until the error minimizes itself and starts to come back up some other value. That's how gradient descent works in a nutshell. I'm not going to get into all the hard core mathematics of it all; the concept is what's important here because gradient descent is how we actually train our neural networks to find an optimal solution.

Now you can see there are some areas of improvement for this idea. First of all, you can actually think of this as sort of a ball rolling downhill, so one optimization that we'll talk about later is using the concept of momentum. You can actually have that ball gain speed as it goes down the hill here, and slow down as it reaches the bottom. That's the way to make it converge more quickly when you're doing things and can make the actual training of your neural networks even faster. Another thing we're talking about is the concept of local minima. So what if I randomly picked a point that ended up over in the leftmost dip in this curve? I might end up settling into this minima here which isn't actually the point of the least error. The point with the least error in this graph is over to the right. That's a general problem with gradient descent. How do you make sure that you don't get stuck in what's called a local minima? Because if you just look at this part of the graph, that looks like the optimal solution, and if I just happen to start over here that's where I'm going to get stuck .

There are various ways of dealing with this problem. Obviously you could start from different locations, try to prevent that sort of thing, but in practical terms it turns out that local minima aren't really that big of a deal when it comes to training neural networks. This just doesn't really happen that often; you don't end up with shapes like this in practice, so we can get away with not worrying about it too much. That's a very good thing, because for a long time people believed that AI would be limited by this local minima effect, but in practice it's really not that big of a deal.


autodiff

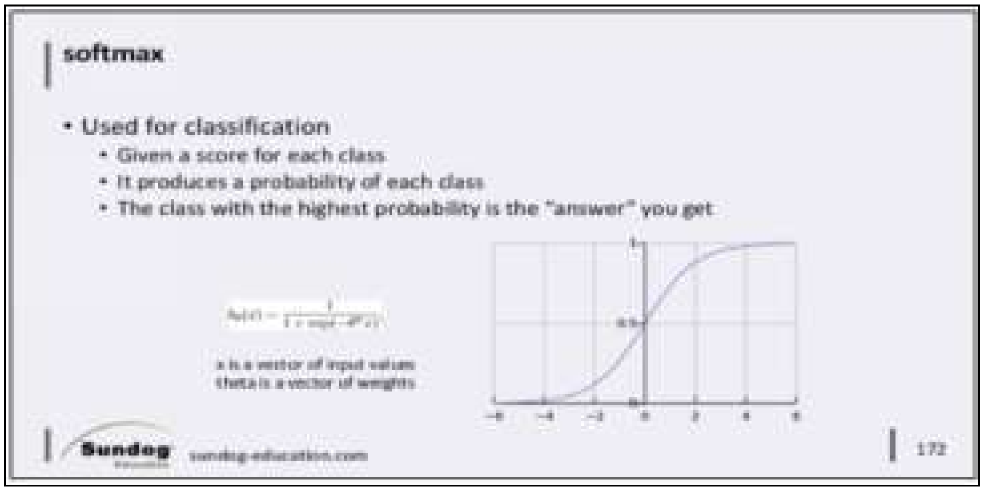
- Gradient descent requires knowledge of, well, the gradient from your cost function (MSE)
- Mathematically we need the first partial derivatives of all the inputs
  - This is hard and inefficient if you just throw calculus at the problem
- Reverse-mode autodiff to the rescue!
  - Optimized for many inputs + few outputs (like a neuron)
  - Computes all partial derivatives in # of outputs + 1 graph traversals
  - Still fundamentally a calculus trick – it's complicated but it works
  - This is what Tensorflow uses

Sundog sundog-education.com 171

Another concept we need to familiarize yourself with something called "autodiff." We don't really need to go into the hard core mathematics of how autodiff works, you just need to know what it is and why it's important. So when you're doing gradient descent, somehow you need to know what the gradient is, right? So we need to measure what is the slope that we're taking along our cost function, and to do that mathematically you need to get into calculus. If you're trying to find the slope of a curve and you're dealing with multiple parameters, then we're talking about computing partial derivatives, which sounds pretty intense. The first partial derivatives figure out the slope that we're heading in. It turns out that this is very mathematically intensive and inefficient for computers to do, so taking a brute force approach to gradient descent turns out to be very hard to do.

Autodiff is a technique for speeding that up. Specifically we use something called reverse-mode autodiff. What you need to know is that it can compute all the partial derivatives you need just by traversing your graph as many times as you have outputs plus one. This works out really well in neural networks because in a neural network you tend to have artificial neurons that have very many inputs, but probably only one output, or very few outputs in comparison to the inputs. So this turns out to be a pretty good little calculus trick. It's complicated – look up the details if you really want to get into the hardcore math behind it - but it works. Autodiff is what the Tensorflow library uses under the hood to implement its gradient descent. You’re never going to have to actually implement gradient descent from scratch or implement autodiff from scratch; these are all baked into the libraries that we're using such as TensorFlow. But they are terms that we throw around a lot, so it's important that you at least know what they are and why they're important.

So to review, gradient descent is the technique we're using to find the local minima of the error that we're trying to optimize for, given a certain set of parameters. Autodiff is a way of accelerating that process, so we don't have to do quite as much math or quite as much computation to actually measure that gradient of the gradient descent.



softmax

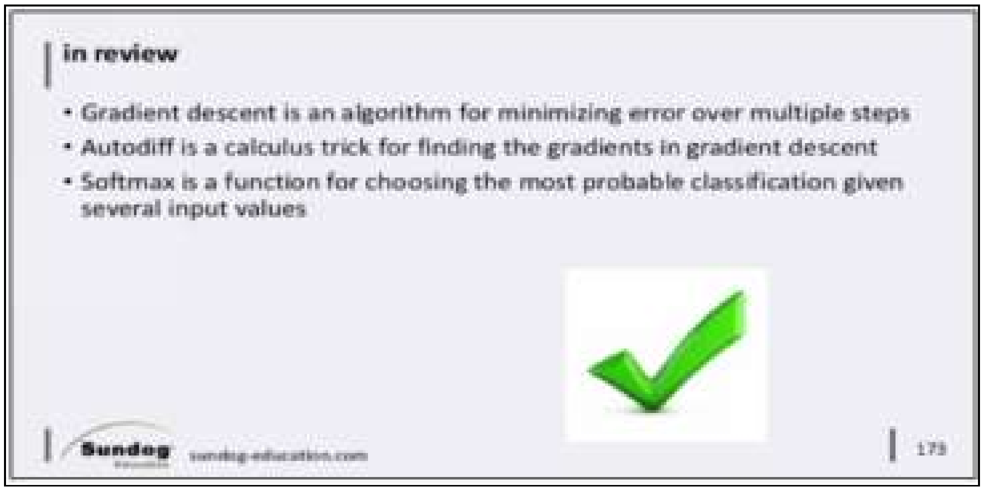
- Used for classification
  - Given a score for each class
  - It produces a probability of each class
  - The class with the highest probability is the "answer" you get

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)}$$

x is a vector of input values
theta is a vector of weights

One other thing we need to talk about is softmax. The mathematics aren't so complicated here, but again, what's really important is understanding what it is and what it's for. When you have the end result of a neural network you end up with a bunch of what we call weights that come out of the neural network at the end. So how we make use of those final weights? How do we make practical use of the output of our neural networks? Well, that's where softmax comes in. Basically it converts each of the final weights that come out of your neural network into a probability. So if you're trying to classify something in your neural network like, for example, deciding if an image is a picture of a face or a picture of a dog or a picture of a stop sign, you might use softmax at the end to convert those final outputs of the neurons into probabilities for each class. Then you can just pick the class that has the highest probability. So it's just a way of normalizing things into a comparable range and in such a manner that if you actually choose the highest value of the softmax function from the various outputs, you end up with the best choice of classification at the end of the day. It's just a way of converting the final output of your neural network to an actual answer for a classification problem .

So again, you might have the example of a neural network that's trying to drive your car for you, and it needs to identify pictures of stop signs or yield signs or traffic take your image and classify it as one of those sign types.



in review

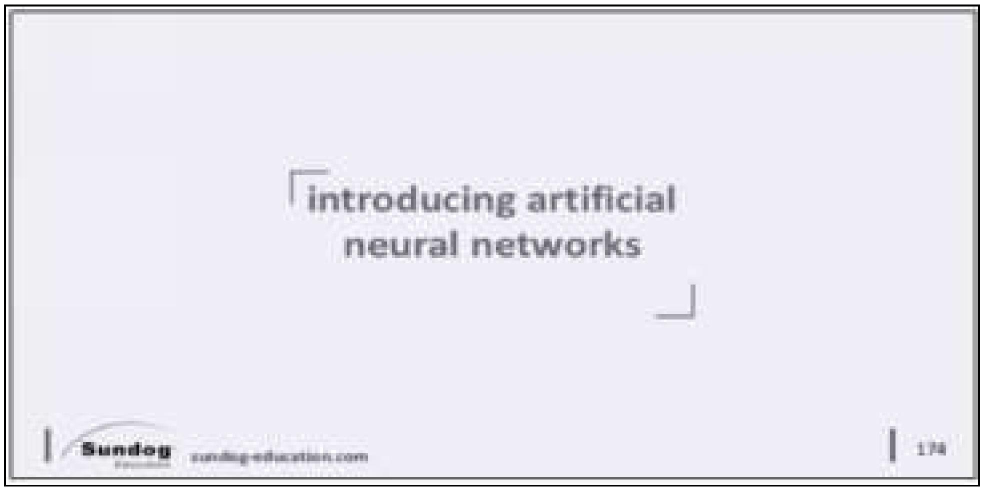
- Gradient descent is an algorithm for minimizing error over multiple steps
- Autodiff is a calculus trick for finding the gradients in gradient descent
- Softmax is a function for choosing the most probable classification given several input values

Sundog sundog-education.com 173

So again, just to recap: Gradient descent is an algorithm for minimizing error over multiple steps. Basically, we start with some random set of parameters, measure the error, move those parameters in a given direction, see if that results in more error or less error and just try to move in the direction of minimizing error until we find the actual bottom of the curve. That point is where we have a set of parameters that minimizes the error of whatever it is you're trying to do. Autodiff is just a calculus trick for making gradient descent faster; it makes it easier to find the gradients in gradient descent just by using some calculus trickery. Softmax is just something we apply on top of our neural network at the very end to convert the final output of our neural network to an actual choice of classification given several classification types to choose from .

So those are the basic mathematical terms or algorithmic terms that you need to understand to talk about artificial neural networks. With that under our belt, let's talk about artificial neural networks next.

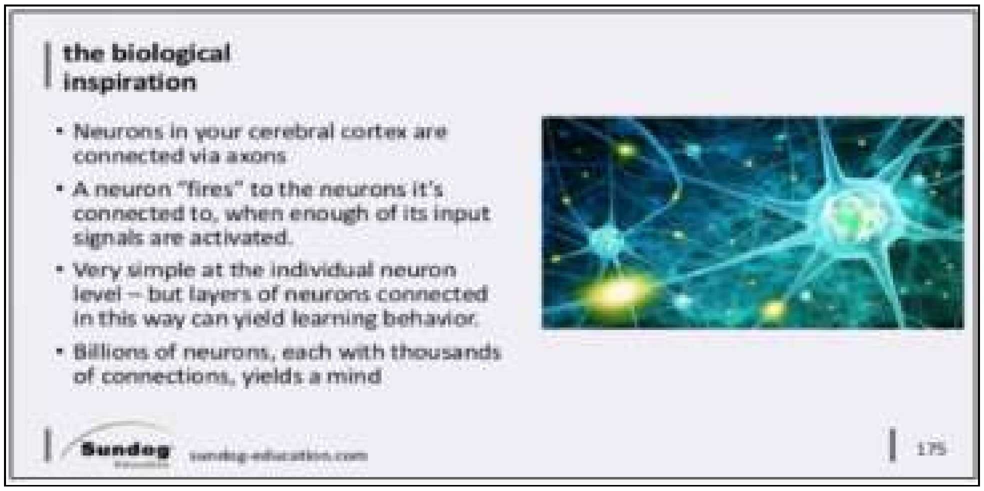
Artificial Neural Networks


introducing artificial neural networks

Sundog Education sundog-education.com

174

Let's dive into artificial neural networks and how they work at a high level. Later on we'll actually get our hands dirty and actually create some, but first we need to understand how they work and where they came from.


the biological inspiration

- Neurons in your cerebral cortex are connected via axons
- A neuron "fires" to the neurons it's connected to, when enough of its input signals are activated.
- Very simple at the individual neuron level – but layers of neurons connected in this way can yield learning behavior.
- Billions of neurons, each with thousands of connections, yields a mind

Sundog Education sundog-education.com

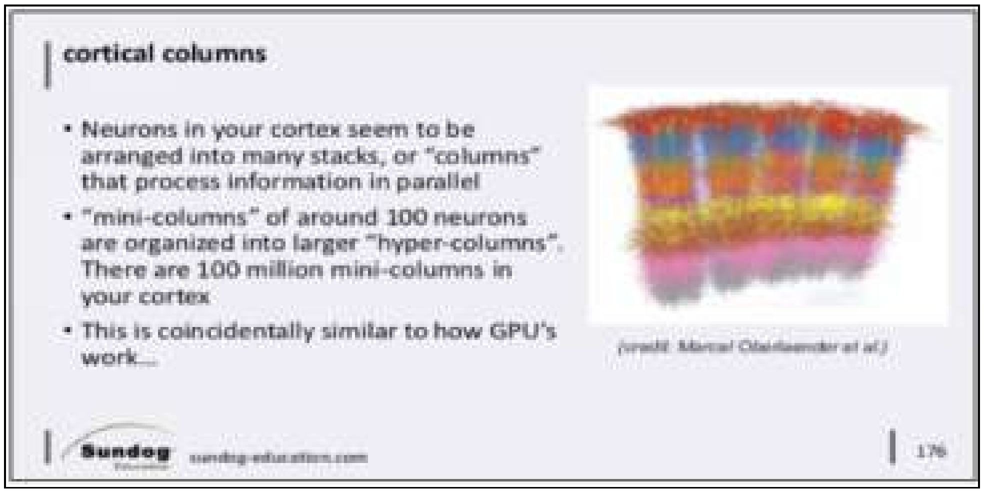
175

It's pretty amazing stuff. This whole field of artificial intelligence is based on an understanding of how our own brains work. Over millions of years of evolution, nature has come up with a way to make us think - and if we just reverse engineer the way that our brains work, we can gain some insights on how to make machines that think.

Within your brain, specifically within your cerebral cortex which is where all of your thinking happens, you have a bunch of neurons. These are individual nerve cells, and they are connected to each other via axons and dendrites. You can think of these as connections -- wires, if you will, that connect different axons together. Now an individual neuron will fire or send a signal to all the neurons that it's connected to when enough of its input signals are activated. At the individual neuron level it's a very simple mechanism: you just have this neuron that has a bunch of input signals coming into it, and if enough of those inputs signals reach a certain threshold, it will in turn fire off a set of signals to the neurons that it in turn is connected to as well .

But when you start to have many, many, many of these neurons connected together in many, many different ways with different strengths between each connection, things get very complicated. This is a perfect example of "emergent behavior;” you have a very simple concept, a very simple model, but when you stack enough of them together, you can create very complex behavior that can yield learning behavior. This actually works. Not only does it work in your brain, it works in our computers as well.

Now think about the scale of your brain. You have billions of neurons, each of them with thousands of connections. That's what it takes to actually create a human mind. And this is a scale that we can still only dream about in the field of Deep Learning and Artificial Intelligence, but it's the same basic concept. You just have a bunch of neurons with a bunch of connections that individually behave very simply, but once you get enough of them together wired in enough complex ways, you can actually create very complex thoughts -- and even consciousness. The plasticity of your brain is basically tuning where those connections go to and how strong each one is, and that's where all the magic happens.



cortical columns

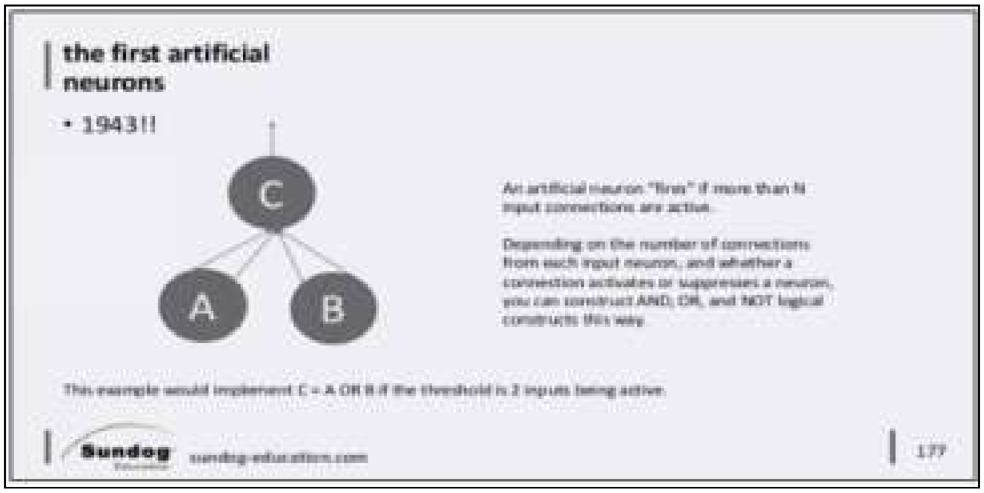
- Neurons in your cortex seem to be arranged into many stacks, or "columns" that process information in parallel
- "mini-columns" of around 100 neurons are organized into larger "hyper-columns". There are 100 million mini-columns in your cortex
- This is coincidentally similar to how GPU's work…

(credit: Marcel Oberlaender et al.)

Sundog sundog-education.com 176

Furthermore, if we look deeper into the biology of your brain, you can see that within your cortex neurons seem to be arranged into stacks or cortical columns that process information in parallel. So, for example, in your visual cortex different areas of what you see might be getting processed in parallel by different columns, or cortical columns, of neurons. Each one of these columns is in turn made of mini-columns of around 100 neurons per mini-column. Mini-columns are then organized into larger hyper-columns, and within your cortex there are about 100 million of these mini columns. So again, they just add up quickly.

Coincidentally, this is a similar architecture to how the 3D video card in your computer works. It has a bunch of very simple, very small processing units that are responsible for computing how little groups of pixels on your screen are computed. It just so happens that that's a very useful architecture for mimicking how your brain works. So it's sort of a happy accident that the research behind your favorite video games lent itself to the same technology that made Artificial Intelligence possible on a grand scale and at low cost. The same video cards you're using to play your video games can also be used to perform Deep Learning and create artificial neural networks. Think about how much better it would be if we actually made chips that were purpose-built specifically for simulating artificial neural networks. Well, turns out some people are designing chips like that right now, by the time you watch this they might even be a reality. I think Google's working on one as we speak.



the first artificial neurons

- 1943!!

An artificial neuron "fires" if more than N input connections are active.

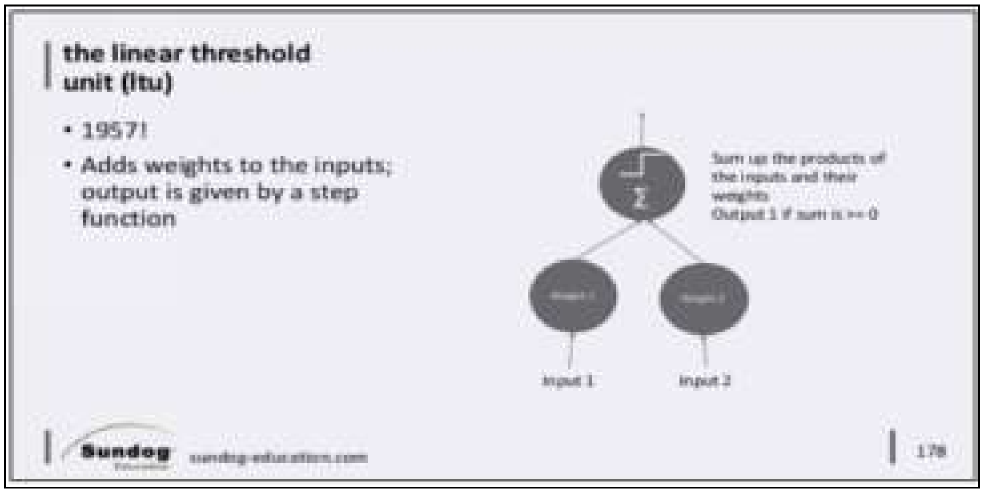
Depending on the number of connections from each input neuron, and whether a connection activates or suppresses a neuron, you can construct AND, OR, and NOT logical constructs this way.

This example would implement C = A OR B if the threshold is 2 inputs being active.

Sundog sundog-education.com 177

So at one point, someone said "hey! The way we think neurons work is pretty simple, it actually wouldn't be too hard to actually replicate that ourselves and maybe try to build our own brain." This idea goes all the way back to 1943. People proposed a very simple architecture where if you have an artificial neuron, maybe you can set up an architecture where that neuron fires only if more than a certain number of its input connections are active. When they thought about this more deeply in a computer science context, people realized you can actually create logical expressions, or boolean expressions by doing this. Depending on the number of connections coming from each input neuron, and whether each connection activates or suppresses a neuron, you can actually implement logical expressions in artificial or natural neurons.

This particular diagram is implementing an OR operation, so imagine that our threshold for our neuron was that if you have two or more inputs active, you will in turn fire off a signal. In this setup here, we have two connections to neuron A and two connections coming in from neuron B. If either of those neurons produce an input signal, that will actually cause neuron C to fire, so you can see we have created an OR relationship here where if either a neuron A or neuron B feeds neuron C two input signals, that will cause neuron C to fire and produce a “true” output. We've implemented here the boolean operation C = A OR B just using the same wiring that happens within your own brain. It’s also possible to implement AND and NOT via similar means.



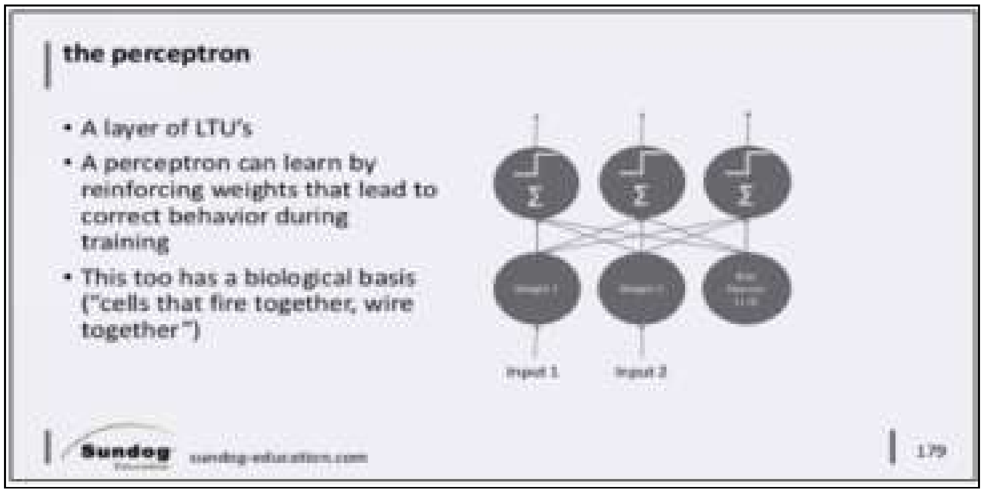
the linear threshold unit (ltu)

- 1957!
- Adds weights to the inputs; output is given by a step function

Sundog sundog-education.com

178

Then we start to build upon this idea. We created something called the Linear Threshold Unit, or LTU for short, in 1957. This just built on things by assigning weights to those inputs, so instead of just simple ON and OFF switches, we now have the concept of having weights on each of those inputs as well. This is working more toward our understanding of the biology; different connections between different neurons may have different strengths and we can model those strengths in terms of these weights on each input coming into our artificial neuron. We're also going to have the output be given by a step function. So this is similar in spirit to how we were using it before, but instead of saying we're going to fire if a certain number of inputs are active, well, there's no concept anymore of active or not-active, there's weights coming in and those weights could be positive or negative. If the sum of those weights is greater than zero, we'll go ahead and fire off a signal. If it's less than zero, we won't do anything. It’s just a slight adaptation to the concept of an artificial neuron, where we're introducing weights instead of just simple binary ON and OFF switches.



the perceptron

- A layer of LTU's
- A perceptron can learn by reinforcing weights that lead to correct behavior during training
- This too has a biological basis ("cells that fire together, wire together")

Σ
Σ
Σ
Input 1
Input 2

Sundog sundog-education.com 179

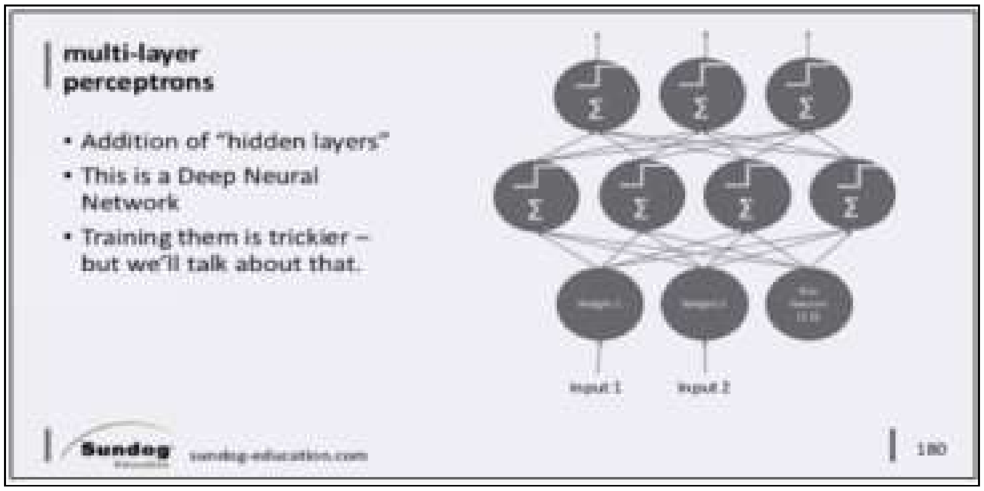
Let's build upon that even further, and create something called the perceptron. A perceptron is just a layer of multiple linear threshold units. Now we're starting to get into things that can actually learn!

By reinforcing weights between these LTU's that produced the behavior we want, we can create a system that learns over time how to produce the desired output. This also is working more toward our growing understanding of how the brain works. Within the field of neuroscience there's a saying that "cells that fire together wire together," and that's speaking to the learning mechanism going on in our artificial perceptron, where we have weights that are leading to the desired result that we want. We can think of those weights as strengths of connections between neurons, we can reinforce those weights over time and reward the connections that produce the behavior that we want.

You see here we have our inputs coming into weights just like we did in LTU's before, but now we have multiple LTU's grouped together in a layer. Each one of those inputs gets wired to each individual neuron in that layer. We then apply a step function to each one, which will produce a final set of outputs that can be used to classify something, like what kind of image this perceptron is looking at.

Another thing we introduce here is something called the Bias Neuron off there on the right. It's something to make the mathematics work out; sometimes we need to add in a little fixed constant value to make the neurons fire at the right values, and this bias amount can also be learned as the perceptron is trained.

So, this is a perceptron. We've taken our artificial neural network, moved that to a linear threshold unit, and now we've put multiple linear threshold units together in a layer to create a perceptron. We now have a system that can actually learn as you optimize all of the weights between the neurons in each layer, and you can see there's a lot of those weights at this point which can capture fairly complex information.

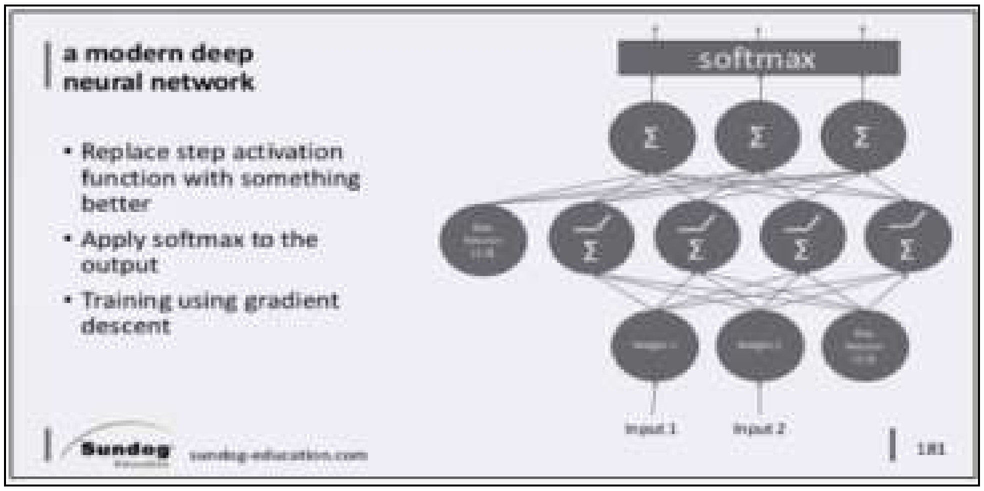


multi-layer perceptrons

- Addition of "hidden layers"
- This is a Deep Neural Network
- Training them is trickier – but we'll talk about that.

If you have every one of those inputs going to every single LTU in your layer, they add up fast, and that's where the complexity of deep learning comes from. Let's take that one step further, and we'll have a multi-layer perceptron. So now instead of a single layer perceptron of LTU's, we're going to have more than one. We actually have now a hidden layer in the middle there. You can see that our inputs are going into a layer at the bottom, the outputs are layered at the top, and in-between we have this hidden layer of additional LTU's, linear threshold units, that can perform what we call Deep Learning. So here we have already what we would call today a Deep Neural Network. Now there are challenges of training these things because they are more complex, but we'll talk about that later on. The thing to really appreciate here is just how many connections there are, so even though we only have a handful of artificial neurons, you can see there's a lot of connections between them and there's a lot of opportunity for optimizing the weights between each connection.

So that's how a multi-layer perceptron works. You can see that again we have emergent behavior here. An individual linear threshold unit is a pretty simple concept, but when you put them together in multiple layers all wired together, you can get very complex behavior because there's a lot of different possibilities for all the weights between all those different connections.


a modern deep neural network

- Replace step activation function with something better
- Apply softmax to the output
- Training using gradient descent

Finally, we'll talk about a modern Deep Neural Network. Really this is all there is to it - the rest of this section will just be talking about ways of implementing something like this. So all we've done here is replace that step function with something better. We'll talk about alternative activation functions; this one is illustrating something called ReLU, which we’ll examine more deeply very soon. The key point is that a step function has a lot of nasty mathematical properties, especially when you're trying to figure out their slopes and their derivatives, so turns out that other functions work out better and allow you to converge more quickly when you're trying to train a neural network.

We'll also apply softmax to the output, which we talked about in the previous lecture. That's just a way of converting the final outputs of our neural network or deep neural network into probabilities from which we can just choose the classification with the highest probability. And we will also train this neural network using gradient descent or some variation thereof. Maybe that will use autodiff, which we also talked about earlier, to actually make that training more efficient. So that's pretty much it!

In the past five minutes or so that we've been talking I've given you the entire history of deep neural networks and Deep Learning. It's not that complicated! That's really the beauty of it. It's emergent behavior, you have very simple building blocks, but when you put these building blocks together in interesting ways, very complex and sometimes mysterious things can happen. Let's dive into more details on how it actually works up next.

let's play

Sundog sundog-education.com 182

So now that we understand the concepts of artificial neural networks and Deep Learning, let's mess around with it! It's surprisingly easy to do.

The folks behind TensorFlow at Google have created a nice little web site called playground.tensorflow.org that lets us experiment with creating our own neural networks, and you don't have to write a line of code to do it. It's a great way to gain a hands-on, intuitive feel of how they work, so let's dive in.

Head over to playground.tensorflow.org and let it load up. I definitely encourage you to play around with it yourself and get sort of intuitive hands-on feel of how Deep Learning works; this is a very powerful thing if you can understand what's going on in this web page. What we're trying to do here is classify a bunch of points just based on their location in this 2D image. So this is our training dataset, if you will. We have a bunch of points here, and the ones in the middle are classified as blue and the ones on the outside are classified as orange, so our objective is to create a neural network that given no prior knowledge can actually figure out if a given point should be blue or orange, and predicts successfully which classification it should be.

So think of this is our training data. We know ahead of time what the correct classifications are for each one of these points, and we're going to use this information to train our neural network to learn that stuff in the middle of the image should be blue, and stuff on the outside should be orange. Now, here we have a diagram of the neural network itself and we can play around with it. We can manipulate it, we can add layers to it, take layers out, add more neurons to layer -whatever you want to do. Let's review what's going on here.

So first of all we're selecting the dataset that we want to play with. We're starting with this default one that's called “circle”. The inputs are simply the X and Y coordinates (the vertical and horizontal position of each data point). As our neural network is given a point to classify, all it has to work with are those two values: its horizontal position and its vertical position. Those start off as equally weighted being horizontal or vertical, so we can define the position of any one of these points in terms of its horizontal and vertical position. For example, this point would have a horizontal position of negative one and a vertical position of about negative 5. And then we feed it into our network. You can see that these input nodes have connections to each one of these four neurons in our hidden layer, and we can manipulate the weights between each one of these connections to create the learning that we want. Those in turn feed into two output neurons here that will ultimately decide which classification we want at the end of the day. So keep in mind this is a binary classification problem; it's either blue or orange, so ultimately we just need a single signal, and that's what comes into this output here. Let's go ahead hit play and see what happens .

It starts a bunch of iterations where it learns from this training data, so we're going to keep feeding it input from this training dataset. As it iterates through it, it will start to reinforce the connections that lead to the correct classifications through gradient descent or some similar mechanism. And if we do that enough times, it should converge to a neural network that is capable of reliably classifying these things. So let's just watch it in action. Keep your eye on that image to the right there. Over time you'll see that we've converged on a solution, and we can go ahead and pause that now.

You can see it has successfully created this pattern where stuff that fits into this middle area here is classified as blue, and stuff on the outside is classified as orange. We can dive into what actually happened here. The thickness of all these connections represent their weights, so you can see the individual weights that are wired between each one of these neurons. At the beginning, you can see these are more or less equally weighted with a few exceptions, but what it leads to is this behavior in the middle. So we start off with equally weighted X and Y coordinates, those go to the hidden layer, and one neuron is saying "I want to weight things a little bit more heavily in this corner," for example. And things that are in the lower left hand corner are weighted less, and then another one is picking out stuff on the top and bottom. In another neuron, it's a little bit more diagonal to the bottom-right and another is even more bottom-right heavy. If you combine these things together, you end up with these output layers that lead to a final classification.

We end up with these two blobby things where we're giving a boost to things on the right and giving a boost to things that lie within sort of this more blobby circular area, and then when we combine those together we end up with our final output. Now this might look different from run to run, as there is some randomness to how this is all initialized .

Do we actually even need a deep neural network to do this, though? One optimization is to remove layers to see if we can get away with it. Maybe we don't even need deep learning. Really this is kind of a simple thing -- stuff in the middle is blue, stuff on the outside is orange. Let's go ahead and remove one of these neurons from the output layer, as all we need is a binary result anyway, Can it still work? It does! In fact it's just as quick, so do I even need that layer at all? Let's go ahead and remove that final layer entirely. Still works, right? So for this very basic problem I don't even need to do “deep“ learning. All I have here is a single layer. It’s not even a multi-layer perceptron, it's just a perceptron. Do I even need four neurons in there? Well I think maybe I do, but one of them isn't really doing much. It’s basically doing a pass through and the inputs coming into it have been weighted down to pretty much nothing, so I bet you don't even need that one. Let's get rid of it. It still works. Isn't that kind of cool? I mean, think about that, we only have three artificial neurons and that's all it takes to do this problem. Compare that to the billions of neurons that exist inside your head.

We probably can't get away with less than that. Let's go ahead and try to do two neurons and see what happens. That doesn’t really work. So for this particular problem all you need is three neurons, but two won't cut it.

Let's play around some more, and try a more challenging dataset. So here's a spiral pattern, and you can tell this will be harder. We can't just say stuff in this corner is going to be this classification; we need a much finer-grained way of identifying these individual spirals. Again we're going to see if we can just train a neural network to figure that rule out on its own. Obviously two neurons won’t cut it; let's go back to four, and see if that's enough. I bet it isn't.

You can see it's trying, but it's really struggling. You can let this run for a while and you can see it's starting to kind of get there. The blue areas are converging on some blue areas. It’s really trying hard, but it's just not enough neurons to pull this one off. Let's go ahead and add another layer, and see if that helps. You can see it's doing more complicated things now that it has more neurons to work with, but still can't quite get to where it needs to be. Let's add a couple more neurons to each layer. Generally speaking you can either add more neurons to a layer or add more layers, it will generally produce the same results, but it might affect the speed in which it converges depending on which approach you take. It's just fascinating watching this work, isn't it? This one got stuck; it still can't quite pull it off. So let's add one more layer.

This is actually a very common pattern you'll see, you start off with a lot of layers at first and kind of like narrow them down as you go. So we're going to go to a initial input layer of six neurons to a hidden layer of four neurons, and then a layer of two neurons which will ultimately produce a binary output at the end. I think it's getting there. Here we go.

So technically... it's still kind of like refining itself, but it kind of did it, right? This is what we call "overfitting" to some extent. Obviously it has these tendrils that are cutting through here and that's not really part of the pattern we're looking for. But, it's still going. Those tendrils are getting weaker and weaker. It still doesn't have quite enough neurons to do exactly the thing that we would do intuitively, but this is a pretty complicated classification problem. It figured it out and it may be overfitting a little bit, but all we have is what, 12 neurons here? That’s pretty amazing!

This also illustrates that once you get into multiple layers it becomes very hard to intuitively understand what's going on inside the neural network. This gets kind of spooky. What does this shape inside a given neuron really mean? Once you have enough neurons, it's hard to fit inside your own head what these patterns all really represent. The first layer is pretty straightforward -- it's basically breaking up the image into different sections, but as you get into these hidden layers things start to get a little bit weird as they get combined together.

Let's go ahead and add some more, shall we? Actually let's add two more to this output layer and add one more layer at the end. Let's see if that helps things converge a little bit more quickly. It starts to struggle a little bit. See that? Like it's actually got a spiral shape going on here now. So with those extra neurons it was able to do something more interesting. We still have this this little spike here that's doing the wrong thing and it can't seem to quite think its way out of that one. Give it a few more neurons, though, and it might be able to figure it out. These ones are also misclassified, but I find it interesting that it actually created a spiral pattern here on its own, so maybe with a few more neurons or one more layer you could actually create an even better solution, but I will leave that as an exercise for you.

Just play around with this. I really encourage you to just mess around with it and see what kind of results you can get. The spiral pattern is in particular an interesting problem.

Just to explain some of the other parameters, we're doing a classification problem here. That's what we were doing throughout this section so far. For the activation function, we talked about not using a step function and using something else. ReLU is actually very popular right now. The regularization function, we haven't talked about yet. The learning rate is just basically the step size in the gradient descent that we're doing, so you can adjust that if you want to as well.

Let's see if ReLU actually makes a difference. I would expect it to just, you know, affect the speed. Look at that! That's pretty darn close to what we want, right? I mean, there's a part from this little tiny spike here which isn't really even there, it's a little bit of overfitting going there, but we have basically created that spiral shape just out of this handful of neurons.

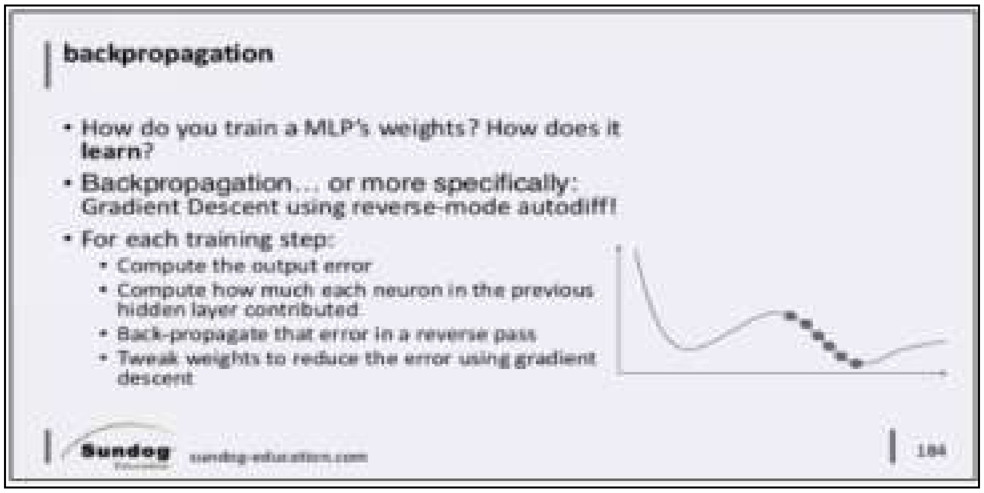
I could do this all day, and I hope you will too. Just play around -- it's so much fun, and it gives you such a concrete understanding of what's going on under the hood, Look at this hidden layer here, that's where the spiral shape is starting to emerge and come together. When you think about the fact that your brain works in very much the same way, it's quite literally mind blowing. Anyway, mess around with this -- it's a really great exercise, and I hope you have some fun with it.

Deep Learning Networks

deep learning

Sundog sundog-education.com 183

All right, I know you're probably itching to dive into some code by now. But there's a little more theory we need to cover with Deep Learning. I want to talk a little bit about exactly how they are trained and some tips for tuning them, now that you've had a little bit of hands-on experience with them using the TensorFlow playground.


backpropagation

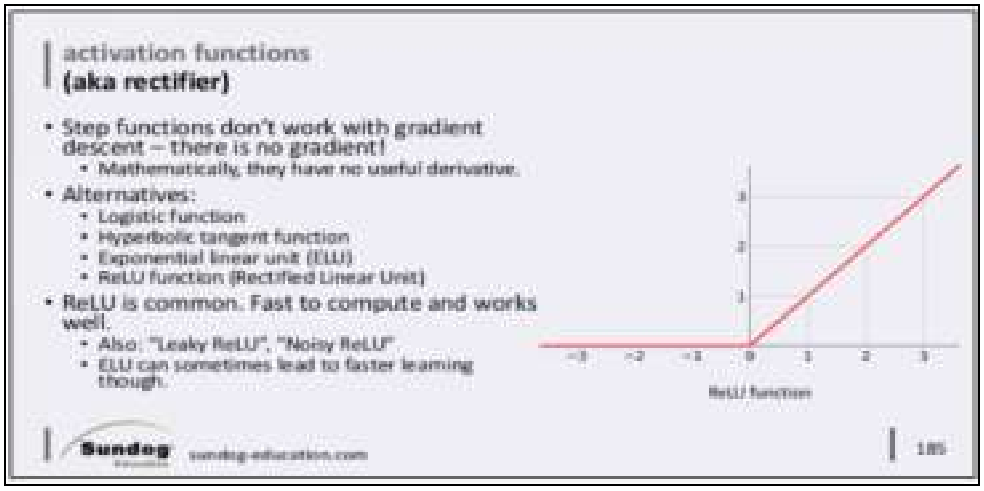
- How do you train a MLP's weights? How does it **learn**?
- Backpropagation… or more specifically: Gradient Descent using reverse-mode autodiff!
- For each training step:
  - Compute the output error
  - Compute how much each neuron in the previous hidden layer contributed
  - Back-propagate that error in a reverse pass
  - Tweak weights to reduce the error using gradient descent

Sundog sundog-education.com 184

How do you train a multilayer perceptron? Well, it's using a technique called "backpropagation." It's not that complicated really at a conceptual level, all we're doing is gradient descent, like we talked about before, using that mathematical trick of reverse mode autodiff to make it happen efficiently. For each training step we just compute the output error for the weights that we have currently in place for each connection between each artificial neuron, and then this is where the backpropagation happens.

Since there's multiple layers to deal with, we have to take that error that is computed at the end of our neural network and back-propagate it down in the other direction. We push it back through the neural network backwards. In that way, we can distribute that error back through each connection all the way back to the inputs using the weights that we are currently using at this training step.

It’s a pretty simple concept. We just take the error, and we use the weights that we are currently using in our neural network to back-propagate that error to individual connections. Then we can use that information to tweak the weights through gradient descent to actually try and arrive at a better value on the next pass, which is called an epoch, of our training passes. So that's all backpropagation is: we run a set of weights, we measure the error, we back-propagate the error using that weights, tune things using gradient descent and try again, and we just keep doing this over and over again until our system converges.



activation functions
(aka rectifier)

- Step functions don't work with gradient descent – there is no gradient!
  - Mathematically, they have no useful derivative.
- Alternatives:
  - Logistic function
  - Hyperbolic tangent function
  - Exponential linear unit (ELU)
  - ReLU function (Rectified Linear Unit)
- ReLU is common. Fast to compute and works well.
  - Also: "Leaky ReLU", "Noisy ReLU"
  - ELU can sometimes lead to faster learning though.

Sundog sundog-education.com 185

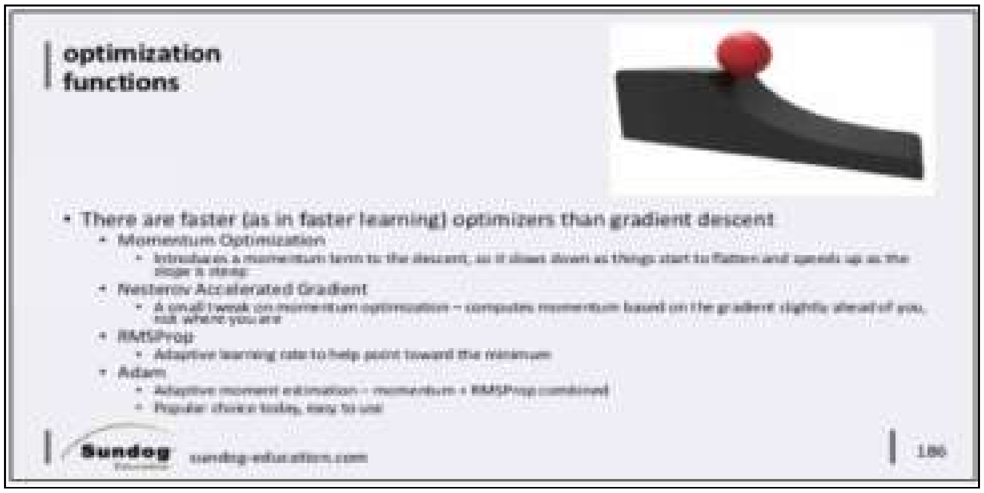
We should talk a little bit about activation functions. So in our previous exercise using the TensorFlow playground we were using the hyperbolic tangent activation function by default, and then we switched to something called ReLU and we saw that the results were a little bit better. What was going on there?

Well, the activation function is just the function that determines the output of a neuron given the sum of its inputs. So you take the sum of all the weights of the inputs coming into neuron, and the activation function is what takes that sum and turns it into an output signal. Like we talked about way back in the first lecture of this section, using a step function is what people did originally, but that doesn't really work with gradient descent because there is no gradient there. It's a step function, -there is no slope, it's either on or off. It's either straight across or up and down. There's no useful derivative there at all, so that's why alternative functions work a better in practice.

There are some other ones called the "logistic function" or "the hyperbolic tangent function," that produces more of a, well, curvy curve. If you think about what a hyperbolic tangent looks like, it doesn't have that sharp cut off there at zero at the origin, so that can work out pretty well. There's also something called the "exponential linear unit," which is also a bit more curvy. What we ended up using though, was ReLU, which stands for "Rectified Linear Unit." That's what this graph here is showing. Basically it's zero if it's less than zero, and if it's greater than zero, it climbs up at a 45 degree angle, so it's just giving you the actual sum of the input weights as it's output IF that output is greater than zero. The advantage that ReLU has is that it's very simple, very easy, and very fast to compute. So if you're worried about converging quickly and about your computing resources, ReLU is a really good choice.

Now there are variants of ReLU that work even better, if you don't care so much about efficiency. One is called "Leaky ReLU." Instead of being flat left of zero, it actually has a little bit of a slope there as well, a very small slope, and again, that's for mathematical purposes to have an actual meaningful derivative there to work with, so that can provide even better convergence. There’s also something called "Noisy ReLU," which can also help with convergence; but these days ELU, the exponential linear unit, will often produce faster learning. ELU is gaining popularity now that computing resources are becoming less and less of a concern now that you can actually do deep learning over a cluster of PCs on network in the cloud.

That's what activation functions are all about.



optimization functions

- There are faster (as in faster learning) optimizers than gradient descent
  - Momentum Optimization
    - Introduces a momentum term to the descent, so it slows down as things start to flatten and speeds up as the slope is steep
  - Nesterov Accelerated Gradient
    - A small tweak on momentum optimization – computes momentum based on the gradient slightly ahead of you, not where you are
  - RMSProp
    - Adaptive learning rate to help point toward the minimum
  - Adam
    - Adaptive moment estimation – momentum + RMSProp combined
    - Popular choice today, easy to use

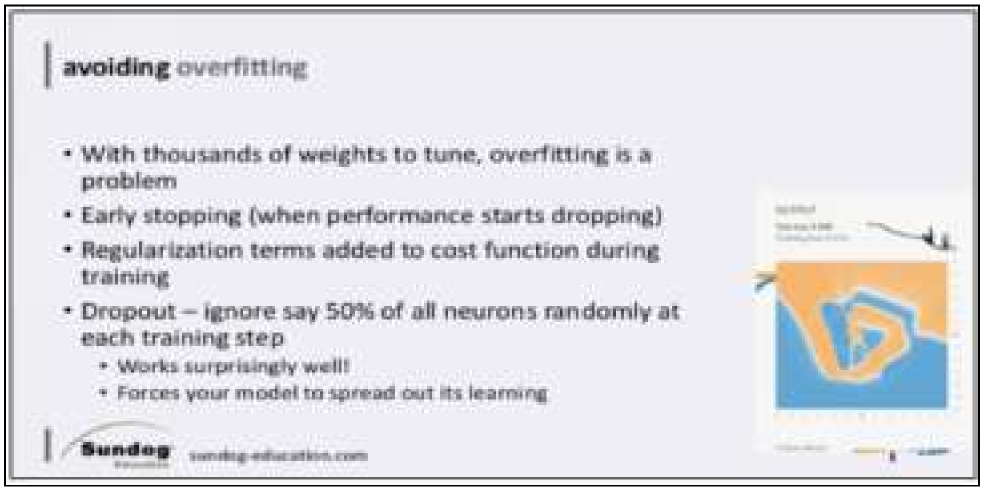
Sundog sundog-education.com 186

You can also choose different optimization functions. We've talked in very general terms about gradient descent, but there are variations of gradient descent that you can use as well. We talked a little bit earlier about momentum optimization; the idea there is to speed things up as you're going down a hill and slow things down as you start to approach that minima, so it's a way of just making the gradient descent happen faster by skipping over those steeper parts of your learning curve.

There's also something called the "Nesterov accelerated gradient," which is just a tweak on top of momentum optimization. It’s looking ahead a little bit to the gradient in front of you to take that information into account, so that works even better. There's also something called "RMSProp," which is just using an adaptive learning rate, that helps point you in the right direction toward the minimum. Remember back to how gradient descent works -- it's not always obvious which direction you're going to be heading in given the change in parameters, so RMSProp is just a more sophisticated way of trying to figure out the right direction .

Finally, there's something called "Adam," stands for "adaptive moment estimation," basically it's the momentum optimizer and RMSProp combined. It gives you the best of both worlds. Adam is a popular choice today because it works really well, and it's very easy to use.

Again, the libraries you will use for deep learning are very high-level and very easy to use, so it's not like you're going to have to implement Nesterov accelerated gradient from scratch. You're just going to say “optimizer equals adam” and be done with it. It's just a matter of choosing the one that makes sense for what you're trying to do, and making your own tradeoffs between speed of convergence, computational resources, and time required to actually do that convergence.



avoiding overfitting

- With thousands of weights to tune, overfitting is a problem
- Early stopping (when performance starts dropping)
- Regularization terms added to cost function during training
- Dropout – ignore say 50% of all neurons randomly at each training step
  - Works surprisingly well!
  - Forces your model to spread out its learning

Sundog sundog-education.com

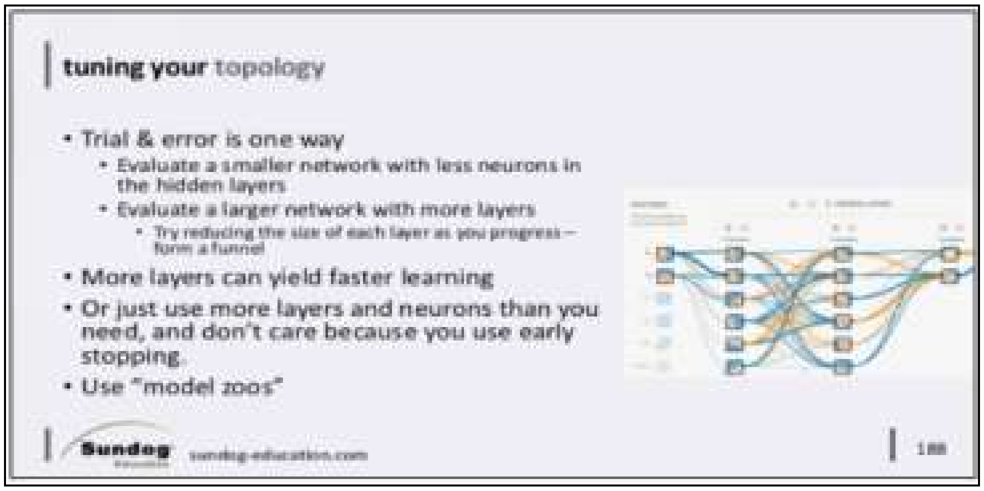
Let's talk about overfitting as well. You can see you'll often end up with patterns like this where you're not really getting a clean solution. You end up with these weird spikes sometimes, and sometimes if you let things go a little bit too long, it ends up reinforcing those spikes - those overfitted areas where you're not really fitting to the pattern you're looking for, you're just fitting to the training data that you were given .

Obviously, if you have thousands of weights to tune, those connections between each neuron and each layer of your neurons can add up really quickly. So it is very easy for overfitting to happen. But fortunately, there are ways to deal with it.

One is called "early stopping," so as soon as you see performance start to drop, that might be nature's way of telling you that it might be time for you to stop learning. At this point, maybe you're just overfitting.

There are also "regularization terms" you can add to the cost function during training, which are like the bias term that we talked about earlier. That can help too.

A surprisingly effective technique is called "dropout." It’s an example of a very simple idea that is very effective. The idea is just to ignore, say, half of the neurons randomly at each training step. Pretend that they don't exist at all. The reason this works is because it forces your model to spread out its learning. If basically you're taking away half of its brain at each training step, you're going to force the remaining half of those neurons to do as much work as possible. This prevents things where you have individual neurons taking on more of the work than they should. You even saw some of the examples that we ran in the TensorFlow playground where we would end up with neurons that were barely used at all, and by using dropout, that would have forced that neuron to have been used more effectively. Dropout is a very simple concept and very effective in making sure that you're making full use of your neural network, and it’s also effective at combating overfitting.



tuning your topology

- Trial & error is one way
  - Evaluate a smaller network with less neurons in the hidden layers
  - Evaluate a larger network with more layers
    - Try reducing the size of each layer as you progress – form a funnel
- More layers can yield faster learning
- Or just use more layers and neurons than you need, and don't care because you use early stopping.
- Use "model zoos"

Sundog sundog-education.com 188

Let's talk about tuning your topology. Another way to improve the results of your Deep Learning network is to just play games with how many neurons you have and how many layers of neurons you have – that arrangement of neurons is what we call your topology.

One way of finding the optimal topology is just trial and error, like what we did in TensorFlow playground. But there can be a methodology to your experiments. You can start off with the strategy of evaluating a smaller network with less neurons in the hidden layers, or you can evaluate a larger network with more layers. The question you're asking is: can I get away with a smaller network and still get good results? Then you just keep on making it smaller and smaller until you find the smallest it can safely be. Or, you can try to make your network larger and larger, and see at what point it stops providing more benefits to you. In short, just start sizing things differently and see what works and what doesn't.

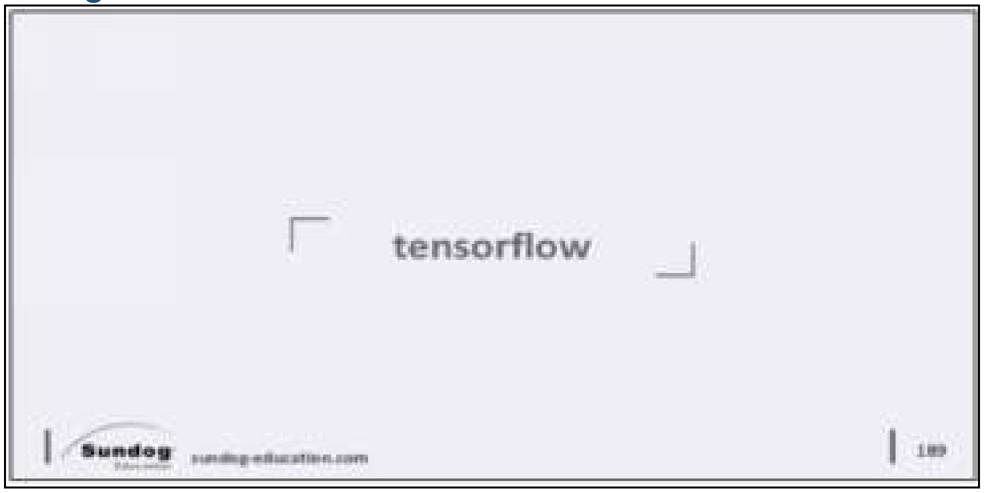
There's sort of a spooky aspect to how this stuff all works together. It's very hard to understand intuitively what's going on inside of a deep neural network, so sometimes you just have to use your intuition to try to tune the thing and get at the right number of resources you need. Also, in today's modern computing environment, sometimes you don't really care so much. It's probably OK to have a deep neural network that has a few more neurons that it really needs. I mean, what's the real expense involved in that these days? Probably not much, unless your data is truly massive like YouTube or something.

In general however, more layers will often yield faster learning then having more neurons and less layers. If you care about speed of convergence, adding more layers is often the right thing to do.

You can also use something called "model zoos." There are actually libraries out there of neural network topologies for specific problems. If you don't think you're the first person in the world to solve a specific problem with a deep neural network, maybe you should check out one of the model zoos out there to see if someone's already figured out the optimal topology for what you're trying to achieve, instead of trying to reinvent the wheel. People share these things for a reason, and it can save you a lot of time.

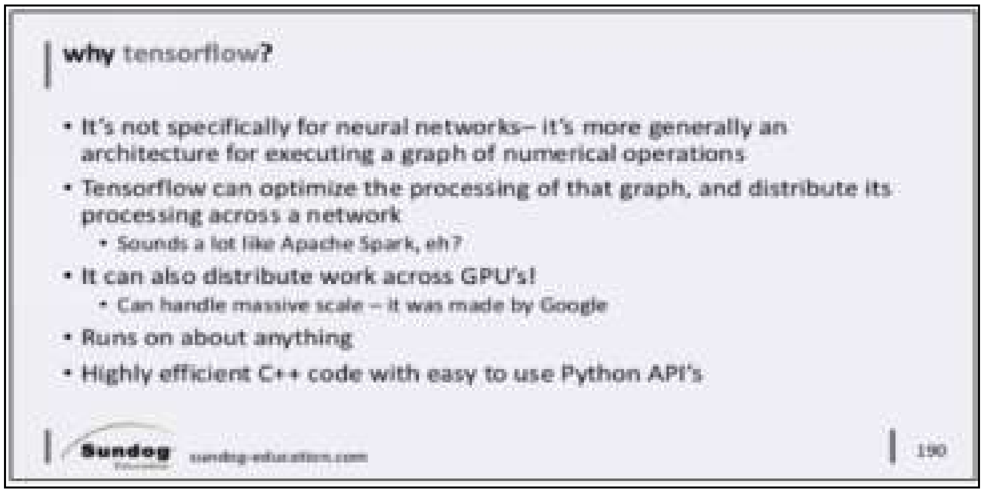
Enough theory - in our next lecture will get our hands dirty with TensorFlow and start writing some real Python code to implement our own neural networks.

Using TensorFlow


tensorflow

Sundog Education sundog-education.com 189

If you've done any previous research in deep learning, you've probably heard of the TensorFlow library. It's a very popular framework developed by the folks at Google, and they've been kind enough to make it open source and freely available to the world. So let's talk about what TensorFlow is all about, and how it can help you construct artificial neural networks.


why tensorflow?

- It's not specifically for neural networks– it's more generally an architecture for executing a graph of numerical operations
- Tensorflow can optimize the processing of that graph, and distribute its processing across a network
  - Sounds a lot like Apache Spark, eh?
- It can also distribute work across GPU's!
  - Can handle massive scale – it was made by Google
- Runs on about anything
- Highly efficient C++ code with easy to use Python API's

Sundog Education sundog-education.com 190

The thing that kind of took me by surprise when I first encountered TensorFlow was that it's not really purpose-built for deep learning, or even for neural networks in general. It's a much more general purpose tool that Google developed that just happens to be useful for developing deep learning and neural networks. More generally, it's an architecture for executing a graph of numerical operations. It's not just about neural networks; you can have any sequence of operations and define a graph of how those operations fit together. What Tensorflow actually does is figure out how to distribute that processing across the various GPU cores on your PC, or across various machines on a network, and make sure that you can do massive computing problems in a distributed manner. In that respect, it sounds a lot like Apache Spark. If you've taken any other courses from me, you've probably heard me talk about Spark; it's a very exciting technology, and Spark is also developing Machine Learning and AI and deep learning capabilities of its own. So in some ways,

TensorFlow is a competitor to Apache Spark, but there are some key differences that we should talk about.

It's not just about distributing graphs of computation across a cluster or across your

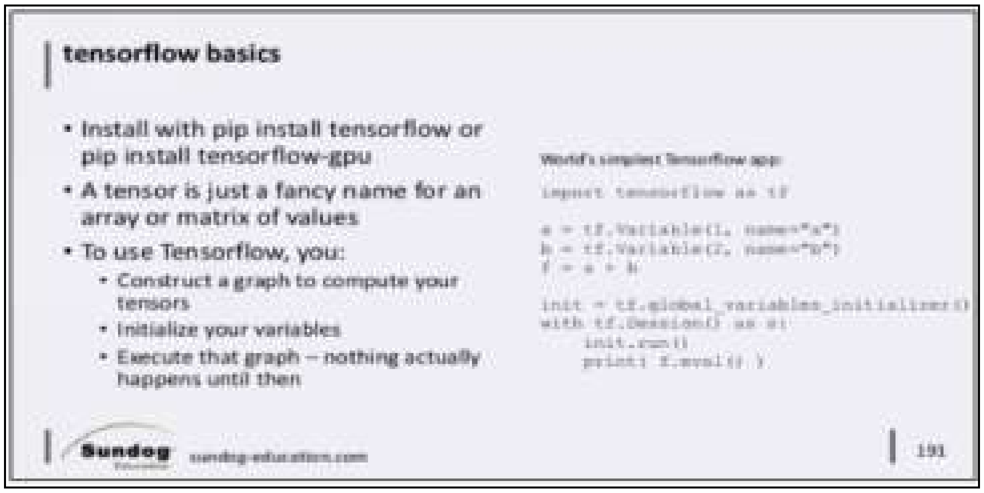
GPU. You can also run on just about anything. So one thing that's special about

TensorFlow is that I can even run it on my phone if I want to; it's not limited to running on computers in a cluster in some data center. That's important because in real world, you might want to push that processing down to the end user's device. Let's take the example of a self-driving car. You wouldn't want your car to suddenly crash into a wall just because it lost its network connection to the cloud now, would you? The way that it actually works is that you might push the actual trained neural network down to the car itself and actually execute that neural network on the computer that's running embedded within your car, because the heavy lifting of deep learning is training that network, right? So you can do that offline, push the weights of that network down to your car, which is relatively small, and then run that neural network completely within your car itself. By being able to run TensorFlow on a variety of devices, it opens up a lot of possibilities of actually doing deep learning on the edge -- on the actual devices where you're trying to use it on.

Tensorflow is written in C++ under the hood, whereas Spark is written in Scala, which ultimately runs on top of a JVM. By going down to the C++ level with Tensorflow, that's going to give you greater efficiency. But at the same time, it has a Python interface, so you can talk to it just like you would any other Python library. That makes it easy to program and easy to use as a developer, but very efficient and very fast under the hood.

The other key difference between TensorFlow and something like Spark is that it can work on GPU’s. A GPU is just your video card, the same video card that you're using to play Call of Duty on or whatever it is you play. You can actually distribute the work across the GPU cores on your PC, and it's a very common configuration even have multiple video cards on a single computer and actually use that to gain more performance on clusters that are purpose built for deep learning.

Plus, Tensorflow is free and it's made by Google. Just the fact that it's made by Google has led to a lot of adoption. There are competing libraries out there to TensorFlow, but TensorFlow as of right now is still by far the most popular.



tensorflow basics

- Install with pip install tensorflow or pip install tensorflow-gpu
- A tensor is just a fancy name for an array or matrix of values
- To use Tensorflow, you:
  - Construct a graph to compute your tensors
  - Initialize your variables
  - Execute that graph – nothing actually happens until then

World's simplest Tensorflow app:

```
import tensorflow as tf

a = tf.Variable(1, name="a")
b = tf.Variable(2, name="b")
f = a + b

init = tf.global_variables_initializer()
with tf.Session() as s:
    init.run()
    print( f.eval() )
```

Sundog sundog-education.com 191

Installing Tensorflow is really easy. All you have to do is use the pip command in your Python environment to install TensorFlow, or you can use Anaconda Navigator to do it all through a graphical user interface. There's also a tensorflow-gpu package you can install instead if you do want to take advantage of GPU acceleration. If you're running this on Windows, I wouldn't quite go there yet -- I have had some trouble getting tensorflow-gpu to work on my own Windows system. You’ll find that a lot of these technologies are developed primarily for Linux systems running on a cluster, so if you're running on a purpose built computer in a cluster on EC2 that's made for deep learning, go ahead and install tensorflow-gpu, although it's probably going to be installed for you already.

Let’s talk about what TensorFlow is all about. What is a tensor, anyway? Well, this is another example of fancy pretentious terminology that people use to make themselves look smart, At the end of the day, a tensor is just a fancy name for an array or a matrix of values. It’s just a structured collection of numbers. That's it. That's all a tensor is .

Using TensorFlow can be a little bit counterintuitive, but it's similar to how something like Apache Spark would work, too. You don't actually execute things right away; instead you build up a graph of how you want things to execute, and then when you're ready to execute it you say "OK TensorFlow, go do this." Tensorflow will then go and figure out the optimal way to distribute and parallelize that work across your entire set of GPU's and computers in your cluster.

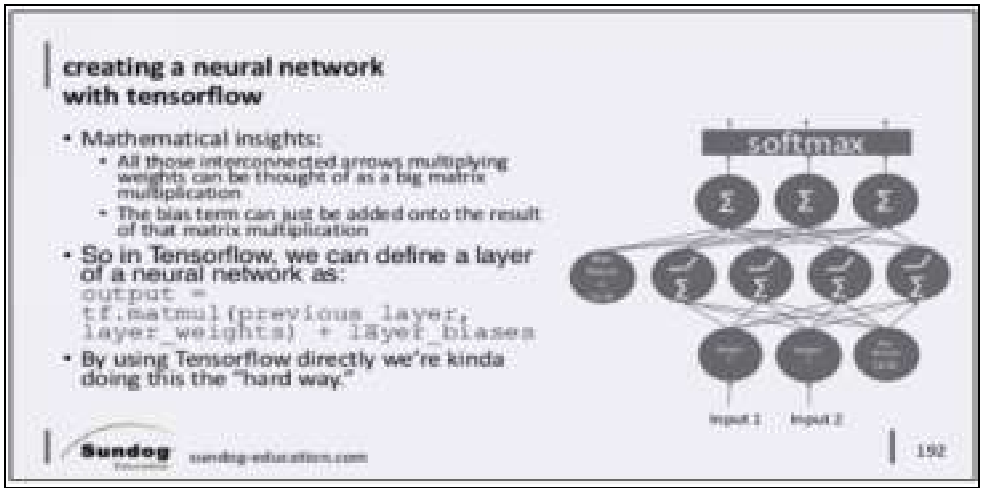
So let's take a look at the world's simplest TensorFlow application here in Python. All this is going to do is add one plus two together. But it's a good illustrated example of what's actually going on under the hood. We start by importing the TensorFlow library. We're going to refer to it as "tf" as a shorthand. We'll start off by saying a = tf.variable(1 ,name = "a“), and all that is doing is setting up a variable in TensorFlow, a variable object which contains a single value, 1, and which it's going by the name of "a," The name is what will appear in visualization tools for your graph if you're using something like that. Internally we're going to assign that to a variable in Python called "a" as well. Then we set up a "b" variable as well that is assigned the value 2, and given the name "b."

Here’s where the magic starts to happen. We say f = a + b, and you might think that that would put the number 3 into the variable f, but it doesn’t. “f” is actually your graph; it's the connection that you're building up between the A and B tensors to add them. So f = a + b does not do anything except establish that relationship between a and b, and their dependency together on that f graph that you're creating.

The next thing we need to do is actually initialize those global variables. You explicitly need to say, "OK, now is the time that I want to go and put those initial values into my variables." We create an object called "init" which is our global variables initializer. Then we define a session within TensorFlow; we're going to call that session "s," although we're not actually going to refer to it explicitly within this block. We will take our global variables initializer object and run it; that will actually stuff the values 1 and 2 inside the a and b variables. It initializes them to the initial variables we defined when we created them.

Next we call f.eval(), and this is where computation will actually finally happen. Once we call eval() on our f object, it will say: "OK, I need to create a graph that takes the a variable, which contains 1, and the b variable, which contains 2, and add them together." It will figure out how to distribute that incredibly complicated operation (I'm being sarcastic), across your entire cluster, and that will ultimately print the result 3.

So we have just created the most complicated way imaginable of computing 1 plus 2; but, if these were larger tensors, dealing with larger data sets or, for example, a huge array or a matrix of weights in a neural network, that distribution of the work becomes important. So although adding 1 plus 2 isn't a useful exercise to do with TensorFlow, once you scale this up to the many, many connections in a big neural network, it becomes very important to be able to distribute these things effectively.



creating a neural network with tensorflow

- Mathematical insights:
  - All those interconnected arrows multiplying weights can be thought of as a big matrix multiplication
  - The bias term can just be added onto the result of that matrix multiplication
- So in Tensorflow, we can define a layer of a neural network as:

```
output =
tf.matmul(previous_layer,
layer_weights) + layer_biases
```

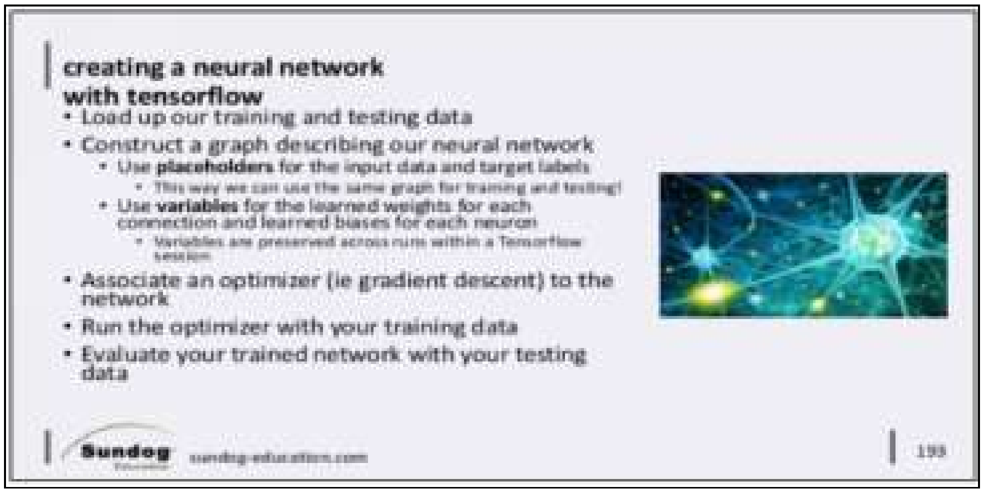
- By using Tensorflow directly we're kinda doing this the "hard way."

Sundog sundog-education.com 192

So, how do we extend this idea to neural networks? Well, the thing with TensorFlow is that it's not really made explicitly for neural networks. But, it can do things like matrix multiplication, and it turns out that if you think about applying all the different weights and sums that happen within a single layer of a perceptron, you can model that just as a matrix multiplication. You can just take the output of the previous layer in your multilayer perceptron, and do a matrix multiplication with a matrix that describes the weights between each neuron of the two layers that you're computing. Then you can add in a vector that contains the bias terms as well.

At the end of the day, you can modify this fancy diagram here of what a perceptron looks like, and just model it as a matrix multiplication. Go back and read up on your linear algebra if you want to know more about how that works mathematically, but this is just a straightforward matrix multiplication operation with a vector addition at the end for the bias terms.

By using TensorFlow we're kind of doing this the hard way, but there are higher-level API's in TensorFlow that make it much simpler and more intuitive to define deep neural networks. As we're describing TensorFlow at a low level, it's purpose in life is really just to distribute mathematical operations on groups of numbers – tensors -- and it's up to us to describe what we're trying to do in mathematical terms. It turns out it's really not that hard to do with a neural network.



creating a neural network with tensorflow

- Load up our training and testing data
- Construct a graph describing our neural network
  - Use **placeholders** for the input data and target labels
    - This way we can use the same graph for training and testing!
  - Use **variables** for the learned weights for each connection and learned biases for each neuron
    - Variables are preserved across runs within a Tensorflow session
- Associate an optimizer (ie gradient descent) to the network
- Run the optimizer with your training data
- Evaluate your trained network with your testing data

Sundog sundog-education.com 193

For us to actually create a complete deep learning network from end to end, there's more to it than just computing the weights between different layers of neurons. We have to actually train this thing somehow and actually run it when we're done, so the first thing we need to do is load up the training data that contains the features that we want to train on, and the target labels. To train a neural network, you need to have a set of known inputs with a set of known correct answers that you can use to actually descend or converge upon the correct solution of weights that lead to the behavior that you want.

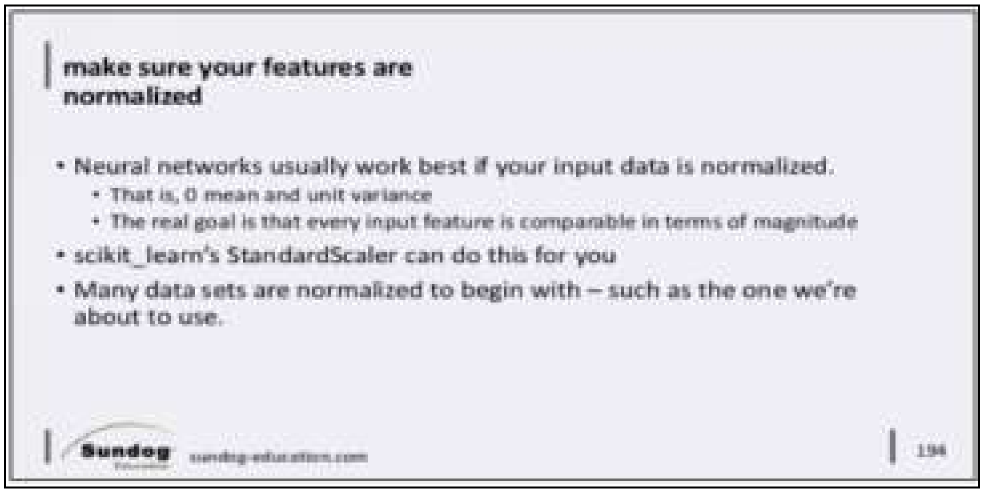
We're going to construct a graph just like we did before, when we had a graphic that just added “a” and “b” variables together. Let’s introduce a new concept called a "placeholder" as well. It gives us a placeholder within your graph for various data, which allows us to use the same graph structure for both our training and for our testing. We can use the same placeholder for input values and feed in our training data, and do gradient descent on the resulting network. After we're done, we can use a different set of data and feed it into those same input placeholders, and actually use that to test the results of our trained neural network on data that's never seen before.

We will use variables, like we saw before, to keep track of the learned weights for each connection. As we iterate through different training steps on our neural network, we're going to use variables to make sure that we have some memory in between runs of what those weights were between these connections and between each artificial neuron, and tweak those weights using gradient descent as we iterate through more and more training steps.

After that, we need to associate some sort of an optimizer to the network.

TensorFlow makes that very easy to do. It can be gradient descent or some variation thereof, such as Adam. We will then run our optimizer using our training data, and again, TensorFlow makes that pretty easy to do as well. Finally, we'll evaluate the results of our trained network using our test datasets.

To recap at a high level: we're going to create a given network topology, and fit the training data using gradient descent to actually converge on the optimal weights between each neuron in our network. When we're done, we can evaluate the performance of this network using a test dataset that it's never seen before, and see if it can correctly classify that data that it was not trained on.



make sure your features are normalized

- Neural networks usually work best if your input data is normalized.
  - That is, 0 mean and unit variance
  - The real goal is that every input feature is comparable in terms of magnitude
- scikit_learn's StandardScaler can do this for you
- Many data sets are normalized to begin with – such as the one we're about to use.

Sundog sundog-education.com 194

One other gotcha: when you're using neural networks, it's very important to make sure that your input data is normalized, meaning that's all scaled into the same range. Generally speaking, you want to make sure that your input data has a mean value of 0 and unit variants. That's just the best way to make the various activation functions work out mathematically. What's really important is that your input features are comparable in terms of magnitude, otherwise it's hard to combine those weights together in a meaningful way. Your inputs are all at the same level at the bottom of your neural network, and fitting into that bottom layer, it's important that they're comparable in terms of magnitude.

For example, if I already created a neural network that tries to classify people based on their age and their income, age might range from zero to 100, but income might range from zero to a million. Those are wildly different ranges, so those are going to lead to real mathematical problems if not scaled down to the correct range at first. Fortunately, Python's scikit_learn library has a StandardScaler package that you can use to do that automatically with just one line of code. All you have to do is remember to use it, and many datasets that we use while researching will be normalized to begin with. The one we're about to use is already normalized, so we don't actually have to do that. Later on the course, I'll show you an example of actually using StandardScaler.

With that, let’s dive in and actually try it out.

let's try it out

Sundog sundog-education.com 195

We're finally at a point in the course where we're going to actually write some real code and actually build a real neural network using tensorflow, so let's dive in with our next lecture and see how it works in detail.

All right, enough talking about TensorFlow, let's actually apply it! So, first thing we need to do is actually install TensorFlow itself. The recommended method is to use pip to install Tensorflow instead of conda. To do this, open up Anaconda Navigator, and select the RecSys environment. From here, you want to click on the little arrow next to the RecSys environment name, and select “open terminal.” In that terminal, enter:

```
pip install -–ignore-installed -–upgrade tensorflo w
```

If you know about the tensorflow-gpu package that accelerates tensorflow, resist the temptation to install it instead. There are currently issues with it on Windows that we don’t want to deal with.

Make sure everything goes smoothly, and that you end up with at least TensorFlow version 1.9. We’re also going to need to install a package called matplotlib, which we’ll use for visualizing things in our notebook. To do that, go back to Anaconda Navigator and your RecSys environment, and select “not installed.” From here, search for matplotlib, select it, and hit the apply button to start its installation process.

All right, with that let's actually open up our notebook. While you still have the RecSys environment active, switch to the Home view in Anaconda Navigator, and launch the Jupyter Notebook application. If you skipped the “Intro to Python” section of the course, you may need to install Jupyter Notebook first by hitting the “install” button, and let it do its thing for a while. Once it’s launched, it should open up a page in your browser with a list of scripts; you need to import our script in here. Click on “upload”, then navigate to the “DeepLearningIntro” folder in your course materials, and select the Tensorflow.ipynb file. Click the “upload” button to import it, and then you can select the Tensorflow notebook and actually bring it up.

Let's actually dive in and actually play with TensorFlow, shall we? Again,TensorFlow’s lower level API is kind of a hard way of doing things, but if you can get through this you'll find the rest of this section pretty easy. So let's start by looking at the World’s Simplest TensorFlow Application that we did in the previous lecture, and see how that works.

If we just click in this little code block here, and we've already installed TensorFlow, all we need to do is import it. Then we'll call it "tf" as shorthand. As we did before, we're going to create an "a" variable that contains the number 1 initially, a "b" variable that contains a number 2, and create a graph called "f" that just adds those two variables together. We will then create a global variables initializer, create a session in TensorFlow, run the global variables initializer to actually put the initial values of 1 and 2 into "a" and "b" respectively, and actually evaluate that graph that we actually defined to add the two things together and print the results of that. So again, we've made the world's most complicated way of adding one and two together.

Let's go ahead and hit shift-enter to actually execute that block... and it's actually spinning up TensorFlow and figuring out how to distribute 1 + 2 across my entire system. Sure enough, the answer is 3, so TensorFlow is working. If you got some error there, then there's some installation issue you need to go back and deal with.

Next, let's do something a bit more interesting than adding 1 and 2 together. We'll implement some actual handwriting recognition. This is going to be using something called the MNIST dataset, and if you've looked at other tutorials in the field of Deep Learning, you might be saying "Oh no! Not another MNIST example!" But, there's a reason people use the MNIST handwriting data set so much. First of all, it's built right into TensorFlow, so it's very easy to use. It's also very fun -- we're actually going to be doing handwriting recognition here! The MNIST data set contains 70000 handwriting samples of people writing the number 0 through 9, and our challenge is to predict which number each handwritten image represents. It's a fun task - it's mimicking to some extent how your brain actually recognizes handwritten numbers. Some people's handwriting is pretty bad, so sometimes this can be a challenge.

Let's talk about the nature of the data. It's broken up into a training dataset and a testing dataset, and each image is a 28 by 28 image of grayscale pixels. For the purpose of this example we're going to be starting things off a little bit simple; we're just going to treat each pixel as an individual input into a deep neural network, so we're just flatten all these images into a flat 1D array of 784 pixels. So, we take the first 28 rows from the top of the image, stick the next 28 pixels from the next row, then stick on the next 28 pixels from the next row, and ultimately we just have a list of 784 values that represent the grayscale value of each pixel of that 28 by 28 image.

It makes more sense when we look at some examples. Let's start by loading the data set up. As I said, it's built into TensorFlow itself. Then, we'll create an interactive session in TensorFlow, which makes it easier to work with TensorFlow within an iPython environment. This removes the need to explicitly create a session and loop within it. We want to break up this work across multiple code blocks, and that’s what an interactive session lets us do. Finally we'll just call mnist.load_data() to actually load up the MNIST dataset.

A quick word about notation, here. In machine learning, we typically call feature data “x” and label data “y”. So, “x_train” refers to the training images themselves, while “y_train” refers to the labels for those training images – that is, which number each image actually represents. For the data we’ve reserved for testing, “x_test” refers to the test images, and “y_test” to the test labels.

We need to do some pre-processing of our data to get it into the format our neural network wants. First we’re going to “flatten” the 2D image data into one dimension, so instead of 28x28 pixels for each image, we’ll consider each image as a onedimensional array of 784 pixels instead. That’s what these “reshape” functions do. Next, we divide that image data by 255, because the source image data is 8-bit grayscale data where each pixel is an integer between 0 and 255, 0 representing black and 255 representing white. Neural networks want their inputs normalized, so by dividing each pixel by 255, we end up with values between 0 and 1 instead .

We want our output of the neural network to be “one-hot encoded.” Ultimately, this neural network is going to have 10 output units, and each one of those outputs will represent some weight of how much it thinks that individual number represents the classification of that numerical image. So one-hot means that every number from 0 through 9 will be encoded as a binary pattern. Let's skip ahead a little bit here down here where I talk about it a little bit more. For example, the number 1 would be encoded as 0 1 0 0 0 0 0 0 0 0. We have this 1D array of binary values, 0 or 1, and where that 1 appears indicates what the actual label of that image is. If I have an image with the number 1 that somebody wrote, I would encode that in one-hot format as a array of 10 values where the value 1 appears in element 1. Remember we start counting at zero in Python world, so if there were the number 0, the 1 would be in the first slot. For the number 2 the number 1 would be in the third slot instead. That's what one-hot encoding means. It's just a more optimal way of doing classification that's more finely tuned to the output of our neural network.

To convert our label data into “one-hot” format, we use the handy to_categorical function included with TensorFlow here.

Let's talk about the shape of our data. We have a training dataset of 60,000 images and then a test dataset of 10000 images. We use the test data to evaluate the performance of our neural network by giving it data that's never seen before, and measuring how well it can predict the correct classification of each image. So we say that the shape of our training data is (60,000, 784.) That means that we have 60,000 images each with 784 data points within them. Again, each image has 28 rows and columns of pixels, and 28 times 28 is 784, so our input tensor shape is going to be 60,000 by 784 – 60,000 images with 784 values on each image.

The test data is just going to be the label data, so that will be of shape (60000,10.) Again, we have 60,000 input training data points and we have our one-hot encoding that is a array of 10 values. Wherever the number 1 appears represents the correct label. So more specifically here when we talk about a shape of (60000, 10), that's going to be the shape of our training label data. That's what we're working toward when we’re training the network.

During the test phase itself we only have 10000 samples, so the shape of that test data would be (10000, 10) in that case. Let’s hit shift-enter to execute this code block and load everything up.

Let's take a look at our sample data here, just so we have more of an intuitive understanding of what we're up against. It’s very important to always understand your input data because often there's going to be problems with your source data. Perhaps there are outliers that you need to deal with, and you want to really understand at an intuitive level what you're up against. So let's go ahead and display some samples here. What's going on in this next block is a little function called display_sample for a given input element. so when I say, for example, display_sample(1234), that means pick the 1,234th input sample from our training dataset and display it to me. What we're doing here extracting the label in one-hot format for that particular sample of the training data. Then we convert that to a human readable number by using the argmax function. All that does is say "take the array element with the largest value and display that array element's value back to me," so that's a little quick way of converting one-hot format back to a human readable digit. We will then reshape that 1D image. Remember each set of inputs is just a flat array of 768 pixels. So we need to take each row and just stick it next to each other to actually visualize that as a two dimensional image in the same way that your brain sees it. We're going to want to reshape that to a 28 by 28 2D array, and then we can use Python's built-in matplotlib library to actually display that as an image. Let's go ahead and hit shift-enter, and we can see that the training data sample 1234 actually corresponds to an image of someone drawing the number 3. Here is what the one-hot labeled data looks like there -- you can see there's just a 1 in the fourth value that represents the value 3. And we reshape that to 28 by 28 and display it as a greyscale image, we see it's not a bad example that actually looks like a 3. Someone's handwriting was halfway decent there, at least! So that's the kind of training data that we're dealing with here. All we're getting as an input is this image, flattened, along with the target label. And we have 60,000 such samples with which we can train our neural network.

Just to drive home exactly what your neural network is "seeing," let's go ahead and visualize what the input data looks like. What this image is showing is the first five hundred training samples flattened out in the same format that your neural network is going to get it. So this is stretching out over 784 numerical values for each value. From 0 to 500, we have each individual input image flattened out into a 1D array, 784 values. Instead of constructing this as individual 2D images, which is how your brain might process it, instead we're going to create a neural network that actually tries to recognize patterns in these flattened rows instead. We're just going to go through the first 500 here, reshape them to the 1D format that we're actually going to be feeding into our neural network, concatenate those all into a single image and display that.

You can see here we have these bands that correspond to the center of the image on each row. It's a little bit that this is the raw input into the neural network that we're going to build. Each row represents a training sample being fed into our neural network as we train it. What that neural network is "seeing" is just one row of that data at a time. Your own brain would have a very hard time trying to figure out, for example, does this particular row represent the number 2 or not? There's no way in heck you’re going to figure that out just by looking at it, but we're going to create our own little special neural network that can. Later on we'll talk about how to do image processing in a way that's closer to how your brain actually processes things, but I think it's remarkable that this works at all using this naive approach.

Let's actually start running some code to put it all together and build our graph in TensorFlow. The first thing we need to do is create some placeholders for the input training data and the target labels, so we will set up a placeholder called "input_images" of shape (None, 784.) What that means is we’re creating a placeholder for the training data, or the data being input into our neural network, that has some arbitrary number of input images, and each one of those images will contain 784 values. That just defines the shape of the data that's going into our neural network. We also need to create a placeholder for the target labels and that will have a shape of some arbitrary number of target images and a one-hot array of 10 values that we use to evaluate the actual label. So the point of doing a placeholder here is that we can re-use this same neural network for training and for testing, and for actually using it in practice. For our first pass where we're training our neural network, our input images will be assigned to our training dataset images, the target labels placeholder will be assigned to the labels of that training dataset and we will use that to actually run gradient descent and actually try to tune the weights between each neuron. Later on when we're testing the performance of this neural network, our input images will be assigned to the images in our test dataset, the target labels will be assigned to the known labels of those testing dataset images, and then we can measure how well our neural network predicted the correct label versus that target label set. And if you're going to actually use this in the real world, you wouldn't be using the target labels at all -- you would just assign the input images to the images that you want to classify and look at the output and target labels and see what it came up with.

We still need to run this stuff, so let's go ahead and execute that block with the input image there and see that it actually works. In this block we'll go ahead and set those placeholders, to select that block, hit shift-enter, and that should be all set up. Now again, nothing really happens in TensorFlow until we actually evaluate things and actually execute things; we’re just setting up the graph at this point.

Let's go ahead and set up the topology of our neural network itself. We're going to set up a multilayer perceptron here. We'll start off with 784 input values, so what this first line is saying is to create a tensor called input_weights that's going to be a variable that is normalized, of shape (784, hidden_nodes.) That means that we're going to create a layer here that has 784 input signals, that's one for each input pixel of each image, and it's going to have an output of 512 hidden nodes going to the next layer. This is setting up a layer of our neural network that takes 784 inputs and has outputs going to 512 inputs to the next layer.

We also need to set the input biases, which is just going to be a 1D array of the 512 hidden nodes in between, so this is going to be the bias terms on that layer between the input layer and the hidden layer. Now again, in TensorFlow we have to do this all the hard way; we're just describing things in terms of linear algebra here, so we're just going to have these matrices of weights and vectors of biases that we're passing around. For the hidden layer itself, we're going to have an input of 512 and output of 10 going to our final classification layer in one-hot format, and we will also have a bias vector applied to that as well that is 10 wide, that will be applied to the final output layer .

Let's go ahead and hit shift-enter to get that set up. Next, we need to put it all together, so we're going to create an input layer that does a matrix multiplication between that input image layer and the input weights that we set up as a variable.

Again, remember these input weights are a variable that will contain those weights.

Over time we will tweak them, so it maintains memory between each training run. We will then define our hidden layer as using the Relu activation function on the sum of that input layers, matrix multiplication, and the bias vector that we set up previously. Finally, we'll compute our output layer as doing a matrix multiplication between that hidden layer’s output and the weight matrix of that hidden layer, and add in that hidden bias factor of 10 as well.

Let’s go ahead and hit shift-enter before we forget to actually set that up.

Moving back up to review block 6, what we're doing in this block is creating variables to hold the weights and biases between each training run. So these are maintaining the memory of the actual weights and bias terms within our neural network from run to run.

In the next block, we’re actually defining the connections between the layers themselves, so we're saying here that we're going to have our input layer that uses that variable that we created to store the weights of the input values. We'll create a hidden layer that also uses the output of that input layer and the bias variables that we assign for that. We also use the hidden weights and hidden biases variables as we define our final digit weights layer here that ultimately goes through our file classification step. It's a lot to wrap your head around, but we're doing it the hard way -- remember this is going to get easier, I promise .

What we've done is set up a very simple multilayer perceptron with an input layer of 784 inputs, one for each pixel of each image. There is a hidden layer of 512 neurons that goes to an output layer of 10, which represents the one-hot format of the final classification.

The next thing we need to do is set up a loss function. How do we actually measure the correctness of our results? The way we're going to do that is by using softmax, which we've talked before. We're going to use specifically softmax with something called "cross entropy;" all that does is apply a logarithmic scale to penalize incorrect classifications more than ones that are close, so that's a way of making things converge a little bit more quickly. We will assign the final output layer of our digit weights and use the target labels as one of these so-called correct answers that we want to converge toward. So, we create a loss function that takes a look at our final digit weights, applies softmax to them, and compares that to the target labels in onehot format. Let’s hit shift-enter to set that up.

Next we need to define an optimizer that will apply to this neural network. We're going to just use a straight-up GradientDescentOptimizer. The 0.5 is the learning rate, that's something you can play with if you want to. Obviously if it's too big, you might overshoot the actual minimum that you're looking for, but if it's too small, it might take too long. We will assign it the last function that we just defined to actually descend upon. Hit shift-enter.

Finally we need to create a function that measures the accuracy of our resulting neural network, so we will set up a function called correct_prediction. This is another graph in TensorFlow that we're setting up that will just evaluate whether the argmax of the final digit weights matches the argmax of the target labels, so if the largest final weight output of our neural network matches the largest one-hot format of our target labels, that'll give it a 1 value, otherwise it is 0. We will measure accuracy by taking the mean of all of those correct prediction results over all the test data that we actually measured. Shift-enter to execute that.

Let's actually run this thing now. So all we've done up to this point is define the actual graph within TensorFlow that we're going to run. Let's go ahead and run the global variables initializer to get those initial values into it. What we're going to do now is run 20 training steps of 100 images on each batch. This code randomizes the training data on each epoch, and then processes all of the randomized training data in batches of 100 images at a time. We pass in this dictionary in Python that just matches placeholders to actual data. So for this training run we're going to assign the batch of 100 training images to the input_images placeholder, and we will take the labels data in one-hot format and assign that to the target_labels placeholder and let that run and be optimized.

For every step, we will print out what epoch we're currently on and the accuracy that we've seen. This will just let us see the progress over time because if you didn't install the GPU accelerated version of TensorFlow, this can actually take a while. So let's go ahead and hit shift-enter and kick this off.

It's actually running pretty quickly, and you can see how that accuracy is getting better and better over time. Although at this point, we probably could have done early stopping because at around 94 percent it has a hard time doing any better.

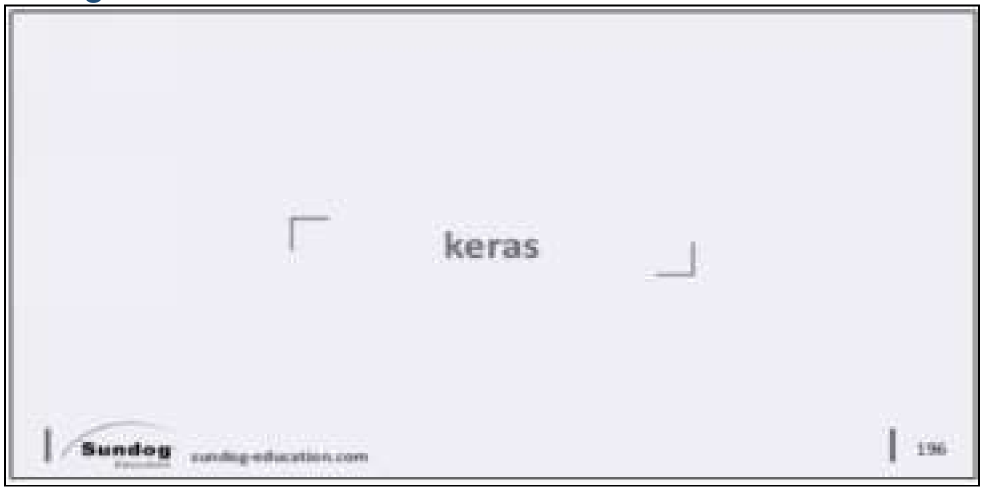
So there we have it! That's not bad, we got about 95 percent accuracy when correctly predicting the correct classification of a bunch of images out of the dataset, so, that's kind of cool. Let's actually take a look at our test data and evaluate the performance of this thing for real. Let’s go through the first 100 test images, and just print out any ones that are actually incorrectly classified, so we can get a more intuitive feel of the sorts of images that our neural network is having trouble with.

So we’ll go through the first 100 images, take each one, and the image will be flattened to a 784 wide array of pixels, which is what our neural network expects as input. We'll also pluck out the labels as well, and convert that to a human readable label using the argmax function. We will then run using the same session that we were using before, with the same neural network, using the input images of x_train_one that we extracted from the test dataset, and actually get the argmax result of that classification. So basically this is going through one image at a time, and feeding it into our neural network and seeing if it actually matches the predictive label or not. So we'll take our prediction that comes out of the neural network here, compare that to the label that we know is the one that was correct for this image, and if it does not match we'll print out the actual picture of the one that it got wrong. So let's run that.

Interesting. The numbers you see after running it may be different due to the random nature of training a neural network, but here’s a description based on what I saw. So for example, here is somebody who was trying to draw the number 5 and made this thing that our system thought was a 4. Not exactly the best handwriting there; you can kind of understand how it got confused. Here is, I guess that was supposed to be a 4, and we thought it was a 6, which is really not unreasonable. This is someone trying to draw the number 2, and for some reason we thought it was a 3 -- it's very messy, too. Here’s someone trying to draw a 6 that we thought was a 0, and this is an example of where your human brain probably wouldn't have much of a trouble with that. But, the fact that we're just trying to create a very limited neural network here that only deals with these things in a 1D fashion means that we're limited in effectiveness here. Later on, we'll talk about more techniques that can do this more efficiently, but for now, it's not too bad.

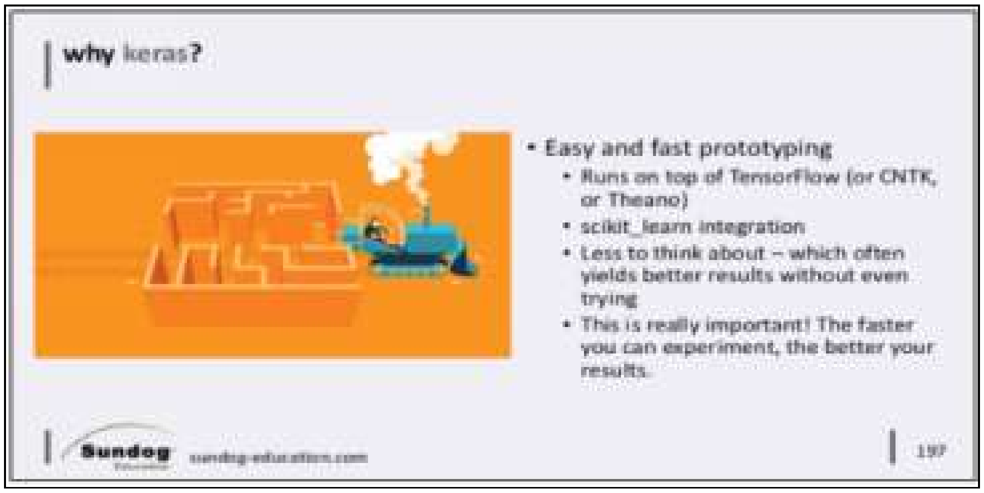
We actually constructed a neural network that can classify the correct number that someone was trying to draw more often than not! That's kind of cool. To see if you can improve upon it as an exercise for you to fiddle around with things, try using more hidden neurons, or try using fewer. What's the least amount that you can use before quality starts to suffer? Try adding another hidden layer! Try different batch sizes! Just mess around with things, get your hands dirty, see if you can do a better job. Can you actually top my results? Have some fun with it. And in our next lecture, we'll talk about how to make this all a lot easier using the Keras API.

Using Keras


keras

Sundog sundog-education.com 196

So we've had a look at developing neural networks using Tensorflow's lower-level APIs, where instead of really thinking about neurons or units, you're thinking more about tensors and matrices and multiplying them together directly. And that's a very efficient way of doing it, but it's not really intuitive. It can be a little bit confusing especially when you're starting to try to implement a neural network in those terms.



why keras?

- Easy and fast prototyping
  - Runs on top of TensorFlow (or CNTK, or Theano)
  - scikit_learn integration
  - Less to think about – which often yields better results without even trying
  - This is really important! The faster you can experiment, the better your results.

Sundog sundog-education.com 197

Fortunately, there's a higher level API called Keras that's now built into Tensorflow. It used to be a separate product that was on top of Tensorflow. But as of Tensorflow 1.9, it's actually been incorporated into Tensorflow itself as an alternative higher-level API that you can use. And it's really nice because it's purpose-built for deep learning, so all the code is very much built around the concept of artificial neural networks. That makes it very easy to construct the layers of your neural network and wire them together, and use different optimization functions on them. It's a lot less code and a lot less things that can go wrong as a result.

Another benefit of Keras, in addition to its ease of use, is its integration with the scikit-learn library. If you're used to doing machine learning in Python, you probably use scikit-learn a lot. Keras can actually integrate your deep neural networks with scikit-learn .

You may have noticed in our previous lecture that we kind of glossed over the problem of actually doing train/test or cross validation on our neural network, because it would have been rather difficult to implement. But with scikit-learn, it's very to do cross validation and perform proper analysis and evaluation of this neural network. That makes it easier to validate what we're doing, and to integrate it with other models, or even chain a neural network with other deep learning or machine learning techniques. There's also a lot less to think about -- and that means that you can often get better results without even trying.

With Tensorflow’s original API, you have to think about every little detail at the linear algebra level of how these neural networks are constructed because it doesn't really natively support neural networks out of the box. You have to figure out: how do I multiply all the weights together? How do I add in the bias terms? How do I apply an optimizer to it? How do I define a loss function? In contrast, Keras can take care of a lot of those details for you. Why is that important? Well, the faster you can experiment and prototype things, the better your results will be. If it's that much easier for you to try different layers in your neural network -- different topologies, different optimizers, different variations -- it’s going to be that much easier and quicker for you to converge on the optimal neural network for the problem you're trying to solve. The more time you can spend on the topology and tuning of your neural network, and less on the implementation of it, the better your results will be.

You might find that Keras is ultimately a prototyping tool for you. It's not as fast as just going straight to Tensorflow. Sometimes, you want to converge on the topology with Keras, and then go back and implement that at the lower-level Tensorflow layer. But again, just that ease of prototyping alone is well worth it.

Let's take a closer look. Let’s go back to the Jupyter Notebook – make sure you are in the RecSys environment first - and import the Keras.ipynb file from the DeepLearningIntro folder of your course materials. Once it’s uploaded, select it to fire up that notebook.

Again, Keras is just a higher-level API in Tensorflow that makes deep learning a lot easier. We’ll start off by importing the stuff that we need -- the Keras API and some specific modules from it. We have the MNIST dataset here that we're going to experiment with. And we’ll import the Sequential model, which is a very quick way of assembling the layers of a neural network. We're going to import the Dense and Dropout layers as well, so we can actually add some new things into this neural network to make it even better and prevent overfitting. Finally we will import the RMSprop optimizer, which is what we're going to use for our gradient decent. Hit shift-enter to process that block.

Let's go ahead and load up the MNIST data set that we used in the previous example. Keras' version is a little bit different. It actually has 60,000 training samples as opposed to 55,000, but still 10,000 test samples. That’s just a one line operation.

Now we need to convert this to the shape that Tensorflow expects under the hood.

So we're going to reshape the training images to 60,000 by 784. We’re going to still treat these as 1D images. We're going to flatten these all out into one dimension of 784 pixels for each 28 by 28 image. We also have our test data set of 10,000 images, each with 784 pixels apiece. And we'll explicitly cast the images as floating point 32bit values, which is just something to make the libraries a little bit happier. Furthermore, we're going to normalize these things by dividing by 255. The image data here is actually 8-bit at the source, so it ranges from zero to 255. To convert that to the range zero to one, we convert it to a floating point number first and then divide it by 255 to rescale that input data to the range zero to one. We talked before about the importance of normalizing your input data, and that's all we're doing here. Hit shift-enter to run this block, and we'll move on.

As before, we will convert our labels to one-hot format. That's what to_categorical does for you. It just converts the label data on both the training and test data set to one-hot zero-to-ten values. Let's select this block and run it as well.

Next we'll visualize some of that data just to make sure that it loaded up successfully. This is pretty much the same as the previous example. We're just going to look at our input data for sample number 1234. And we can see that our one-hot label here is showing one in position four. We start counting at zero: zero, one, two, three – our label is three. Argmax gives us back the human-readable label and by reshaping that 768 pixel array into a 2D shape, we can see that this is somebody's attempt to draw the number three. So far, so good. Our data looks like it makes sense and it was loaded correctly.

Now, remember back in when we were dealing with Tensorflow, we had to do a whole bunch of work to set up our neural network. Well, look at how much easier it is with Keras. All we needed to do is say that we're setting up a sequential model, which allows us to add individual layers to our neural network one layer at a time - sequentially, if you will. We start off by adding a dense layer of 512 neurons with an input shape of 784 neurons. So this is our first layer that takes our 784 input signals from each image, one for each pixel. It feeds it into a hidden layer or 512 neurons. And those neurons will have the Relu activation function associated with them. So with one line of code, we've done a whole lot of work that we had to do in TensorFlow before. And then on top of that, we'll put a softmax activation function on top of it to a final layer of 10 output units, which will map to our final classification of what number this represents from zero to nine. That was easy! Let's execute this block before we forget.

We can even ask Keras to give us back a summary of what we set up just to make sure that things look the way that we expected and sure enough, we have two layers here. One that has 512, and then going to a 10 neuron layer for the final classification.

Now, you also might remember that it was kind of a pain to set the optimization and loss function in the lower-level API, but that's a one liner in Keras. All we have to do is say that our loss function is categorical cross-entropy and it will know what to do there. We're going to use the RMSprop optimizer. You could use Adam instead if you wanted to. There are other choices like Adagrad and SGD, and you can read up on those in this link in the notebook if you want to. We will measure the accuracy as we go along. Go ahead and run that, and Keras will build the underlying graph that we want to run.

So now we actually have to run it. And again, that's just one line of code with Keras.

All we need to do is say that we're going to fit this model using this training data set. These are the input features and labels that we're going to train with. We want to use batch sizes of 100. We're going to run that over 10 epochs. I'm going to set up verbose level of two because that's what works best with a Jupyter notebook. And for validation, we will provide the test dataset as well. So instead of writing this big function that does this iteration of learning by hand like we did in Tensorflow, Keras does it all for us. Let's go ahead and hit shift enter and kick that off. Keras used to be considerably slower than using Tensorflow’s lower-level API’s, because it's doing a little bit more work under the hood. But in today’s Tensorflow it’s pretty fast, and you'll see that the results are really good. I mean, even on the second iteration, we've already surpassed the accuracy that we got after 20 iterations in our hand-coded Tensorflow implementation. At epoch 6, we're approaching 99% accuracy in our training data. Keep in mind this is measuring the accuracy in the training data set. And we're almost there. I mean, even with just 10 Epochs, we've done a lot better than using Tensorflow. And again, Keras is doing a lot of the right things for you automatically without making you even think about it That's the power of Keras. Even though it can be slower, it might give you better results with less effort at the end of the day.

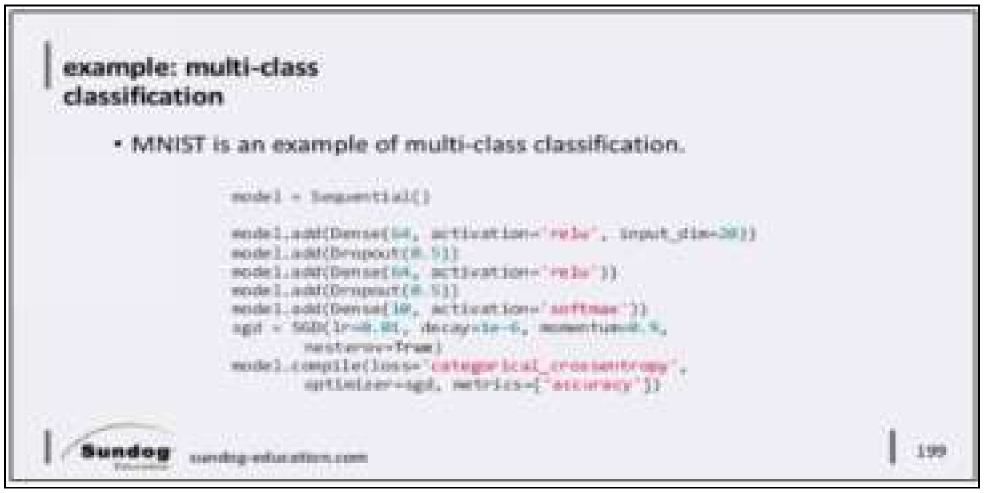
Now, here's something that we couldn't really do easily with Tensorflow. It's possible, but I didn't get into it because the previous lecture was long enough as it was. But remember that we can actually integrate Keras with scikit-learn. So we can just say model.evaluate, and our Keras model is just like a scikit-learn model as far as Python's concerned. And now we can actually measure accuracy based on our test data set. Using the test data set as a benchmark, it had a 98% success rate in correctly classifying those images. So that's not bad. Now, mind you, a lot of research goes into optimizing the MNIST data set problem, and 98% is not really considered a good result. Later in the course, we'll talk about some better approaches that we can use. But hey, that's a lot better than we got in the previous lecture, isn't it?

As before, let's go ahead and take a look at some of the ones that it got wrong, just to get a feel of where it has trouble -- things where our neural network has challenges. The code here is similar. We just go through the first 1,000 test images you see here. And since it does have a much higher accuracy rate, we have to go deeper into that test data to find the examples of things that went wrong. We'll reshape each image into a flat 784-pixel array which is what our neural network expects as input. We'll call argmax on the resulting classification in one-hot format, and see if that predicted classification matches the actual label for that data. If not, print it out.

You can see here that this model really is doing better. The ones that it's getting wrong are pretty wonky. So in this case, we predicted that this was number nine and if I were to look at that myself, I might guess that was a nine as well. Turns out this person was trying to draw the number four. But you know, this is a case where even a human brain is going to run into trouble as to what this person was actually trying to write. I don't know what that's supposed to be. Apparently, they were trying to draw the number four. Our best guess what number six. Not unreasonable given the shape of things. Here is somebody who was trying to draw two but it looks a whole lot more like a seven. Again, I wouldn't be too sure about that myself. Even though we flatten this data to one dimension, this neural network that we've constructed is already rivaling the human brain in terms of doing handwriting recognition on these numbers. I mean, that's kind of amazing. That one, I probably would've guessed a three on that one. But again, you can see that the quality of this stuff that it has trouble with is really sketchy. What is that? A scorpion? Apparently, that was supposed to be an eight. And our best guess was a one. Some people really can't write! That's a seven? You get the point here. Just by using Keras alone, we've gotten better accuracy. We've achieved a better result because there's less for us to think about.

You can probably improve on this even more. So again, as before, I want you to go back and see if you actually improve on these results. Try using a different optimizer than RMSprop. Try different topologies. And the beauty with Keras is that it's a lot easier to try those different topologies now.

Keras actually comes with an example of using MNIST, and here is the actual topology that they used in their example. Go back and give that a try. See if it's actually any better or not. See if you can improve upon things. One thing you can see here is that they're actually adding dropout layers to prevent overfitting. So it's very easy to add those sorts of features here. Basically, what we've done here is add that same dense layer of 512 hidden neurons. Taking the 784 features, and then we're going to dropout 20% of the neuron's thinnest layer to force the learning to be spread out more and prevent overfitting. It might be interesting to see if that actually improves your results on the test data set by adding those dropout layers. So go play with that, and when we come back we'll do some even more interesting stuff using Keras.



example: multi-class classification

- MNIST is an example of multi-class classification.

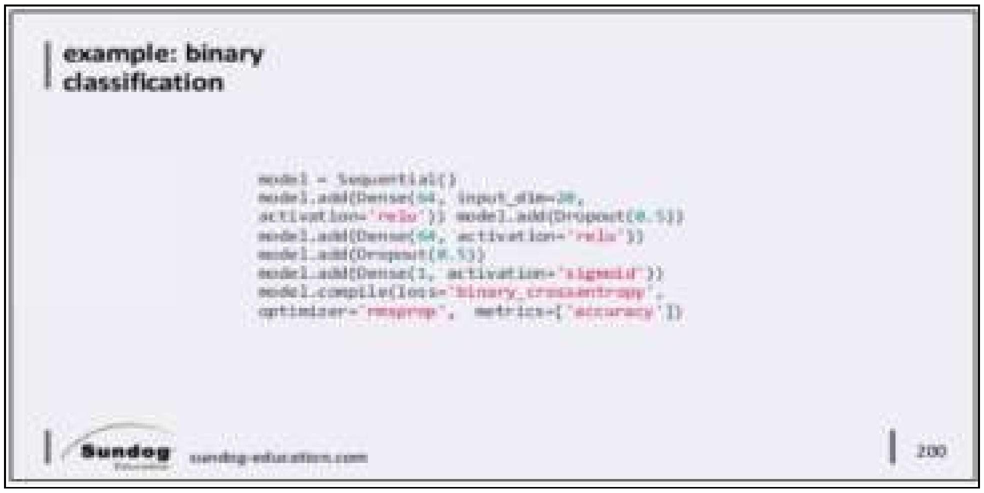
```
model = Sequential()

model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9,
          nesterov=True)
model.compile(loss='categorical_crossentropy',
          optimizer=sgd, metrics=['accuracy'])
```

Sundog sundog-education.com 199

So that was a lot easier using Keras, wasn't it? The MNIST data set is just one type of problem that you might solve with a neural network. It's what we call “multiclass classification”. It’s “multiclass” because the classifications we are fitting into range from the numbers zero through nine. So in this case, we have ten different possible classification values, and that makes this a multiclass classification problem.

Based on Keras's documentation and examples, they have general advice on how to handle different types of problems. Here's an example of how they suggest setting up a multiclass classification problem in general. You can see here that we have two hidden layers. We have an input dimension of however many features you have coming into this system. In this example, there's 20. But depending on the nature of your problem, there may be more or less. It's setting up two Relu activation function layers, each with 64 neurons. Again, that's something that you'd want to tune, depending on the complexity of what you're trying to achieve. It's also sticking in a dropout layer, to discard half of the neurons on each training step. Again, that's to prevent over-fitting. And at the end, it's using a softmax activation to one of ten different output values.

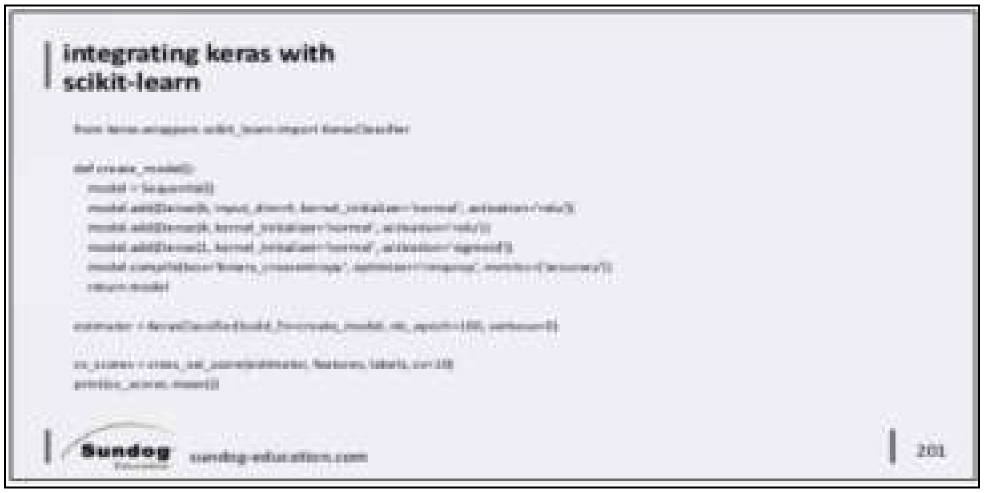
So, this is how they go about solving the MNIST problem within their own documentation. They then use an SGD optimizer and a categorical crossentropy loss function. The Keras documentation is a good resource for some general starting points on where to begin from, at least when you're tackling a specific kind of problem. Again, the actual numbers of neurons, the number of layers, the number of inputs and outputs will vary depending on the problem that you're trying to solve, but they offer general guidelines on what the right loss function is to start with, or what the right optimizer to start with might be.


example: binary classification

```
model = Sequential()
model.add(Dense(64, input_dim=20,
activation='relu')) model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
optimizer='rmsprop',  metrics=['accuracy'])
```

Sundog sundog-education.com 200

Another type of classification problem is binary classification. Maybe you're trying to decide if images of people are pictures of males or females. Maybe you're trying to decide if someone's political party is Democrat or Republican. If you have an either-or sort of problem, that's what we call a binary classification problem. And you can see here, their recommendation is to use a sigmoid activation function at the end instead of softmax. You don't really need the complexity of softmax if you're just trying to go between zero and one. Sigmoid is the activation function of choice in the case of binary classification. They're also recommending the RMSprop optimizer. And the loss function in this case will be binary crossentropy in particular. So, those are a few things that are special about doing binary classification as opposed to multiclass.


integrating keras with scikit-learn

```
from keras.wrappers.scikit_learn import KerasClassifier

def create_model():
    model = Sequential()
    model.add(Dense(6, input_dim=4, kernel_initializer='normal', activation='relu'))
    model.add(Dense(4, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
    return model

estimator = KerasClassifier(build_fn=create_model, nb_epoch=100, verbose=0)

cv_scores = cross_val_score(estimator, features, labels, cv=10)
print(cv_scores.mean())
```

Sundog sundog-education.com 201

Finally I want to talk a little bit more about using Keras with scikit-learn. It makes it a lot easier to do things like cross-validation. Here's a little snippet of code of how that might look.

This small function creates a model that can be used with scikit-learn. This create_model function creates our actual neural network. We're using a sequential model, putting in a dense layer with four inputs and six neurons in that layer. That feeds to another layer of four neurons. And finally, it's going to a binary classifier at the end, with a sigmoid activation function. So we have another little example of setting up a binary classification neural network.

We can then set up an estimator using the Keras classifier function there, and that allows us to get back an estimator that's compatible with scikit learn. So you can see at the end there, we're actually passing that estimator into scikit-learn's cross_val_score function, and that will allow scikit-learn to run your neural network just like it were any other machine learning model built into scikit-learn. That means cross_val_score can automatically train your model and then evaluate its results using k-fold cross-validation, and get you a very meaningful result for how accurate it is, and for its ability to correctly predict classifications for data it's never seen before.

So with those snippets under our belt, let's try out a more interesting example. Let's finally move beyond the MNIST sample. What we'll do is try to predict the political parties of Congressmen just based on their votes in Congress, using the Keras library.

So let's try this out, and play some more with Keras. Make sure you are in the RecSys environment in Anaconda navigator, and we first need to add the “Pandas” package to our RecSys environment. Select “Not Installed,” search for Pandas, and select it and hit apply to install it. We also need to install the scikit_learn package, so follow the same procedure to install it as well.

Then open up your Jupyter notebook, and upload the PoliticsExercise.ipynb file from the DeepLearningIntro folder in your course materials. Once it’s imported, find it in the list and click on it to open it. The house votes data files we need are also included in the course materials .

This example is actually an exercise that I want you to try yourself. I’ll help you load up this data and clean it up, but after that I will leave it up to you to actually implement a neural network with Keras to classify these things. So to recap, we will load up some data about a bunch of congressional votes that various politicians made. And we're going to try to see if we can predict if a politician is Republican or Democrat just based on how they voted on 17 different issues. This is older data from 1984, so you definitely need to be of a certain age to remember what these issues were.

If you're from outside of the United States, just to give you a brief primer in US politics, there are two main political parties in the United States: the Republicans, which tend to be more conservative, and the Democrats which tend to be more progressive. Their stances have changed over the years, but that's the current deal. So let's load up our sample data. I'm going to use the Pandas library that's part of our scientific python environment here to actually load up these CSV files. They're just comma separated value data files. We’ll massage that data, clean it up a little bit, and get it into a form that Keras can accept.

If you’re new to Pandas and want a primer on it, you’ll find a PandasTutorial.ipynb file included in the course materials as well that will walk you through the basics. If you’d like, you can pause now and go through that first.

Let’s start by importing the pandas library; we'll call it pd for short. We need to tell our notebook where the data is stored, so you need to change this next line to indicate where exactly the DeepLearningIntro folder of the course materials is stored on your system. Change this to indicate the file path to your own course materials, which is definitely different from mine .

I've built up this array of column names, because it's not included as a header in this CSV file, so I need to provide that by hand. The columns of the input data are going to be the political party (Republican or Democrat), and then a list of different votes that they voted on. So for example, we can see whether each politician voted yay or nay on religious groups in schools. I'm not really sure what the details of that particular bill were, but by reading these, you can probably guess the direction that the different parties would probably vote toward. So we'll go ahead and read that CSV file, using panda's read_csv function. We will say that any missing values will be populated with a question mark, and we'll pass in a names array of the feature names so we'll know what to call the columns. And we'll just display the resulting data frame, using the head command. Go ahead and hit shift-enter to get that up, and you should see the first five entries of our dataframe.

So for the first five politicians at the head of our data, we can see how each person's party is, and the label that we've assigned to that person, the known label that we're gonna try to predict, and their votes on each issue. Those NaN values represent missing data, where that particular politician did not vote at all for whatever reason – most likely they just weren’t present that day.

Now we can also use the describe() function on the resulting dataframe to get a high level overview of the nature of the data. For example, you can see there's a lot of missing data. Even though there are 435 people that have a party associated with them, only 387 of them actually had a vote on the water project cost sharing bill, for example. So we have to deal with missing data here somehow, and the easiest thing to do is just throw away rows that have missing data. In the real world, you'd want to make sure that you're not introducing some sort of unintentional bias by doing that. Perhaps there's more of a tendency for Republicans to not vote than Democrats, or vice versa. If that were the case, you might be biasing your analysis by throwing out politicians that didn't vote on every actual issue here. But let's assume that there is no such bias, and we can just go ahead and drop those missing values. That's what this little line here does: dropna(inplace=True) . That just means we're going to drop any rows that are missing data from our voting data frame. We'll describe it again and we should see that every column has the same count, because there is no missing data at this point. So we've winnowed things down to 232 politicians here. Not ideal, but hey, that's what we have to work with.

Next thing we need to do is actually massage this data into a form that Keras can consume. So Keras does not deal with Y's and N's, it deals with numbers. So let's replace all the Y's and N's with ones and zeroes, using the replace function. And similarly, we'll replace the strings Democrat and Republican also with the numbers one and zero. So this is turning this into a binary classification problem. If we classify someone as belonging to label one, that will indicate they're a Democrat. And label zero will indicate that they're a Republican. So let's go ahead and run that to clean up that data. And we should see now, if we run head( ) on that data frame again, everything has been converted to numerical data between zero and one, which is exactly what we want for the input to a neural network.

All right, finally let's extract this data into a numpy array we can actually feed to Keras. So to do that, we're just going call “values” on the columns we care about. We're going to extract all of the feature columns into the all_feature s array. And all of the actual labels, the actual parties, into an all_classe s array. So let's go ahead and hit enter to get that in.

And at this point, I'm going to turn it over to you for now. The code snippets you need for binary classification were actually covered in the slides just prior to coming out to this notebook here, so just refer back to that and that should give you the stuff you need to work off of. I want you to try it yourself. My answer is below, but no peeking just yet! Hit pause, and come back later and you can compare your results to mine.

All right, I hope you did your homework here. Let's go ahead and take a look at my implementation. Again, it's pretty much taken from the slides that I showed you earlier for binary classification. All we're going to do is import the stuff we need from Keras here. We're using Dense, Dropout, and Sequential, and we're also going to use cross_val_scor e to actually evaluate our model and illustrate integrating Keras with scikit-learn. When we're integrating with scikit learn, we need to create a function that creates our model so we can pass that into cross_val_score.

We're going to create a Sequential model, and follow the pattern that we showed earlier of doing a binary classification problem. So in this case, we have 16 different issues that people voted on. We're going to use a Relu activation function with a layer of 32 neurons. A pretty common pattern is to start with a large number of neurons in one layer, and winnow things down as you get to higher layers. So we'll distill those 32 neurons down to another layer of 16 neurons.

I'm using the term “units” in this particular example here. As an aside, more and more researchers are using the term “units” instead of “neuron”. You're seeing that in some of the APIs and libraries that are coming out recently. The reason is that we're starting to diverge a little bit between artificial neural networks and how they work, and how the human brain actually works. In some cases, we're actually improving on biology. So, some researchers are taking issue with calling these artificial neurons because we've moved beyond neurons, and our artificial neurons are becoming their own thing at this point .

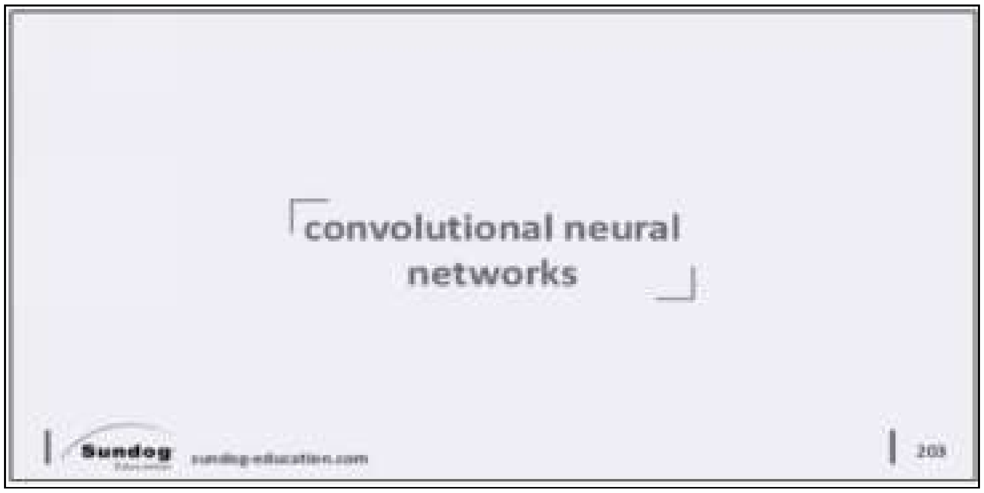
Finally, we'll have one last layer with a single output neuron for our binary classification, with a sigmoid activation function to choose between zero and one. And we'll use the binary cross-entropy function, the Adam optimizer, and kick it off. At that point, we can set up a Keras classifier to actually execute that. And we will create an estimator object from that, which we can then pass into scikit-learn's cross_val_scor e to actually perform K-fold cross-validation automatically. Finally, we will display the mean result when we are done.

So shift-enter, and we'll see how long this takes. Mind you, in 1984, politicians were not as polarized as they are today, so it might be a little harder than it would be today to actually predict someone's parties just based on their votes. It would be very interesting to see if that's the case using more modern data.

After a few minutes, it’s done: 92% accuracy, and that's without even trying too hard. We didn't really spend any time tuning the topology of this network at all, so maybe you can do a better job. If you did get a significantly better result, post that in the course here - I'm sure other students would like to hear about what you did.

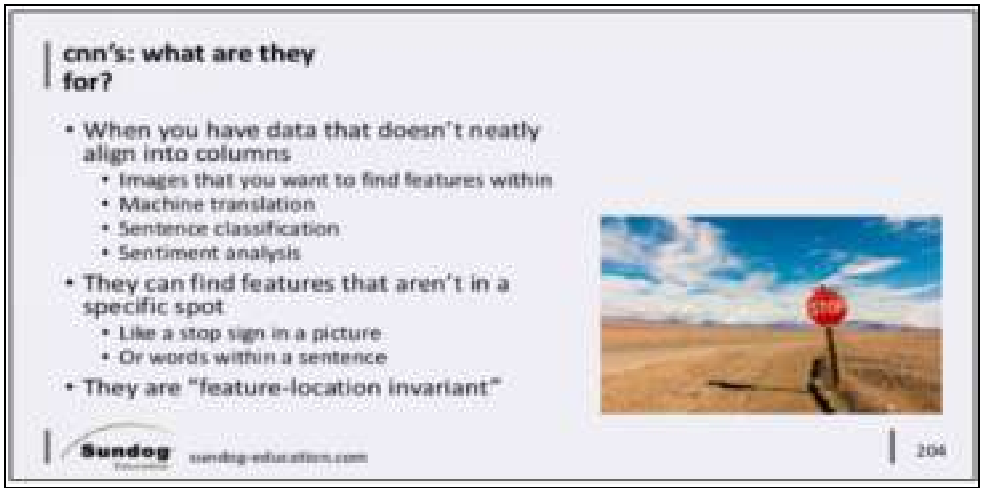
So there you have it: using Keras for a more interesting example with real-world data. We predicted people's political parties using a neural network, and integrated it with scikit-learn to make life even easier. That's the magic of Keras for you !

Convolutional Neural Networks


convolutional neural networks

Sundog sundog-education.com 203

So far, we've seen the power of just using a simple multilayer perceptron to solve a wide variety of problems. But you can kick things up a notch - you can arrange more complicated neural networks together and do more complicated problems with them. Let's start by talking about convolutional neural networks, or CNN's for short.


cnn's: what are they for?

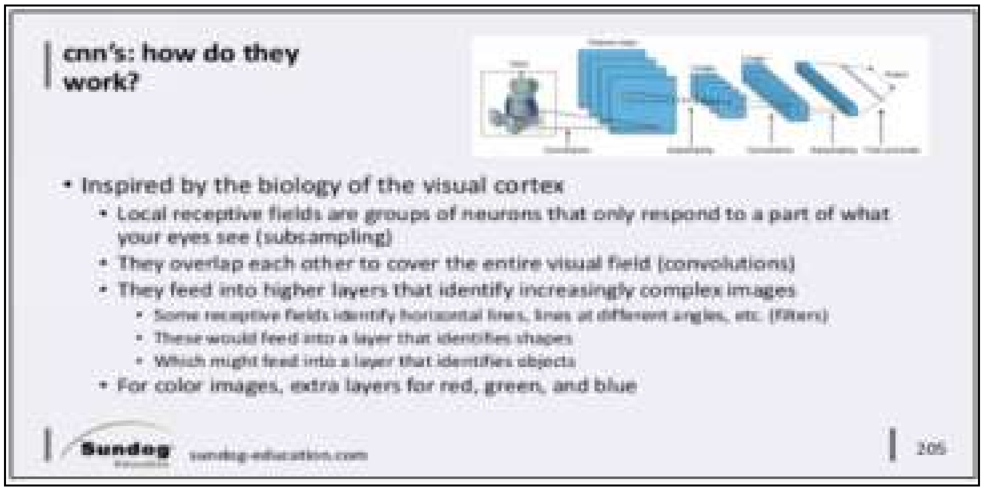
- When you have data that doesn't neatly align into columns
  - Images that you want to find features within
  - Machine translation
  - Sentence classification
  - Sentiment analysis
- They can find features that aren't in a specific spot
  - Like a stop sign in a picture
  - Or words within a sentence
- They are "feature-location invariant"

Sundog sundog-education.com 204

Usually you hear about CNN's in the context of image analysis. Their whole point is to find things in your data that might not be exactly where you expected them to be. Technically we call this "feature location invariant;" that means that if you're looking for some pattern or some feature in your data, but you don't know where exactly it might be in your data, a CNN can scan your data and find those patterns for you wherever they might be.

For example, in this picture here, that STOP sign could be anywhere in the image, and CNN is able to find that STOP sign no matter where it might be. Now, it's not just limited to image analysis, it can also be used for any sort of problem where you don't know where the features you have might be located within your data. Machine translation or natural language processing tests come to mind for that; you don't necessarily know where the noun or the verb or phrase you care about might be in some paragraph or sentence you're analyzing, but CNN can find it and pick it out for you.

Sentiment analysis is another application of CNN, so you might not know exactly where a phrase might be that indicates some happy sentiment or some frustrated sentiment or what whatever you might be looking for, but CNN can scan your data and pluck it out. And you'll see that the idea behind it isn't really as complicated as it sounds. This is another example of using fancy words to make things seem more complicated than they really are.



cnn's: how do they work?

- Inspired by the biology of the visual cortex
  - Local receptive fields are groups of neurons that only respond to a part of what your eyes see (subsampling)
  - They overlap each other to cover the entire visual field (convolutions)
  - They feed into higher layers that identify increasingly complex images
    - Some receptive fields identify horizontal lines, lines at different angles, etc. (filters)
    - These would feed into a layer that identifies shapes
    - Which might feed into a layer that identifies objects
  - For color images, extra layers for red, green, and blue

Sundog sundog-education.com 205

So how do they work? Well, CNN's (convolutional neural networks), are inspired by the biology of your visual cortex. It takes cues from how your brain actually processes images from your retina and it's another fascinating example of emergent behavior.

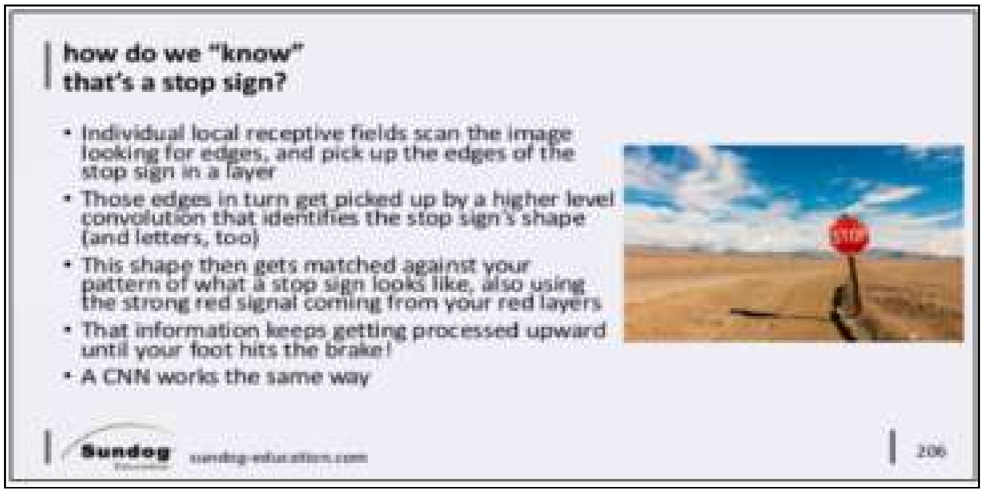
The way your eyes work is that individual groups of neurons service a specific part of your field of vision. We call these "local receptive fields." They are just groups of neurons responding to a part of what your eyes see. They sub-sample the image coming in from your retinas, and they have specialized groups of neurons for processing specific parts of the field of view that you see with your eyes.

Now, these little areas from each local receptive field overlap each other to cover your entire visual field, and this is called "convolution." Convolution is just a fancy word of saying "I'm going to break up this data into little chunks and process those chunks individually," and then the system assembles a bigger picture of what you're seeing higher up in the chain .

The way it works within your brain is that you have many layers, just like a deep neural network, that identifies various complexities of features, if you will. So the first layer that you go into from your convolutional neural network inside your head might just identify horizontal lines, or lines at different angles, or specific kinds of edges. We call these "filters," and they feed into a layer above them that would then assemble those lines that it identified at the lower level into shapes. Maybe there's a layer above that that would be able to recognize objects based on the patterns of shapes that you see. So we have this hierarchy that detects lines and edges, then shapes from the lines, and then objects from the shapes.

If you're dealing with color images, we have to multiply everything by 3 because you actually have specialized cells within your retina for detecting red, green and blue light. Those are processed individually and assembled later.

So that's all a CNN is - it is just taking a source image, or source data of any sort really, breaking it up into little chunks called "convolutions," and then we assemble those and look for patterns at increasingly higher complexities at higher levels in your neural network.



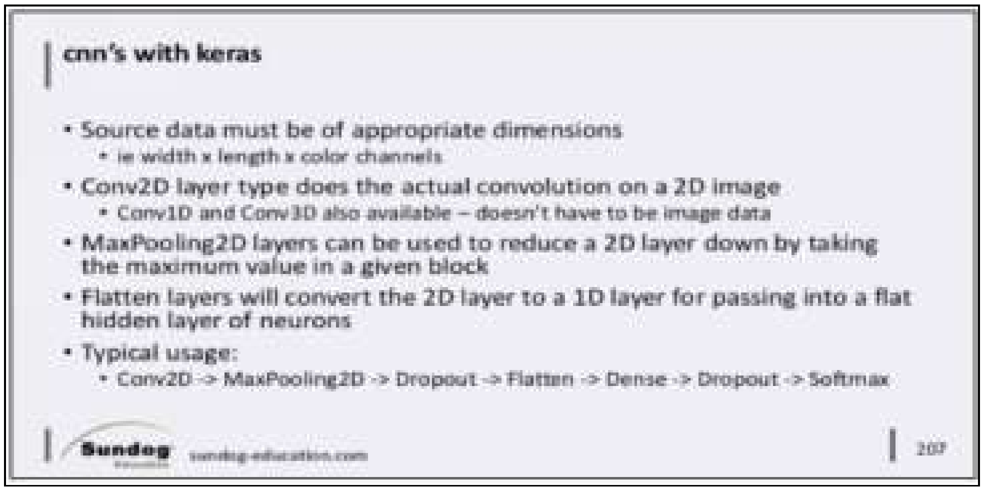
how do we "know" that's a stop sign?

- Individual local receptive fields scan the image looking for edges, and pick up the edges of the stop sign in a layer
- Those edges in turn get picked up by a higher level convolution that identifies the stop sign's shape (and letters, too)
- This shape then gets matched against your pattern of what a stop sign looks like, also using the strong red signal coming from your red layers
- That information keeps getting processed upward until your foot hits the brake!
- A CNN works the same way

Sundog sundog-education.com 206

So how does your brain know that you're looking at a STOP sign there? Let's talk about this in more colloquial language. Like we said, you have individual local receptive fields that are responsible for processing specific parts of what you see, and those local receptive fields are scanning your image and they overlap with each other looking for edges. You might notice that your brain is very sensitive to contrast and edges that it sees in the world - those tend to catch your attention, right? That's why the letters on this slide catch your attention, because there's high contrast between the letters and the white background behind them. So at a very low level, you're picking up the edges of that STOP sign and the edges of the letters on the STOP sign. Now, a higher level might take those edges and recognize the shape of that STOP sign. That layer says: "Oh! There's an octagon there, that means something special to me," or "those letters form the word STOP, that mean something special to me, too," and ultimately that will get matched against whatever classification pattern your brain has of a STOP sign.

So no matter which receptive field picked up that STOP sign, at some layer it will be recognized as a STOP sign and furthermore, because you're processing data in color, you can also use the information that the STOP sign is red and further use that to aid in the classification of what this object really is. So, somewhere in your head there's a neural network that says: "hey! if I see edges arranged in an octagon pattern that has a lot of red in it and says stop in the middle, that means I should probably hit the brakes on my car." At some even higher level where your brain is doing higher reasoning, that's what happened. There's a pattern that says: "hey, there's a STOP sign coming up here, I better hit the brakes in my car," and if you've been driving long enough, you don't even really think about it anymore, do you? It feels like it’s hardwired, and that literally may be the case. Anyway, a convolutional neural network and an artificial convolutional neural network works the same way; it’s the same exact idea.



cnn’s with keras

- Source data must be of appropriate dimensions
  - ie width x length x color channels
- Conv2D layer type does the actual convolution on a 2D image
  - Conv1D and Conv3D also available – doesn’t have to be image data
- MaxPooling2D layers can be used to reduce a 2D layer down by taking the maximum value in a given block
- Flatten layers will convert the 2D layer to a 1D layer for passing into a flat hidden layer of neurons
- Typical usage:
  - Conv2D -> MaxPooling2D -> Dropout -> Flatten -> Dense -> Dropout -> Softmax

Sundog sundog-education.com 207

So how do you build a CNN with Keras? Obviously you probably don't want to do this at the low level TensorFlow layer, even though you can! But CNN's get pretty complicated. A higher level API like Keras becomes essential.

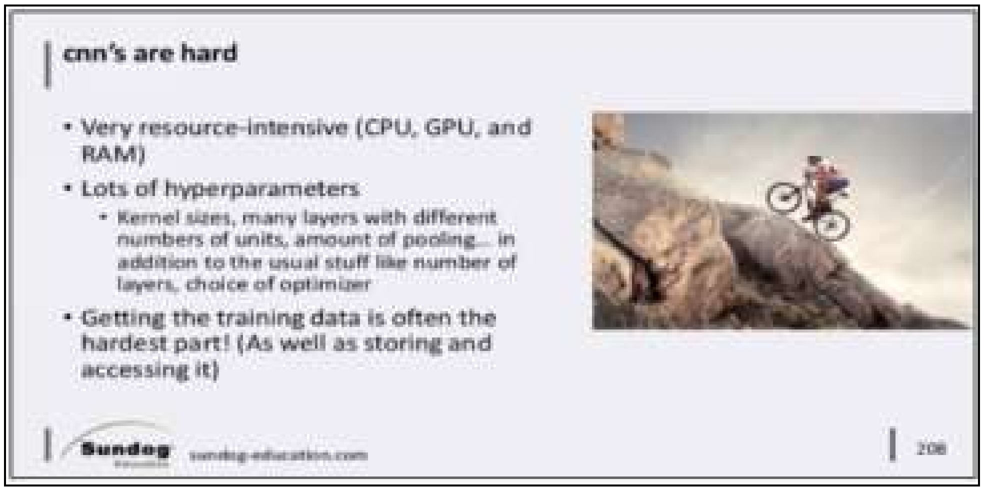
First of all, you need to make sure your source data is of the appropriate dimensions and appropriate shape. You are going to be preserving the actual 2D structure of an image if you're dealing with image data here, so the shape of your data might be the width times the length times the number of color channels. If it's a black and white image there's only one color channel that indicates some gray-scale value between black and white at every point in the image; we can do that with a single value at each point. But if it's a color image, you'd have three color channels, one for red, one for green and one for blue, because you can create any color by combining red, green, and blue together.

There are some specialized types of layers in Keras that you use when you're dealing with convolutional neural networks. For example, there's the Conv2D layer type that does the actual convolution on a 2D image, and again, convolution is just breaking up that image into little subfields that overlap each other for individual processing. There's also a Conv1D and a Conv3D layer available as well. You don't have to use CNN's with images, like we said, it can also be used with text data, for example, that might be an example of one dimensional data. The Conv3D layer is available as well if you're dealing with 3D volumetric data of some sort, so there are a lot of possibilities there. Another specialized layer in Keras for CNN is MaxPooling2D, and there's a 1D and 3D variant of that as well. The idea of that is just to reduce the size of your data down. It just takes the maximum value seen in a given block of an image and reduces it to a layer down to those maximum values. It’s just a way of shrinking the images in such a way that it can reduce the processing load on the CNN.

As you see, processing a CNN is very computing intensive, and the more you can do to reduce the work you have to do, the better. So if you have more data in your image then you need, a MaxPooling2D layer can be useful for distilling that down to the bare essence of what you need to analyze .

Finally, at some point you need to feed this data into a flat layer of neurons, right? At some point it's just going to go into a perceptron, and at this stage we need to flatten that 2D layer into a 1D layer, so we can just pass it into a layer of neurons. From that point it just looks like any other multilayer perceptron. So the magic of CNN's really happens at a lower level. It ultimately gets converted into what looks like the same types of multilayer perceptrons that we've been using before. The magic happens in actually processing your data, convolving it and reducing it down to something that's manageable.

So typical usage of image processing with a CNN would look like this: you might start with a Conv2D layer that does the actual convolution of your image data. Tou might follow that up with a MaxPooling2D layer on top of that that distills that image down, just shrinking the amount of data that you have to deal with. You might then do a Dropout layer on top of that, which just prevents overfitting like we talked about before, and at that point you might apply a Flatten layer to actually be able to feed that data into a perceptron. That's where a Dense layer might come into play. A dense layer in Keras is just a perceptron really, you know, it's a hidden layer of neurons. From there we might do another Dropout pass to further prevent overfitting and finally do a softmax to choose the final classification that comes out of your neural network.



cnn's are hard

- Very resource-intensive (CPU, GPU, and RAM)
- Lots of hyperparameters
  - Kernel sizes, many layers with different numbers of units, amount of pooling… in addition to the usual stuff like number of layers, choice of optimizer
- Getting the training data is often the hardest part! (As well as storing and accessing it)

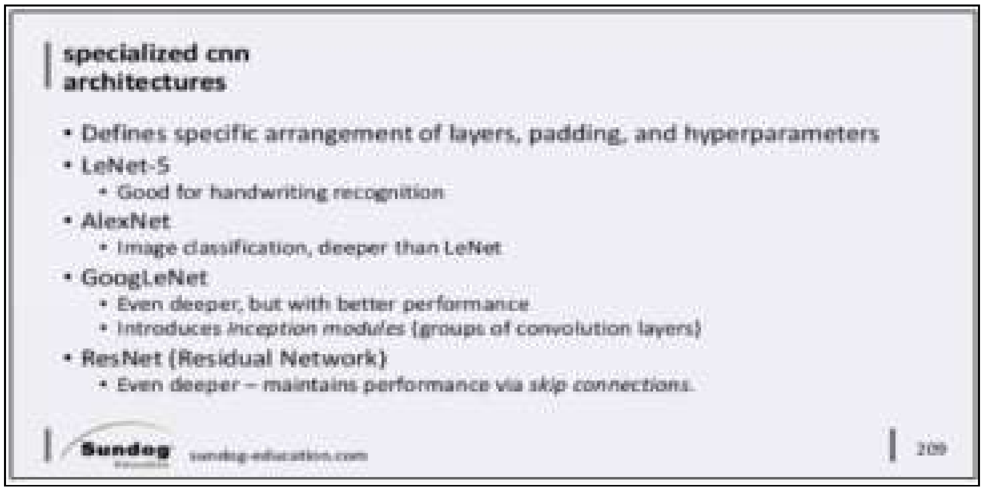
Sundog sundog-education.com 208

As I said, CNN's are very computationally intensive. They are very heavy on your CPU, your GPU and your memory requirements. Shuffling all that data around and convolving it adds up really, really fast. Beyond that, there's a lot of what we call "hyperparameters," a lot of different knobs and dials that you can adjust on CNN's.

So in addition to the usual stuff, you can tune the topology of your neural network, or what optimizer you use, or what loss function you use, or what activation function you use. There are also choices to make about the kernel sizes - that is the area that you actually convolve across - how many layers do you have, how many units do you have, and how much pooling do you do when you're reducing the image down.

There's a lot of variance here; there are almost an infinite amount of possibilities for configuring a CNN.

But often just obtaining the data to train your CNN with is the hardest part. So, for example, if you own a Tesla, that's actually taking pictures of the world around you and the road around you, and all the street signs and traffic lights as you drive. Every night it sends all those images back to some data server somewhere, so Tesla can actually run training on its own neural networks based on that data. So if you slam on the brakes while you're driving a Tesla, at night, that information is going to be fed into a big data center somewhere and Tesla is going to crunch on that and say: "hey, is there a pattern here to be learned from what I saw from the cameras from the car? That means you should slam on the brakes." When you think about the scope of that problem, just the sheer magnitude of processing and obtaining and analyzing all that data, that becomes very challenging in and of itself.

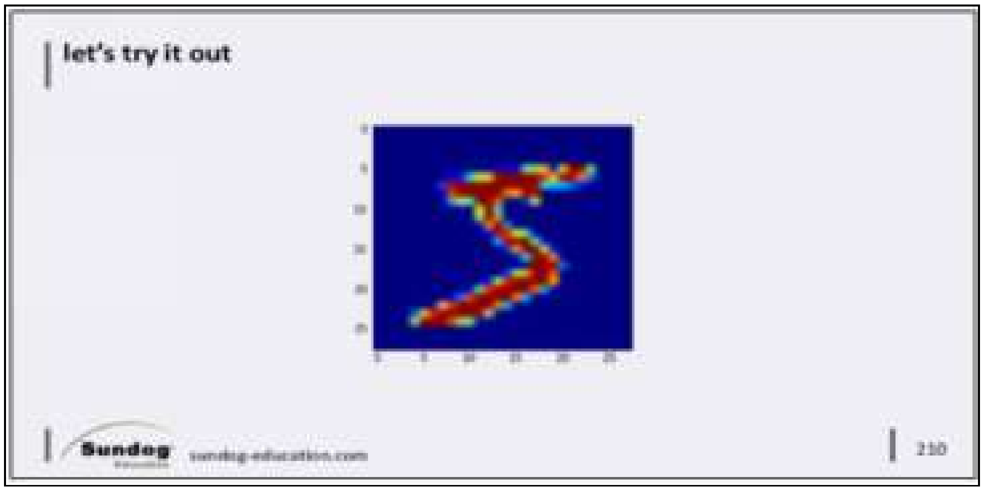


specialized cnn architectures

- Defines specific arrangement of layers, padding, and hyperparameters
- LeNet-5
  - Good for handwriting recognition
- AlexNet
  - Image classification, deeper than LeNet
- GoogLeNet
  - Even deeper, but with better performance
  - Introduces *inception modules* (groups of convolution layers)
- ResNet (Residual Network)
  - Even deeper – maintains performance via *skip connections.*

Sundog sundog-education.com 209

Now, fortunately the problem of tuning the parameters doesn't have to be as hard as I described it to be. There are specialized architectures of convolutional neural networks that do some of that work for you. A lot of research goes into trying to find the optimal topologies and parameters for a CNN for a given type of problem, and you can just think of it like a library you can draw from. So, for example, there's the LeNet-5 architecture that you can use, that's suitable for handwriting recognition in particular. There's also one called AlexNet, which is appropriate for image classification. It's a deeper neural network than LeNet. In the example we talked about on the previous slides, we only had a single hidden layer, but you can have as many as you want. It’s just a matter of how much computational power you have available. There's also something called GoogleLeNet (you can probably guess who came up with that), it's even deeper, but it has better performance because it introduces this concept called "inception modules." Inception modules group convolution layers together, and that's a useful optimization for how it all works. Finally, the most sophisticated one today is called ResNet - that stands for Residual Network. It's an even deeper neural network, but it maintains performance by what's called "skip connections." It has special connections between the layers of the perceptron to further accelerate things, so it builds upon the fundamental architecture of a neural network to optimize its performance. And as you'll see, CNN's can be very demanding on performance.


let’s try it out

Sundog sundog-education.com 210

So with that, let's give it a shot! Let's actually use a CNN and see if we can do a better job at image classification than we've done before using one .

Let’s actually run a Convolutional Neural Network using Keras. As before, make sure you have your RecSys environment selected in Anaconda Navigator, then launch the Jupyter Notebook if you haven’t already. Upload the Keras-CNN notebook file in your course materials’ DeepLearningIntro folder, and then click on it once it has been imported to open it up.

We're going to revisit the MNIST handwriting recognition problem where we try to classify a bunch of images of people drawing the numbers zero through nine, and see if we can do a better job of it using CNNs. Again, CNNs are better suited to image data in general, especially if you don't know exactly where the feature you're looking for is within your image, so we should expect to get better results here.

Let’s start by importing all the stuff we need from Keras: we’ll import the MNIST data set that we're playing with, the Sequential model so we can assemble our neural network, and then we’ll import all these different layer types that we talked about in the slides: the Dense, Dropout, Conv2D, MaxPooling2D, and Flatten layer types, and in this example we'll use the RMSprop optimizer. Go ahead and kick that off. The rest of the code for loading up the training and test data is going to look just like it did before.

We're going to shape this data a little bit differently. So since Convolutional Neural Networks can process 2D data in all their 2D glory, we're not going to reshape that data into flat 1D arrays of 768 pixels. Instead we're going to shape it into the width times the length times the number of color channels. So in this case, our data is grayscale in nature, so there is only a single color channel that just defines how light or dark the image is at a specific pixel. There are a couple of different ways that data can be stored, and we need to handle both cases. It might be organized as color channels by width times length, or it might be width times length times color channels. That’s what this section of code does that checks the “channels_first” flag on the data. We check if it's the “channels first” format or not, and reshape the data accordingly. We store the resulting shape in this thing called “input_shape”. That's the shape of our input testing data and training data.

As before, we are going to scale this data down. It comes in as 8-bit data that ranges from 0-255. We need to convert that into normalized floating point data instead, so we'll convert that data to floating point 32 bit values and then divide each pixel by 255 to transform that into some number between 0 and 1. Go ahead and hit shift enter there to kick that off.

As before, we will convert the label data into one-hot categorical format because that will match up nicely with the output of our neural network.

We'll again do a sanity check to make sure that we successfully imported our data. So we'll pick a random training sample to print out here and display it. Run this block, and you'll see the one-hot format of the "three" label. Zero, one, two, three, that's correct. Our code came up with a human-readable label of 3, and sure enough, that looks like the number three. So, it looks like our data's in good shape for processing.

Now let's actually set up a CNN and see how that works. Let's walk through what's going on in this next code block here. As before, we start off by setting up a Sequential model that just allows us to very easily build up layers to build up our neural network. We will start with a Conv2D layer. So what this syntax means is that our convolutional 2D layer is going to have 32 windows, or 32 regional fields if you will, that it will use to sample that image with. And each one of those samples will be of a three by three kernel size. It also needs to know the shape of your input data, which we stored previously, that's 1x28x28, or 28x28x1, depending on the input format. We then add a second convolutional filter on top of that to identify higher level features. This one will have 64 kernels also of three by three size, and we are going to use a Relu activation function on that. So we built up two convolution layers here.

Again, you want to just reuse any previous research you can for a given problem. There are so many ways to configure CNNs that if you start from scratch you're going to have a very hard time to tune it, especially when we consider how long it takes to iterate between each run. These are very resource intensive, so I have just taken this from the CNN example that comes with the Keras library, and drawn my initial topology from it.

So now we've done our convolution layers, and we’ll next do a MaxPooling2D step, to actually reduce that data a little bit. This takes a 2x2 pool size, so for each 2x2 pixel block at this stage we’ll reduce that down to a single pixel that represents the maximum pixel found within that pool. Note that the pool size can be different from the underlying kernel size from the convolution you did. This is just a technique for shrinking your data down into something that's more manageable.

At this point we'll do a Dropout pass to prevent overfitting, we will then flatten what we have so far. That will take our 2D data and flatten it to a 1D layer, and from this point it's just going to look like any other multilayer perceptron, just like we used before. So all the magic of CNNs has happened at this point, and now we're just going to convert it down to a flat layer that we input into a hidden layer of neurons. In this case, we have 128 units in that layer, again with the Relu activation function. We'll do one more Dropout pass to prevent overfitting, and finally choose our final categorization of the numbers zero through nine by building one final output layer of ten neurons with the softmax activation function on it .

Alright, so let's go ahead and let that run. Again nothing's really happening until we actually kick off the model, so that doesn't take any time at all. We can do a model.summary( ) just to double check that everything is the way that we intended it to be. And you can see that we have our two convolution layers here, followed by a pooling layer, followed by a dropout, a flatten, and from there we have a dense, dropout, and dense multilayer perceptron to actually do our final classification.

Finally we need to compile that model with a specific optimizer and loss function. In this case, we're going to use the Adam optimizer and categorical cross entropy because that's the appropriate loss function for a multiple category classification problem. And finally, we will actually run it. Now, like I said, CNNs are very expensive to run, so let me talk about what this command does first of all. Nothing unusual here, just says that we're going to run batches of 32, which is smaller than before, because there is a much higher computational cost to this. We're only going to run 10 epochs this time because, again, it takes a long time. More would be better, but you don’t want to tie up your computer for a whole day just for this activity. The verbosity is set to 2, which is the best choice for running within a notebook, and we will pass in our validation test data for it to work with as well.

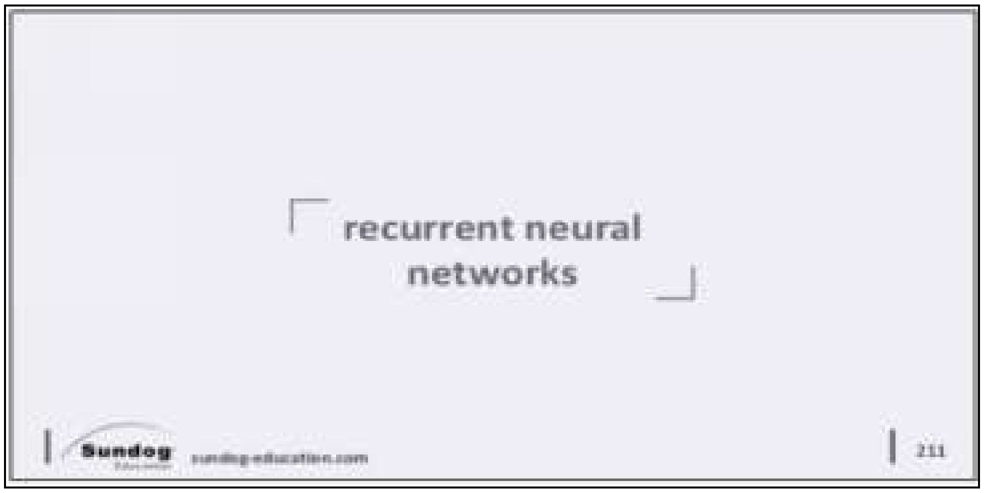
Now, I am not going to actually run this within this lecture, because it could actually take about an hour to run. If you don't have a beefy machine, it might not finish at all if you don't have enough RAM or enough CPU power - this might even be too much for one system. So I’ll skip ahead here, as I actually ran this earlier myself. You can see that it very quickly converges to a very good accuracy value and it was still increasing, so there probably would have been improvements to be had going beyond 10 iterations of the training here. But even after just 10 epochs, or 10 iterations, we ended up with an accuracy of over 99% .

We can actually evaluate that based on our test data and recreate that 99% accuracy, so that's pretty awesome. So CNNs are definitely worth doing if the accuracy is key, and for applications where lives are at stake, such as a self-driving car, that's obviously worth the effort! You want complete accuracy of detecting if there is a stop sign in front of you. Even 0.1% error would be unacceptable in a situation like that.

That's the power of CNNs. They are more complicated and they take a lot more time to run, but as we said the power of Tensorflow, which Keras is running on top of, means you can distribute this work across an entire cloud of computers, and an entire array of GPUs that are on each computer. So there are ways to accelerate this; we're just not taking advantage of that in this little example here. It's just illustrative. So there you have it, your first Convolutional Neural Network. You can see how powerful it is for successfully doing image classification among other things.

Let's move on to another interesting type of neural network next.

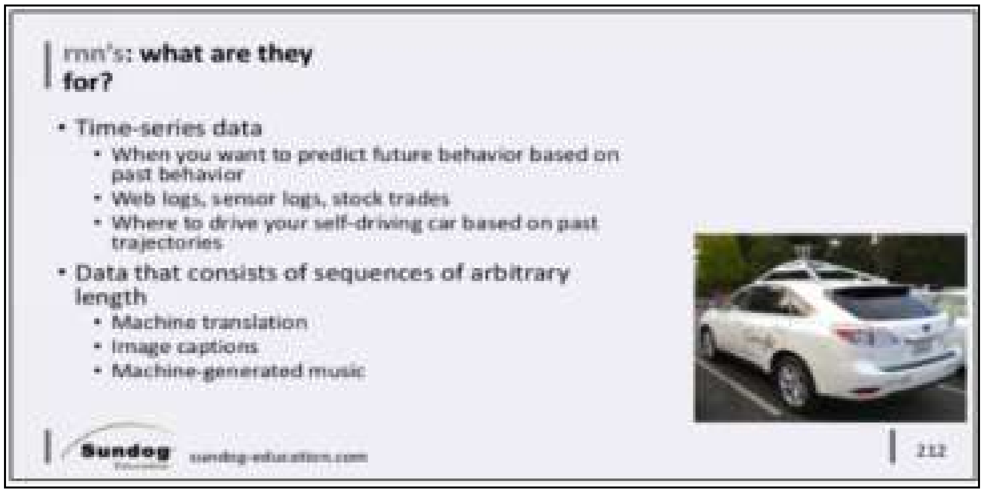
Recurrent Neural Networks



recurrent neural networks

Sundog sundog-education.com 211

Let's talk about another kind of neural network, the Recurrent Neural Network.



rnn's: what are they for?

- Time-series data
  - When you want to predict future behavior based on past behavior
  - Web logs, sensor logs, stock trades
  - Where to drive your self-driving car based on past trajectories
- Data that consists of sequences of arbitrary length
  - Machine translation
  - Image captions
  - Machine-generated music

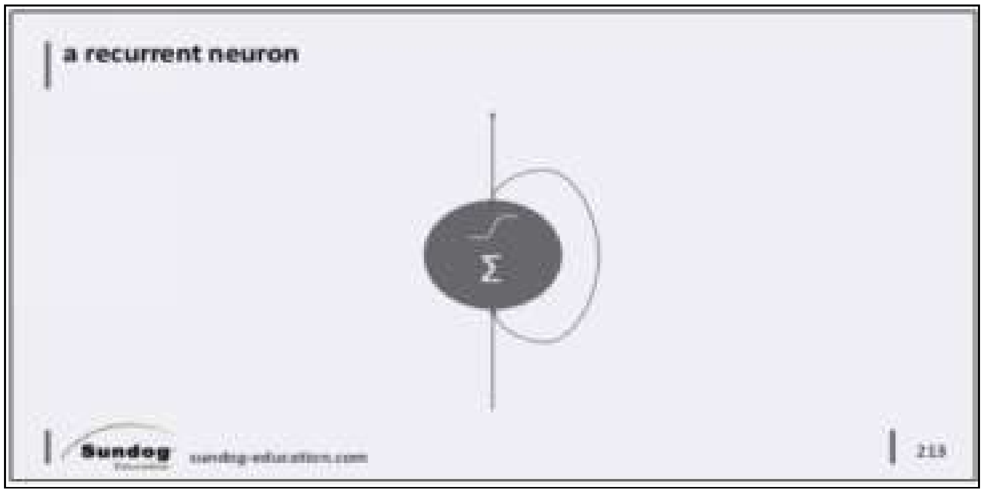
Sundog sundog-education.com 212

What’s a RNN for? Well a couple of things. Basically they're for sequences of data. That might be a sequence in time, so, you might use it for processing time series data, where you’re trying to look at a sequence of data points over time and predict the future behavior of something over time in turn. RNN's are fundamentally for sequential data of some sort. Some examples of time series data might be web logs where you are receiving different hits to your website over time, or sensor logs where you're getting different inputs from sensors from the Internet of Things, or maybe you're trying to predict stock behavior by looking at historical stock trading information. These are all potential applications for recurrent neural networks because they can take a look at the behavior over time, and try to take that behavior into account when it makes future projections.

Another example might be if you're trying to develop a self-driving car. You might have a history of where your car has been, its past trajectories, and maybe that can inform how your car might want to turn in the future. So, you might take into account the fact that your car has been turning along a curve to predict that perhaps it should continue to drive along a curve until the road straightens out.

The sequence doesn't have to just be in time, it can be any kind of sequence of arbitrary length. Something else that comes to mind are languages. Sentences are just sequences of words, right? So you can also apply RNN's to language, or machine translation, or producing captions for videos or images. These are examples of where the order of words in a sentence might matter, and the structure of the sentence and how these words are put together could convey more meaning than you could get by just looking at those words individually without context. So again, an RNN can make use of that ordering of the words and try to use that as part of its model.

Another interesting application of RNN's is machine generated music. You can also think of music sort of like text, where instead of a sequence of words or letters, you have a sequence of musical notes. You can actually build a neural network that can take an existing piece of music, and extend upon it by using a recurrent neural network to try to learn the patterns that were aesthetically pleasing from the music in the past.

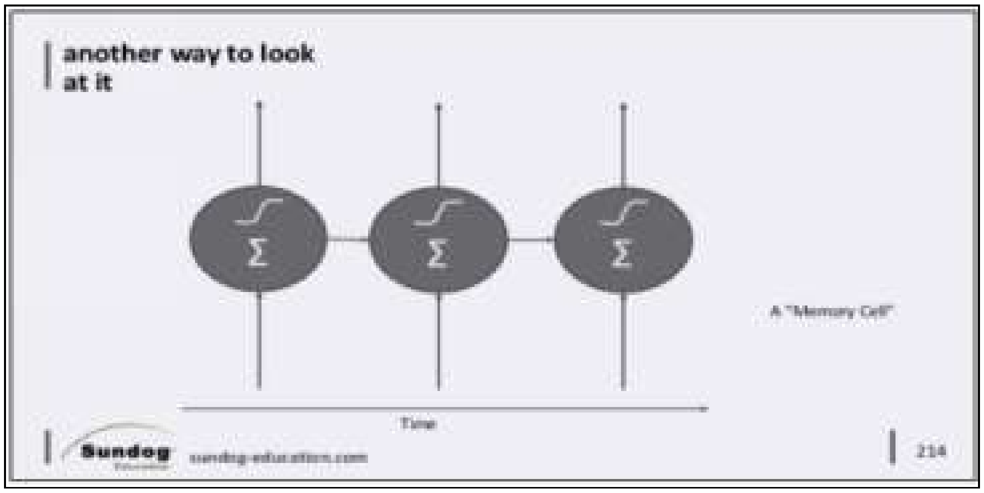


a recurrent neuron

Sundog sundog-education.com 213

Conceptually, this is what a single recurrent neuron looks like in terms of a model. It looks a lot like an artificial neuron that we've looked at before; the big difference is this little loop here going around it. As we run a training step on this neuron, some training data gets fed into it - or maybe this is an input from a previous layer in our neural network. It will apply some sort of step function after summing all the inputs into it. In this case, we're going to be using something more like a hyperbolic tangent, because mathematically we want to make sure that we preserve some of the information over time in a smooth manner.

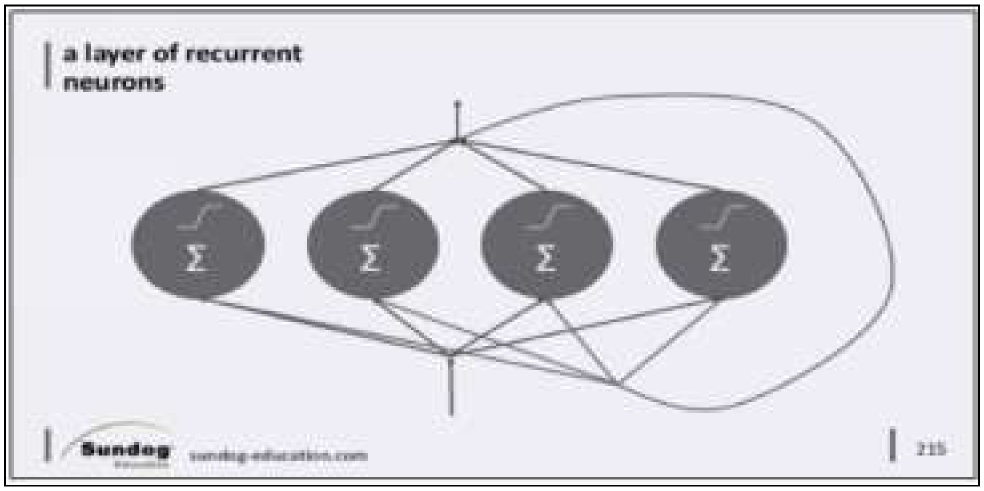
Now, usually we would just output the result of that summation and that activation function as the output of this neuron, but we're also going to feed that back into the same neuron, so the next time we run some data through this neuron, that data from the previous run also gets summed in to the results as an extra input. So as we keep running this thing over and over again, we'll have some new data coming in that gets blended together with the output from the previous run through this neuron, and it just keeps happening over and over and over again. So you can see that over time the past behavior of this neuron influences its future behavior, and it influences how it learns.



Another way of thinking about this is by unrolling it in time. So what this diagram shows is the same single neuron just at three different time steps. When you start to dig into the mathematics of how RNN's work, this is a more useful way of thinking about it.

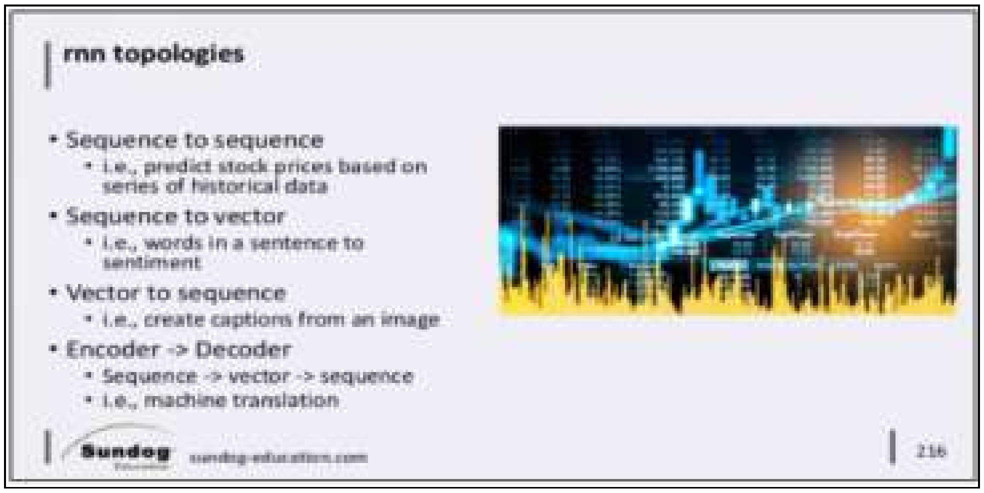
So if we consider this to be time step 0 at the left, you can see there's some sort of data input coming into this recurrent neuron and that will produce some sort of output after going through its activation function. That output also gets fed into the next time step, so if this is time step one in the middle with this same neuron, you can see that this neuron is receiving not only a new input, but also the output from the previous time step and those get summed together. The activation function gets applied to it, and that gets output as well. The output of that combination then gets fed onto the next time step, call this time step 2, where a new input for time step 2 gets fed into this neuron and the output from the previous step also gets fed in, they get summed together, the activation function is run, and we have a new output.

This is called a memory cell because it does maintain memory of its previous outputs over time, and you can see that even though it's getting summed together at each time step, over time those earlier behaviors kind of get diluted, Right? So we're adding in time step 0 to time step 1, and then the sum of those two things end up working into time step 2. So one property of memory cells is that more recent behavior tends to have more of an influence on the current time step. This can be a problem in some applications, but there are ways to work against that that we can talk about later.



Stepping this up, you can have a layer of recurrent neurons. In this diagram, we are looking at four individual recurrent neurons that are working together as part of a layer. You can have some input going into this layer as a whole that gets fed into these four different recurring neurons, and then the output of those neurons can then get fed back to the next step to every neuron in that layer .

So all we're doing is scaling this out horizontally, so instead of a single recurrent neuron we have a layer of four recurrent neurons where all of the output of those neurons is feeding in to the behavior of those neurons in the next learning step. You can scale this out to have more than one neuron and learn more complicated patterns as a result. RNN's open up a wide range of possibilities, because now we have the ability to deal not just with vectors of information or static snapshots of some sort of a state, we can also deal with sequences of data as well.



rnn topologies

- Sequence to sequence
  - i.e., predict stock prices based on series of historical data
- Sequence to vector
  - i.e., words in a sentence to sentiment
- Vector to sequence
  - i.e., create captions from an image
- Encoder -> Decoder
  - Sequence -> vector -> sequence
  - i.e., machine translation

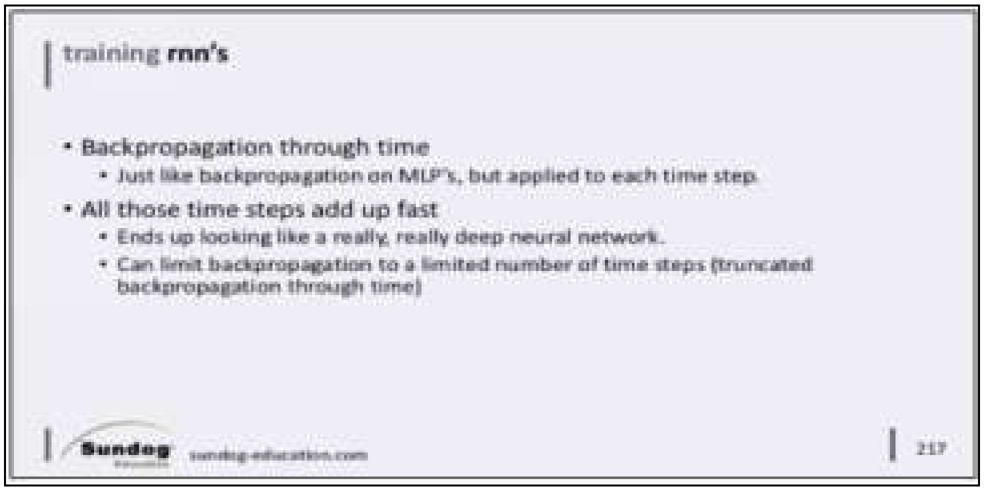
Sundog sundog-education.com 216

There are four different combinations here that an RNN can deal with. We can deal with "sequence to sequence" neural networks. If our input is a time series, or some sort of sequence of data, we can also have an output that is a time series, or some sequence of data as well. So if you're trying to predict stock prices in the future based on historical trades, that might be an example of sequence to sequence topology.

We can also mix and match sequences with the older vector static states that we predicted just using multilayer perceptrons. We would call that “sequence to vector.” If we were starting with a sequence of data, we could produce just a snapshot of some state as a result of analyzing that sequence. An example might be looking at the sequence of words in a sentence to produce some idea of the sentiment that sentence conveys, and we'll actually look at that in an example shortly.

You can go the other way around too - you can go from a vector to a sequence. An example of that would be taking an image which is a static vector of information, and then producing a sequence from that vector. For example, words in a sentence creating a caption from an image.

We can chain these things together in interesting ways as well. We can have encoders and decoders built up that feed into each other. For example, we might start with a sequence of information from a sentence of some language, embody what that sentence means as some sort of a vector representation, and then turn that around into a new sequence of words in some other language. That might be how a machine translation system could work. You might start with a sequence of words in French, build up a vector that embodies the meaning of that sentence, and then produce a new sequence of words in English or whatever language you want. That's an example of using a recurrent neural network for machine translation. There are lots of exciting possibilities here.



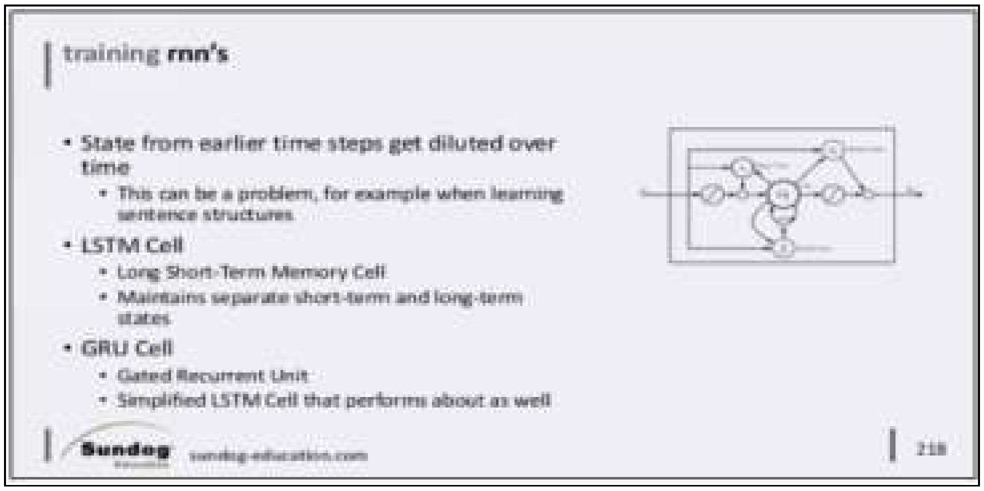
training **rnn's**

- Backpropagation through time
  - Just like backpropagation on MLP's, but applied to each time step.
- All those time steps add up fast
  - Ends up looking like a really, really deep neural network.
  - Can limit backpropagation to a limited number of time steps (truncated backpropagation through time)

Sundog sundog-education.com 217

Training RNN's, just like CNN's, it's hard, and in some ways it's even harder. The main twist here is that we need to back-propagate not only through the neural network itself and all of its layers, but also through time. From a practical standpoint, every one of those time steps ends up looking like another layer in our neural network while we're trying to train it, and those times steps can add up fast. Over time we end up with a deeper and deeper neural network that we need to train, and the cost of actually performing gradient descent on that increasingly deep neural network becomes increasingly large.

So to put an upper cap on that training time, often we limit the backpropagation to a limited number of time steps. We call this "truncated backpropagation through time." It’s something to keep in mind when you're training an RNN - you not only need to backpropagate through the neural network topology that you've created, you also need to backpropagate through all the time steps that you've built up to that point.



training rnn's

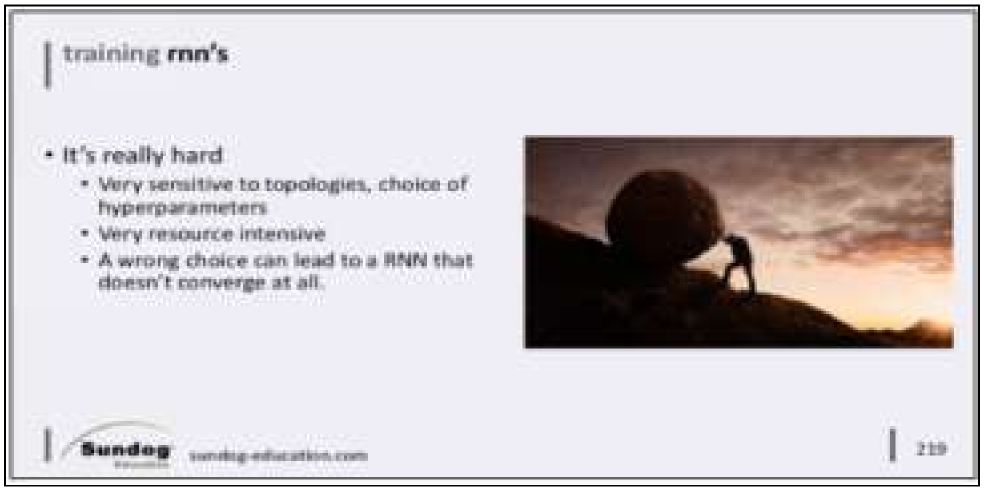
- State from earlier time steps get diluted over time
  - This can be a problem, for example when learning sentence structures
- LSTM Cell
  - Long Short-Term Memory Cell
  - Maintains separate short-term and long-term states
- GRU Cell
  - Gated Recurrent Unit
  - Simplified LSTM Cell that performs about as well

Sundog sundog-education.com 218

Now, we talked earlier about the fact that as you're building up an RNN, the state from earlier time steps end up getting diluted over time because we just keep feeding in behavior from the previous step in our run to the current step. This can be a problem if you have a system where older behavior does not matter less than newer behavior. For example, if you're looking at words in a sentence, the words at the beginning of a sentence might even be more important than words toward the end. If you're trying to learn the meaning of a sentence, there is no inherent relationship between where each word is and how important it might be in many cases. That's an example of where you might want to do something to counteract that effect.

One way to do that is something called the LSTM Cell. It stands for "Long Short-Term Memory cell." The idea here is that it maintains separate ideas of both short-term and long-term states, and it does this in a fairly complex way. Now, fortunately you don't really need to understand the nitty-gritty details of how it works. There is an image of it here for you to look at if you're curious, but the libraries that you use will implement this for you. The important thing to understand is that if you're dealing with a sequence of data where you don't want to give preferential treatment to more recent data, you probably want to use an LSTM Cell instead of just using a straight-up RNN.

There's also an optimization on top of LSTM cell called GRU cells. That stands for "Gated Recurrent Unit." It's just a simplification on LSTM Cells that performs almost as well, so if you need to strike a compromise between performance in the terms of how well your model works, and performance in terms of how long it takes to train it, a GRU Cell might be a good choice. GRU cells are very popular in practice, and later we’ll see how they can be used for certain types of recommender systems as well.



training **rnn's**

- It's really hard
  - Very sensitive to topologies, choice of hyperparameters
  - Very resource intensive
  - A wrong choice can lead to a RNN that doesn't converge at all.

Sundog sundog-education.com 219

Training a RNN is really hard. If you thought CNN's were hard, wait till you see RNN's.

They are very sensitive to the topologies you choose and the choice of hyperparameters. And since we have to simulate things over time and not just through the static topology of your network, they can become extremely resource intensive. If you make the wrong choices, you might have a recurrent neural network that doesn't converge at all. It might be completely useless even after you've run it for hours to see if it actually works. So again, it's important to build upon previous research when you can. Try to find some sets of topologies and parameters that work well for similar problems to what you're trying to do.

let's run an example

Sundog sundog-education.com 220

This all will make a lot more sense with an example, and you'll see that it's really nowhere near as hard as it sounds when you're using Keras. I used to work at IMDb, so I can't resist using a movie related example - so let's dive into that next, and see RNN's - Recurrent Neural Networks - in action.

So let's have some fun with recurrent neural networks. As usual, open up Anaconda Navigator and make sure you're in the RecSys environment, and start up Jupyter Notebook. Upload the Keras-RNN notebook from the course materials in the DeepLearningIntro folder, and then click on it to open it up once it is imported.

Although RNN's sounded pretty scary in the slides, using them in Keras is actually fairly straightforward. It's still tricky to tune things properly, but assuming you have some sort of a template of a similar problem to work from, it's quite easy in practice. What we'll do here is try to do sentiment analysis. This is going to be an example of a sequence to vector RNN problem, where we're taking the sequence of words in a user-written movie review, and we try to output a vector, that's just a single binary value, of whether or not that user liked the movie or not. That is, whether they gave it a positive rating. This is an example of doing sentiment classification using real user review data from IMDb. And since I used to run IMDb's engineering department, this is a little bit too tempting for me to do as an example here.

Just to give credit where credit is due, this is drawing heavily upon one of the examples that ships with Keras: the IMDb LSTM sample. I've embellished on it a little bit here, but the idea is theirs. And it does warm my heart that they include the IMDB dataset as part of Keras, free to experiment with. Anyhow, this is another example of how we're going to use LSTM cells, or long short-term memory cells. When you're dealing with textual data, which is sequences of words in a sentence, it doesn't necessarily matter where in the sentence that word appeared. You don't want words towards the end of the sentence counting more toward your classification than words that are at the beginning of the sentence. In fact, often, it's the other way around.

So we're going to use an LSTM cell to try to counteract that effect that you see in normal RNNs, where data becomes diluted over time, or as the sequence of words progresses, in this example. Let's just dive in and see how it works. We'll start by importing all the stuff that we need from Keras. We're importing sequence, a preprocessing module of the sequential model, so we can embed those different layers together. We're going to introduce a new embedding layer as part of our RNN, in addition to the dense layer that we had before. We'll import the LSTM module. And finally we'll import the IMDb dataset. So let's go ahead and shift-enter to do all of that and get Keras initialized.

Next, we can import our training and testing data. Like I said, Keras has a handy IMDb dataset pre-installed. Oddly, it has 5,000 training reviews and 25,000 testing reviews, which seems backwards to me, but it is what it is. The one parameter you're seeing here for num_words indicates how many unique words that you want to load into your training and testing dataset. So by saying num_words equals 20,000, that means that I'm going to limit my data to the 20,000 most popular words in the dataset. If someone uses some really obscure word, it's not going to show up in our input data. Let's go ahead and load that up. And since it does have to do some thinking, it doesn't come back instantly, but it’s pretty quick.

Let's take a peek at what this data looks like. So let's take a look at the first instance of training data here. It's just a bunch of numbers! It doesn't look like a movie review to me. Well, you can be very thankful to the folks behind Keras for doing this for you. When you're doing machine learning in general, models don't work with words, they work with numbers. So we need to convert these words to numbers somehow as a first step. Keras has done all this preprocessing for you already. So the number one might correspond to the word “the”. I actually have no idea what it corresponds to, but they've encoded each unique word between 0 and 20,000, because we said we wanted the 20,000 most popular words, to numbers. So it's kind of a bummer that we can't actually read these reviews and get an intuitive meaning of what these reviews are saying, but it saves us a whole lot of work. Often, a lot of the work in machine learning is not so much building your models and tuning them, it's just processing and massaging your input data, and making sure that your input data looks good to go. So even though this doesn't look like a movie review, it is a movie review. They've just replaced all of the words with unique numbers that represent each word.

We can also take a look at the training data. So the classification of this particular review was one, which just means that they liked it. So the only classifications are zero and one, which correspond to a negative or a positive sentiment for that review.

Now we just have to go ahead and set things up for our network. Let's start by creating some vectors for input here. Let’s break out our training and test data here. We're going to call sequence.pad_sequences( ) just to make sure that everything has a limit on them to 80 words. The reason we're doing this is because, like we said, RNNs can blow up very quickly. You have to back-propagate through time. So we want to have an upper bound on how many time steps we need to back-propagate to. By saying maxlen = 8 0 , that means we're only going to look at the first 80 words in each review, and limit our analysis to that. So that is a way of truncating our backpropagation through time. It's sort of a low-tech way of doing it, but it's effective. Otherwise, we would be running this thing for days. The only point here is to trim all of these reviews in both the training and the test dataset to their first 80 words, which again have been converted to numbers for us already. Shift-enter to run that as well.

Let's build up the model itself. For such a complicated neural network, I think it's pretty remarkable how few lines of codes are going on here. We'll start by creating a sequential model, meaning that we can just build up the topology of our network one step at a time here. We'll start with some additional pre-processing. We're using what's called an embedding layer here. All that does is convert our input data of words up to the first 80 words in a given review into dense vectors of some fixed size. So it's going to create a dense vector of a fixed size of 20,000 words, and then funnel that into 128 hidden neurons inside my neural network. That's all the embedding layer is doing. It's just taking that input textual data that's been encoded, and converting that into a format that's suitable for input into my neural network. Then, with a single line of code, we build our recurrent neural network. So we just say, “add an LSTM." Let’s go through the properties here. We're asking for 128 recurrent neurons in that LSTM layer. We can also specify dropout terms just in that same command here. So we’re asking for a dropout of 20%. And that's all there is to it. That one line of code sets up our LSTM neural network with 128 recurrent neurons and adds dropout phases of 20%, all in one step. Finally, we need to boil that down to a single output neuron with a sigmoid activation function, because we're dealing with a binary classification problem, and that's it. Run this block to set up our model.

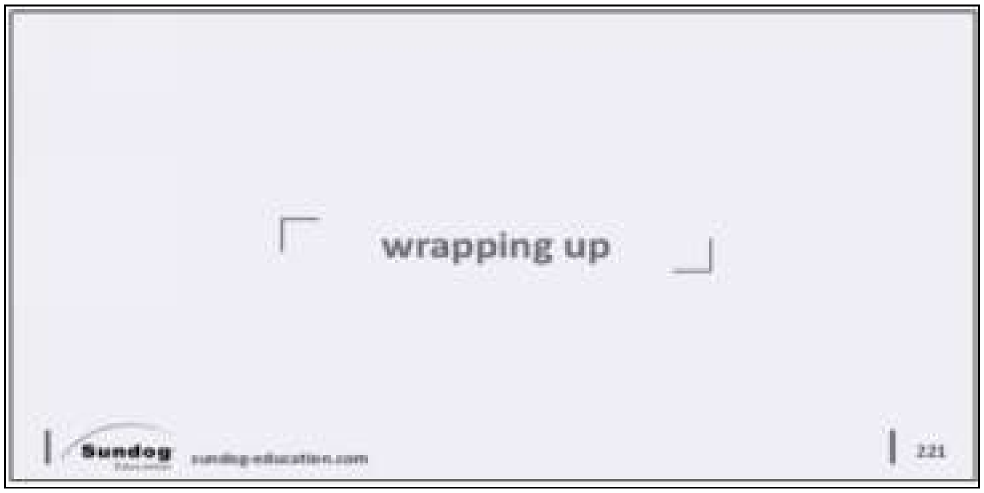
So we've defined the topology of our network with just four lines of code, even though it's a very complicated recurrent neural network, using LSTM cells and dropout phases. But Keras makes that all very easy to do. We then need to tell Keras how to optimize this neural network and how to train it. So we will use binary cross-entropy, because this is ultimately a binary classification problem: “did the person like this movie or not?" We'll use the Adam optimizer this time, just because that's the best of both worlds for optimizers. And then we can kick it off. Hit shift-enter to compile our model.

And at this point, you're ready to actually train your neural network. Let’s just walk through what's going on here. It's very similar to the previous examples. In this case, we're going to use batch sizes of 32 reviews at once. We're going to run it over 15 training steps, or epochs, set a verbosity layer that's compatible with Jupyter Notebooks, and provide the validation data for its use as well. Now, again, I'm not going to actually run this right now, because it will take about an hour. Like I said, RNNs are hard; they take a lot of computing resources. And since all I'm doing is running this on my single CPU, I don't even have things configured to use my GPU, or let alone a cluster of computers, this takes a long time. But I did run it earlier, and you can see the results here.

So over 15 epochs, you can see that the accuracy that it was measuring on the training data was beginning to converge. Seems like after about 13 steps it was getting about as good as it was going to get. And then furthermore, we can actually evaluate that model, given the testing dataset. So let's go ahead and call evaluate( ) on that with our test data. Again, using 32 batches. And if we were to run that, we would see that we end up with an accuracy of 81% on our model here. It doesn't sound that impressive, but when you consider that all we're doing is looking at the first 80 words of each review and trying to figure out just based on that beginning whether or not a user liked the movie or not, that's not too bad.

But, step back and think about what we just made here. We've made a neural network that can essentially read English-language reviews and determine some sort of meaning behind them. In this case, we've trained it how to take a sequence of words at the beginning of a movie review that some human wrote, and classify that as a positive review or a negative review. So in a very real sense, at a very basic level, we’ve taught our computer how to read. How cool is that? And the amount of code that we wrote to do that was minimal. It's kind of mind-blowing. It's really just a matter of knowing what technique to use to build your neural network, providing the appropriate training data, and then your neural network does the rest. It's really kind of spooky when you step back and think about it.

This is a great example of how powerful Keras can be, and a great example of an application of a recurrent neural network - not using the typical example of stock trading data or something like that. Instead we’ve used an RNN for sentiment analysis, where we took a sequence of words and used that to create a binary classification of a sentiment based on that sequence. RNN’s are very powerful tools for a wide variety of problems, and the Keras API makes them easy and fun to work with.



wrapping up

Sundog

sundog-education.com

221

So, that was a rather long diversion into the basics of deep learning, but I had to get you to the point of understanding recurrent neural networks before we start talking about applying deep learning to recommender systems. As it turns out, deep learning is especially suitable to predicting sequences of events, like which videos you're likely to watch next – so this has applications in recommender systems that deal with situations where the order of events matters. So, RNN's play a big role in the current research toward applying deep learning to recommender systems.

But, there are much simpler neural networks that can be used to recommend items too, so we don't have to jump straight to the hard stuff. Stick with me – we're going to get back to recommender systems now, and how to apply what you've just learned about deep learning to them.

And if you do want to keep learning more about deep learning in general, you can find the rest of my free course on deep learning on YouTube – just search for Sundog Education and you'll find it.

Recommendations with Deep Learning

recommendations
with deep learning

Sundog sundog-education.com 222

So now that you understand the theory behind deep learning and neural networks, let’s finally talk about how these techniques can be applied to recommender systems. You can imagine that with all the buzz around artificial intelligence lately, there’s a lot of current research on applying neural networks to recommender systems. But, is it all hype?

Let's get something out of the way right up front – just because a new technology is hot, doesn't mean that it's the right solution for every problem. You *can* train a neural network with user ratings or purchases and use it to make recommendations, but *should* you? As we've seen, deep learning can be very good at recognizing patterns in a way similar to how our brain may do it – it's good at things like image recognition and predicting sequences of events. But is making recommendations really a pattern recognition problem?

Think back to when we talked about Tensorflow – it doesn't really know much about neural networks per se, you actually use it by doing matrix multiplication and addition on the "tensors" that represent your artificial neurons, their weights, and their biases. Neural networks are fundamentally matrix operations, and there are already well established matrix factorization techniques for recommender systems that fundamentally do something similar. In SVD for example, we find matrices that we multiply together using weights that are learned from stochastic gradient descent. It's almost the same thing, just thought of in a different way. So yeah, you can think of recommender systems as looking for patterns – just very complex ones based on the behavior of other people.

So, matrix factorization can be modeled as a neural network. I think the main reason to experiment with applying neural networks to recommender systems is that it lets us take advantage of all the rapid advances in the fields of AI and deep learning. Amazon, for example, has open sourced a system called “Destiny” (that’s DSSTNE), which allows you to run huge neural networks that deal with sparse data on a cluster efficiently. They claim to be using this internally for their own recommender systems. There are also ways to use Tensorflow on a cluster, and take advantage of a whole fleet of GPU’s. And there is always research on new topologies for neural networks that can lead to fresh insights on how to make better recommendations using them. In some cases, approaches using neural networks have been shown to out-perform SVD already, even if it’s by a rather small margin.

So let’s dive into some ways you can apply neural networks to the problem of making recommendations.

Restricted Boltzmann Machines

restricted boltzmann machines (rbm)

Sundog sundog-education.com

226

The grand-daddy of neural networks in recommender systems is the restricted Boltzmann machine, or RBM for short. It's been in use since 2007, long before AI had its big resurgence – but it's still a commonly cited paper and a technique that's still in use today.

Going back to the Netflix Prize, the main things Netflix learned was that matrix factorization and RBM's had the best performance as measured by RMSE, and their scores were almost identical. Again, this shouldn't surprise us too much since we know you can model matrix factorization as a neural network. But they found that by combining matrix factorization with RBM's, the two of them working together provided even better results – they went from an RMSE of 8.9 to 8.8. A few years ago, Netflix confirmed they were still using RBM's as part of their recommender system that's in production.

Let's learn how it works.

rbm's: the paper

Restricted Boltzmann Machines
for Collaborative Filtering

[illegible]

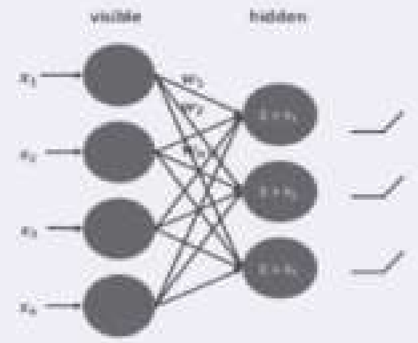
Abstract

[illegible]

Sundog sundog-education.com 225

First of all, if you're serious about using RBM's for recommendations, I recommend tracking down this paper so you can study it later once you understand the general concepts. It's from a team from the University of Toronto, and was published in the Proceedings of the 24 th International Conference on Machine Learning in 2007. If you just Google for the title of the paper: "Restricted Boltzmann Machines for Collaborative Filtering," you should find a free PDF copy of it that's from the author's page on the University of Toronto website, so I think it's legitimately free for you there.

what is a rbm

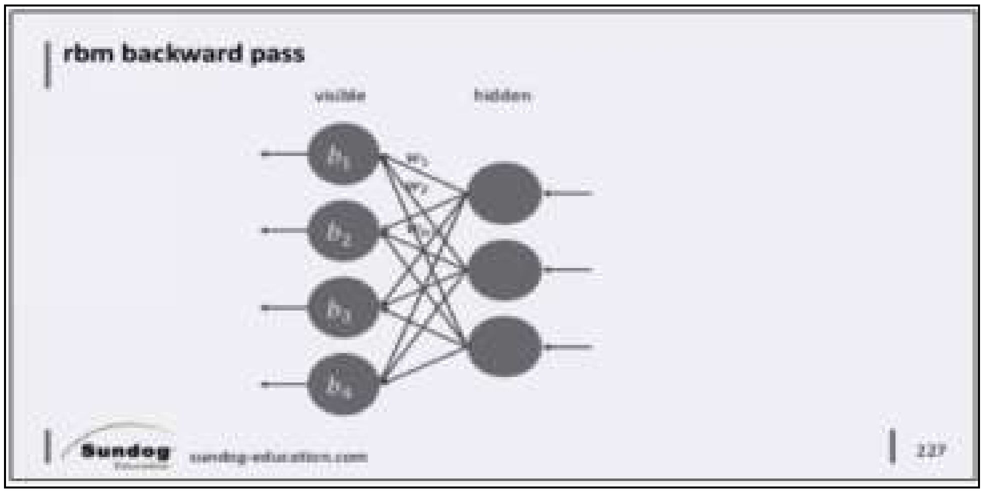


Sundog sundog-education.com 226

RBM's are really one of the simplest neural networks – it's just two layers, a visible layer and a hidden layer. We train it by feeding our training data into the visible layer in a forward pass, and training weights and biases between them during backpropagation. An activation function such as ReLU is used to produce the output from each hidden neuron.

Why are they called Restricted Boltzmann Machines? Well, they are “restricted” because neurons in the same layer can’t communicate with each other directly; there are only connections between the two different layers. That’s just what you do these days with modern neural networks, but that restriction didn’t exist with earlier Boltzmann Machines back when AI was still kind of floundering as a field. And RBM’s weren’t invented by a guy named Boltzmann; the name refers to the Boltzmann distribution function they used for their sampling function. RBM’s are actually credited to Geoffrey Hinston who was a professor at Carnegie Mellon University at the time; the idea actually dates back to 1985.

So, RBM’s get trained by doing a forward pass, which we just described, and then a backward pass, where the inputs get reconstructed. We do this iteratively over many “epochs,” just like when we train a deep neural network, until it converges on a set of weights and biases that minimizes the error. Let’s take a closer look at that “backward” pass.

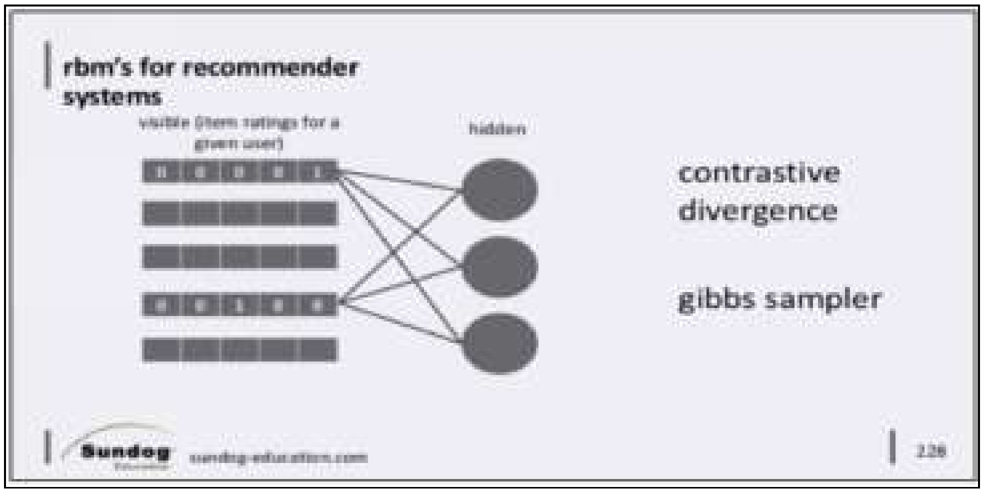


During the backward pass, we are trying to reconstruct the original input by feeding back the output of the forward pass back through the hidden layer, and seeing what values we end up with out of the visible layer. Since those weights are initially random, there can be a big difference between the inputs we started with and the ones we reconstruct. In the process, we end up with another set of bias terms, this time on the visible layer. So, we share weights between both the forward and backward passes, but we have two sets of biases – the hidden bias that's used in the forward pass, and the visible bias used in this backward pass.

We can then measure the error we end up with, and use that information to adjust the weights a little bit during the next iteration to try and minimize that error .

Conceptually, you can see it's not too different from what we call a linear threshold unit in more modern terminology. You can also construct multi-layer RBM's that are akin to modern deep learning architectures. The main difference is that we read the output of an RBM on the lower level during a backward pass, as opposed to just taking outputs on the other side like we would with a modern neural network.

So this all works well when you have a complete set of training data; for example, applying an RBM to the same MNIST handwriting recognition problem we looked at in our deep learning intro section is a straightforward thing to do. When you apply RBM's, or neural networks in general, to recommender systems though, things get weird. The problem is that we now have sparse training data – very sparse, in most cases. How do you train a neural network when most of your inputs nodes have no data to work with?



Adapting an RBM for, say, recommending movies given 5-star ratings data, requires a few twists to the generic RBM architecture we just described .

Let's take a step back and think about what we're doing here. The general idea is to use each individual user in our training data as a set of inputs into our RBM to help train it. So, we process each user as part of a batch during training, looking at their ratings for every movie they rated. So, our visible nodes represent ratings for a given user on every movie, and we're trying to learn weights and biases to let us reconstruct ratings for user/movie pairs we don't know yet.

First of all, our visible units aren't just simple nodes taking in a single input. Ratings are really categorical data, so we actually want to treat each individual rating as five nodes, one for each possible rating value. So, let's say the first rating we have in our training data is a 5-star rating; that would be represented as four nodes with a value of 0, and one with a value of 1, as represented here. Then we have a couple of ratings that are missing – for user/item pairs that are unknown and need to be predicted. Then we have a 3-star rating, represented like this with a 1 in the third slot.

When we're done training the RBM, we'll have a set of weights and biases that should allow us to reconstruct ratings for any user. So to use it to predict ratings for a new user, we just run it once again using the known ratings of the user we're interested in. We run those through in the forward pass, then back again in the backward pass, to end up with reconstructed rating values for that user. We can then run softmax on each group of 5 rating values to translate the output back into a 5-star rating for every item.

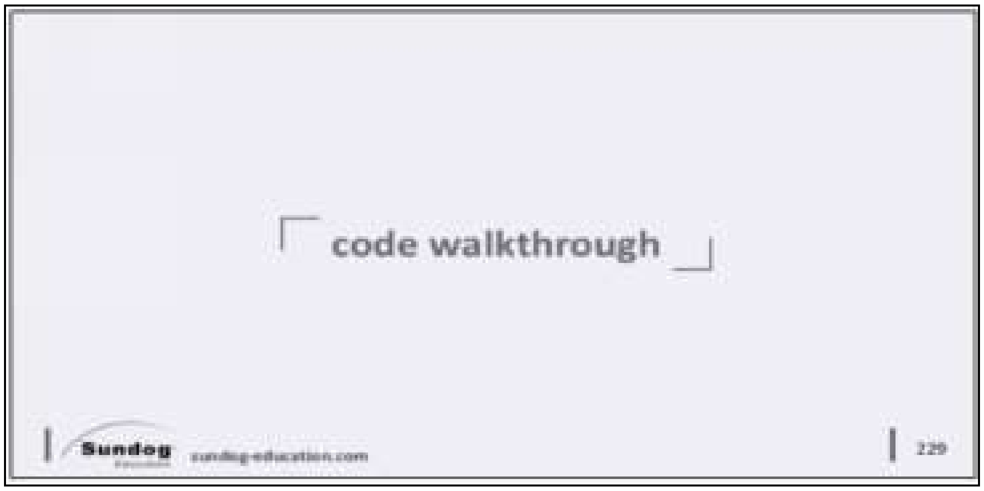
But again, the big problem is that the data we have is sparse. If we are training an RBM on every possible combination of users and movies, most of that data will be missing, because most movies have not been rated at all by a specific user. We want to predict user ratings for every movie though, so we need to leave space for all of them. That means if we have N movies, we end up with N * 5 visible nodes, and for any given user, most of them are undefined and empty.

We deal with this by excluding any missing ratings from processing while we're training the RBM. This is kind of a tricky thing to do, because most frameworks built for deep learning such as tensorflow assume you want to process everything in parallel, all the time. Sparse data isn't something they were really built for originally, but there are ways to trick it into doing what we want. But, notice that we've only drawn lines between visible units that actually have known ratings data in them, and the hidden layer. So as we're training our RBM with a given user's known ratings, we only attempt to learn the weights and biases used for the movies that user actually rated. As we iterate through training on all of the other users, we fill in the other weights and biases as we go.

For the sake of completeness, I should point out that Tensorflow actually does have a sparse tensor these days you can use, and there are other frameworks such as Amazon's DSSTNE system that are designed to construct more typical deep neural networks with sparse data. RBM's will probably become a thing of the past now that we can treat sparse data in the same manner as complete data with modern neural networks, and we will examine that in more depth in an upcoming section of this course.

The other twist is how to best train an RBM that contains huge amounts of sparse data. Gradient descent needs a very efficient expectation function to optimize on, and for recommender systems this function is called contrastive divergence. At least, that’s the function the paper on the topic uses successfully. The math behind it gets a bit complicated, but the basic idea is that it samples probability distributions during training using something called a Gibbs sampler. We only train it on the ratings that actually exist, but re-use the resulting weights and biases across other users to fill in the missing ratings we want to predict.

So, let’s look at some code that actually implements an RBM on our MovieLens data, and play around with it!



code walkthrough

Sundog sundog-education.com 229

Here we go – open up Spyder, and close out everything so you don’t get confused. Now, navigate into the DeepLearning folder of the course materials, and open up everything in there.

Let’s start by looking at the RBM.py file. This is what implements the restricted Boltzmann machine itself, with the twists we talked about that are specific to recommender systems, such as using contrastive divergence. It uses Tensorflow so it can use your GPU to accelerate things. It may look a little complex, and it is, but it’s still under 100 lines of code – so it’s not too bad, considering what a complex algorithm it is. I mean, we’re basically building a brain here that can keep track of hundreds of different people and guess what movies they might like, given a hundred thousand ratings. That exceeds human capabilities by like, a lot .

Our init function takes several different parameters – hyperparameters, if you will – that give us control over how the learning in our RBM works. It turns out these values are pretty important for how well it all works. "Visible dimensions" is kind of nonnegotiable – it's the product of the number of movies, and how many distinct rating values you have, as we discussed earlier. In other words, a pretty big number. "Epochs" is how many iterations you're going to take on the forward and backward passes while we try to minimize the error between our actual rating values and the reconstructed, predicted values. That is, how many times will we train our RBM across all of the users in our training data in hopes that it will converge on a good set of weights and biases we can use for future predictions. "Hidden dimensions" is the number of hidden neurons in the system, which can be surprisingly small compared to the number of visible neurons. Next we have the number of distinct rating values – since our data actually contains half-star rating values, we have 10 possible rating values and not just 5. We then have the learning rate, which controls how quickly we attempt to converge on each iteration – this needs to be balanced so you find the right values in a reasonable number of steps, without skipping over the best results in the process. And finally we have the batch size, which controls how many users we process at a time while training. We store all of these within the RBM object so we can use them in other functions.

The Train function is what we call when we want to create an RBM with weights and biases that will allow us to reconstruct any user's rating for any item. It takes in an array called X, which is a 2D array that has a row of rating values for every user. Those rating values are flattened binary values, one for each possible rating type, like we showed in the slides – so this input array contains a row for every user, and every row contains binary data for the number of movies times the ten possible rating values we have .

Again, we're using Tensorflow, so we'll start by resetting everything in case you try to train the same RBM more than once. Next we call the MakeGraph function, which we'll look at soon – it contains the RBM model itself. We then create a Tensorflow session to run the

RBM within, iterate over every epoch, shuffling the users every time – and divide it into batches of users that get run together as we converge on better weights and bias values – both hidden biases and visible biases.

GetRecommendations is how we get back rating predictions for a given user, using an RBM that's already been trained and has a set of optimized weights and biases in place. The input here is an array of every rating type for every movie for one user, where that user's known ratings are filled in. We define a simple RBM here for this purpose, with just a couple of lines. Our forward pass creates a hidden layer by multiplying the visible layer X with our weights, and adding in the hidden bias terms. We then reconstruct the visible layer by multiplying the hidden layer by the same weights, and adding in the visible bias terms we computed during training – that's the backward pass. What's left are the reconstructed rating values for every movie, which we pass back for further processing.

So let's dive into the meat of all this – the MakeGraph function. There's a lot to wrap your head around in here, so I'm going to take it slow – and feel free to pause if you need to let things sink in. It's not trivial by any means.

First, we set a consistent random seed so we get consistent results from run to run. Simple enough. You don't have to do this, but it will minimize confusion while you're developing.

Next, we need to set up placeholders in Tensorflow for the nodes, weights, and biases that make up our RBM. We start with "X", which is our visible nodes. Remember, we have one for every possible rating value for every possible movie – that product is contained in our "visibleDimensions" property. Even though it will ultimately be binary data, the RBM will place floating point values in there in the backward pass that we need to convert back to binary, so the type is float32.

Next we set up our weights, and these must be initialized with random data. But it can't just be any random data – there are different best practices for randomizing weights in a neural network depending on what activation function you're using. In our case, -4 over 6 times the square root of the total number of nodes is the right thing to do, and that gives us the magnitude of the positive and negative extents of the initial random weights. This runs counter to what's in the RBMs for Collaborative Filtering paper, but presumably they were using different activation functions since modern ones weren't really a thing back then. The paper didn't come with code, so there are some details we have to figure out ourselves.

So, line 55 is what sets up all the weights – we use Tensorflow's random_uniform initializer to set up the weights between the hidden and visible nodes and randomize them to the desired values all at once.

Next, we set aside the bias terms. Remember in an RBM we do both a forward and backward pass, and we end up with bias terms on both the hidden and visible layers as a result. So we have both a hidden bias tensor associated with the hidden nodes, and a visible bias tensor associated with the visible nodes, and we initialize them all to zero.

Notice that we've made X, weights, hiddenBias, and visibleBias all members of the RBM class itself, because these are the parts of trained RBM we need to keep around even after MakeGraph is called – we'll need those to reconstruct the RBM for making predictions later .

Now we need to set up the specifics of the forward and backwards passes, and expose what's needed to allow the system to converge using contrastive divergence while it is learning. Let's start with the forward pass. Skip to line 66.

hProb0 is a new tensor that represents the values of the hidden nodes at the end of the forward pass. We multiply the visible nodes in X, which contain our training data from a given user, by our weights, which initially are random but get better with each batch we train with. Then we add in the hidden bias terms, apply a sigmoid activation function, and we’ve got our hidden layer output from the forward pass. Simple enough.

But for contrastive divergence, we need to perform Gibbs sampling of those hidden probabilities coming out of the hidden nodes. That’s what’s going on in line 68. We have a new tensor, hSample, which is applying the ReLU activation function to the sign of the hidden probabilities with a random value subtracted from them – this has the effect of selecting a random sample of the complete set of hidden node values.

So, the weights for that sampled forward pass is what we actually store in the tensor we’re naming “forward,” and computing those weights is what makes this all happen. Again this is just to implement the Gibbs sampling and contrastive divergence described in the paper. RBM’s will work without this extra complexity; it just exists to allow the algorithm to run faster by having fewer stuff to process on the backward pass – and it also has some of the same benefits you’d get from a dropout phase in Keras to prevent overfitting.

Now we can turn our attention to defining the backward pass, where we reconstruct the visible layer by running the RBM in reverse from the hidden node outputs. That’s exactly what we’re setting up on line 74; the tensor “v” takes our sampled output from the forward pass, which lives in the hSample tensor, multiplies in the weights, and adds in the visible bias terms.

But of course, it’s not quite that simple. We have to deal with missing ratings, and translate the scores that get reconstructed into binary values that reflect actual rating classifications. First, let’s deal with the missing ratings. We know we don’t want to predict anything at all while training for movies a user didn’t rate, since we don’t have a rating to compare the prediction against. This is the sparse data problem we described earlier. So to handle that, we’ll create a mask from the original ratings from the user that exist in the X tensor.

We’ll start by making sure that everything in our mask is either 0 or 1, but applying the sign function to it. Then, we’ll reshape our mask tensor into 3 dimensions: one for each user in our batch, one for each movie, and one for each set of 10 values that represent a single rating.

So at this point, each individual set of 0-9 rating values might contain all zeroes, which would indicate a missing rating that doesn’t exist in our training data at all. Or, there might be a single value of one in it, which would indicate an actual rating value. To figure out which it is, we’ll apply the reduce_max function on line 79 – it finds the maximum value in each rating. So missing ratings end up getting reduced to the value 0, and actual ratings get reduced to the value 1.

With me so far? The vMask3D tensor contains 3 dimensions at this point – one for each user in a batch, one for each movie the user might have rated, and one indicating 1 if the user actually did have a rating for that movie, or 0 if that rating is missing .

OK, so now let’s re-arrange our reconstructed visible layer into those same 3 dimensions, so we have each set of 10 binary rating values organized together for each movie. That’s what’s happening on line 82.

On line 83, we multiply the reconstructed visible layer by our mask, which has the effect of leaving reconstructed data for ratings we actually have data for, and leaving all the missing ratings as zero. We then apply the softmax function to the results, which will only do anything meaningful on the “real” ratings in the reconstructed set.

Now on line 84, we’ll flatten the ratings back into the form that our RBM expects, where individual rating values aren’t treated specially at all – they no longer have their own dimension; we just have users and buckets of data associated with them in 2 dimensions now. At this point, we’re done reconstructing our ratings data – that’s what lives in the vProb tensor. We took our reconstructed inputs by running the RBM backward, filtered out all of the missing rating data we shouldn't be looking at while training, and applied softmax to each block of 10 nodes that represents a single user/item rating.

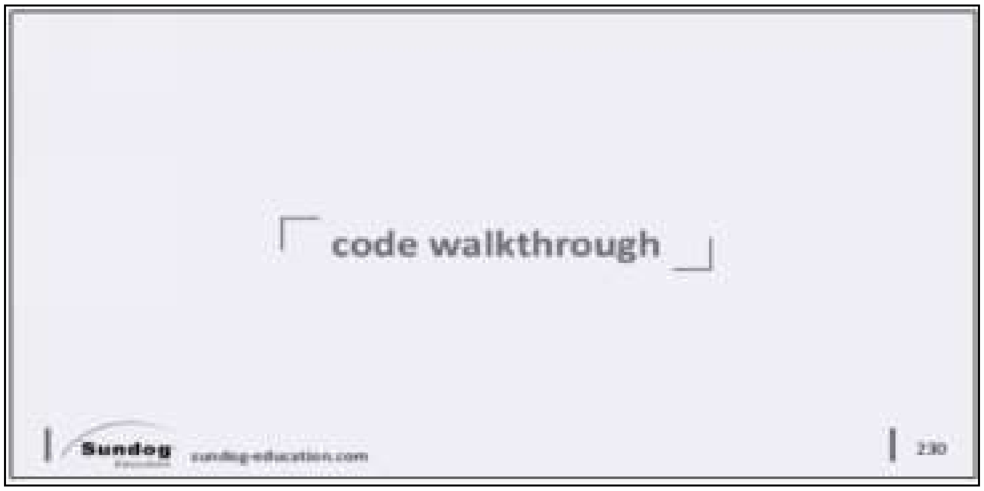
We want to learn the correct hidden biases, so we're going to run things forward again here on line 86 to compute a new set of hidden biases based on our reconstructed data. That allows us to compare the new hidden biases to the ones we started with in this batch, and minimize the error between the two. It's a little unusual to optimize on the hidden output in addition to the errors at the visible layer, but after experimenting with a bunch of different approaches I found that this produced the best results. Put another way, this lets us compare the hidden bias terms we end up with from the original training data to the hidden bias terms we end up with from the reconstructed data from the backward pass .

And, we'll set up the final backward pass itself here on line 87 that makes all of this happen. This whole section of code is a bit convoluted because we had to convince Tensorflow to deal with missing ratings and treat blocks of adjacent rating values as softmax units, but it does the job. In some ways, it might have been easier to code this algorithm up without using Tensorflow, but then we couldn't really use our GPU to do it quickly.

Alright, finally we tie everything together and tell Tensorflow what we want it to do on every batch. We run the forward and backward passes, and update the weights, with what's on line 91. Line 93 updates the hidden bias terms, trying to minimize error between the hidden node outputs from the original data and from the reconstructed data. And line 95 updates the visible bias terms, trying to minimize error between the original training data and the reconstructed visible data.

Then, we stick it all inside self.update, which is ultimately what gets run when we call our Train function.

Whew. That is a lot to digest, and getting this code to work was a massive effort on my part as well. I also want to stress that this is only one way to do it – there are a lot of other little details and suggested improvements in the RBM paper that we chose to ignore here. But we can't use it quite yet… we need to tie it into our recommendations framework first, so let's review that next.


code walkthrough

Sundog sundog-education.com 230

Let's peek next at the RBMAlgorithm.py file. It's basically a wrapper around the RBM module we just looked at, which makes it easier to use and ties it into our recommendation framework.

The initializer just takes the same hyperparameters the underlying RBM module needs, and stores them for when that RBM class gets instantiated.

Next have a little utility function called softmax. If you remember from RBM.py, our GetRecommendations function returns raw results from the backward pass of the RBM in the visible layer. While you can sort of think of these as probabilities for each rating classification for each item, they aren't really probabilities. The softmax function allows us to convert them into values that all add up to 1, and can be treated mathematically like probabilities. This is sort of the normal thing to do when dealing with classifications in neural networks, but as you'll see in a bit it allows us to do another handy trick as well.

The fit function is what our framework calls when it wants to train our recommendation model, and it takes in our training data. After calling the base class, we extract the number of users and number of movies in our training data, and then we use that to build up a new matrix called trainingMatrix that is in the shape our RBM class expects. It’s a 3D array of users, items, and binary arrays of 10 items that represent a specific rating classification. We initialize the training data to contain all zeroes, so we can identify missing ratings later on.

Again, the reason we have 10 rating types is because our ratings data is a five-star scale with half-star increments. So those ten binary fields that represent each rating correspond to 0.5 stars, 1 star, 1.5 stars, etc. A five star rating would be representing by the values 0 0 0 0 0 0 0 0 0 1, for example – that’s nine zeroes and a one at the end, in the slot that corresponds to 5.0 stars.

So now we need to populate this training matrix with the training data we actually have. Again, most of this matrix will remain unfilled – and when dealing with very large scale data you’d probably want to investigate ways of storing this matrix in a sparse representation to save space.

We go through every rating in our training set, which consists of a user ID, item ID, and actual rating score. We then convert the rating into an index into our binary array – this math just translates values that range from 1 to 5 into integers that range from 0 to 9. The converted rating is then used as an index into the final dimension of out training matrix, where we set a 1 to indicate that particular rating score for this particular user/item pair exists.

Our RBM however wants to work with 2D data, so we flatten the items and ratings into a single dimension, with users left as the first dimension. That’s what this reshape call does on line 38. Negative 1 is a special value in reshape, and it just means “divide everything else equally throughout this dimension. ”

Next we’re ready to create our underlying RBM object on line 41. The size of that flattened second dimension determines the number of visible nodes we need to process each individual user, and we pass along the hyperparameters we captured when initializing the RBMAlgorithm.

Now to train our RBM, we just call the Train function, which does all the complicated work of setting up the RBM’s graphs in Tensorflow and running learning on it over however many batches and epochs we specified.

Next, we need to build up rating predictions for all of the missing user/movie pairs using the RBM we have trained, or specifically the weights and biases our RBM has learned from the training data. That will let us very quickly accesses those rating predictions when the framework calls the “estimate” function like a gazillion times.

We start by creating the predictedRatings array; it’s just a 2D array that maps users/item combinations to their predicted rating scores.

We iterate through every user in our training set on line 45, and print out status updates once we’ve processed every 50 users.

As you might recall, we wrote the GetRecommendations function in RBM.py to produce recommendations for an entire user at a time. This is what the RBM does naturally; we trained it using ratings from individual users as inputs on the visible nodes, and we are now using that trained RBM by feeding it the ratings of a user we’re interested in as the new inputs on its visible nodes. Our trainingMatrix is already organized the way our RBM needs it, so we just pull off the ratings for the user in question, and pass them in on line 48. The RBM then runs a forward pass using these ratings, and a backwards pass where it reconstructs the missing ones using the weights and biases it learned during training .

We now want to “un-flatten” the results, so we have our binary rating category data organized nicely by item rating. That’s all this reshape function is doing on line 49.

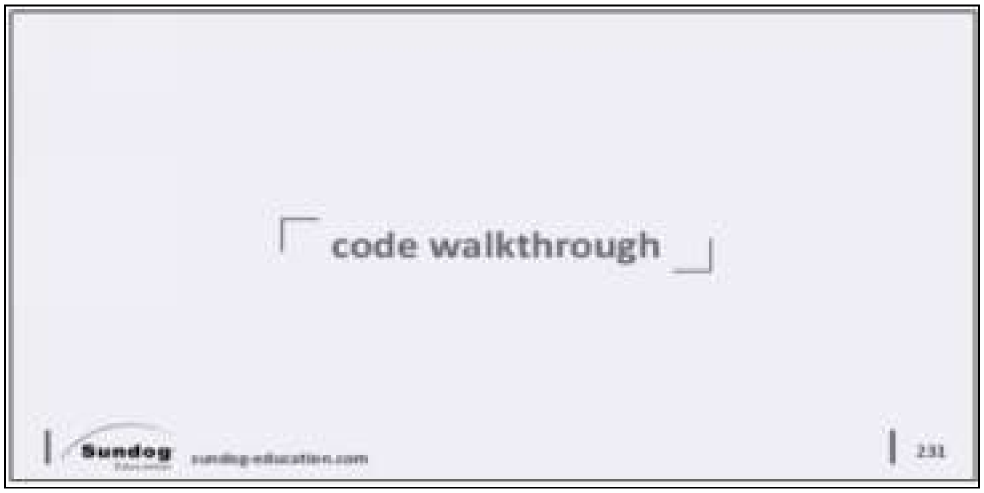
Now things get a little bit interesting. The raw, reconstructed results in those ratings categories aren’t nice zeros and ones like we want; they are numbers that are all over the place, and we need to convert them into an actual usable rating. We could just pick the rating category with the highest value in it, and that would be a perfectly reasonable thing to do. The only problem that this restricts you to the specific rating values that actually exist as categories – you can end up with a rating prediction of 4, 4.5, or 5.0, for example, but there’s no way to get a prediction of say 4.92 with that approach. We're losing some nuance into just how confident the RBM is in a given rating category by just picking the one with the biggest score, and as a result we end up with a huge multi-way tie for movies where its best guess is a 5.0 rating. We then have to arbitrarily pick top-N results from that tie, which doesn't work well. It's yet another example of where an algorithm designed to maximize prediction accuracy runs into real-world trouble when you apply it to top-N recommendations.

What we do instead in the loop between lines 51 and 59 is an alternative approach suggested in the paper. We first normalize the values in all of the ratings categories from the reconstructed data, and since this isn't 2007 we'll use the softmax function for this, which is sort of the standard way to do that with categorical data these days. So now we can treat this set of 10 rating categories as probabilities for each rating score. Given that, we can compute the "expectation" from that probability distribution. "Expectation" is a statistical term, and it's equivalent to a weighted average of these rating scores, weighted by their probabilities. That's what's going on in line 58; we're using Numpy's average function, which lets us compute a weighted average in one line. This approach has its own problem, in that it tends to guess rather low ratings overall – but we're more interested in the top-N results than prediction accuracy here. It's the rankings that matter for us.

Finally we convert the 0-9 rating index into a 1-5 rating score on line 59, and store it in our predictedRatings array for use in the "estimate" function, which is next.

We want "estimate" to be as fast as possible, so not much is going on here. We just check that the user and item for which a rating estimate is being requested actually exist in our data, look up the predicted rating we computed and stored within the "fit" function, and just check that it's not some ridiculously low value that should just be discarded. And that's it!

So we're about ready to run this now, and see what happens.


code walkthrough

Sundog sundog-education.com

231

Alright, we’ve been talking about RBM’s for long enough. Let’s actually run the thing and see how it does .

Select the RBMBakeOff.py file, and take a quick look at it. There’s not a lot to talk about – what we’re doing is pitting our RBM algorithm with 20 epochs and default hyperparameters against random recommendations.

Hit the play button, and get a cup of coffee, catch up on your messages – whatever you need to do. We did pass “true” to the Evaluate function, so we’re going to run all of the top-N metrics on everything which can take quite a bit of time. So pause this video, and resume when you’re ready to review the results with me.

OK, there’s a lot to digest here. Let’s start by looking at the table of all the metrics.

The accuracy measures, RMSE and MAE, are better than random – but they’re not great. We mentioned before that the way we’re computing predicted rating values tends to artificially lower them – the “expectation” values we end up with don’t get any higher than 4 stars. Accuracy is the promised strength of RBM’s though; in the paper, RMSE is the only way in which they evaluated their algorithms, and they managed to very slightly outperform SVD after tuning things sufficiently.

The hit rate metrics however are underwhelming. We're dealing with small numbers here so it's hard to read too much into it, but even random recommendations did better with hit rate no matter how you measure it. That's disappointing, because even if there is a bias in our predicted rating scores, we would hope that the rankings would still be meaningful.

Don't read too much into the coverage metric – it's based on a rating threshold of 4.0, and as we said our predicted ratings just don't get that high with this approach .

Interestingly, RBM produced results that were less diverse than random ones. So, that might mean that it is picking up on coherent themes and recommending the same sorts of things to individual people, implying that it is doing something useful at least. The higher novelty score however implies it's digging fairly deep into the long tail of movies at times to produce recommendations. These might be really insightful recommendations, or they could just as easily simply be spurious results.

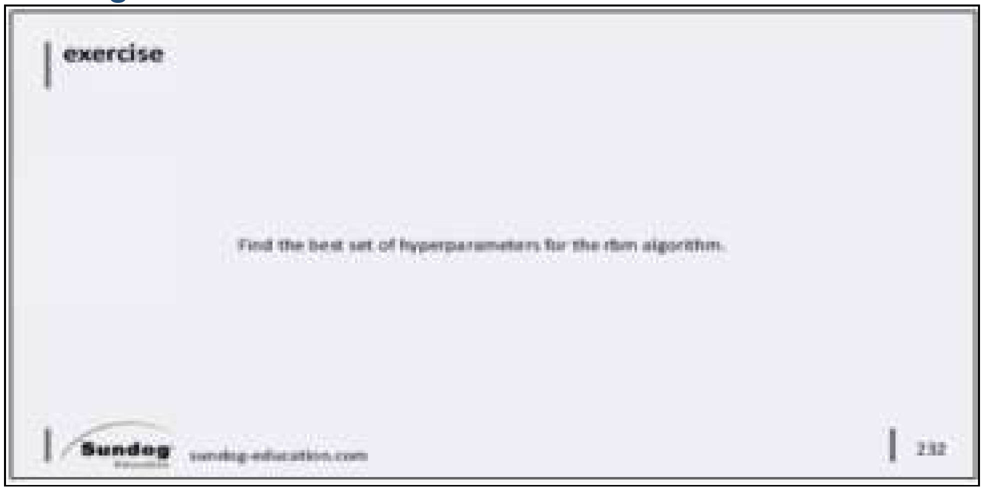
Hm – so not much payoff for all that work, so far – at least not quantitatively. Let's take a subjective look at the results, and look at the top-N recommendations that came back for our favorite user, user 85.

These actually look like decent recommendations to me. "Moon" and "Harry Potter" are good results for a sci-fi and fantasy fan, and we have a couple of classic Japanese animation films too. We've got a couple of popular action/adventure titles in there too. There are a few obscure foreign films sneaking in, which may be spurious – but the mix of recognizable films to obscure ones seems like a reasonable balance.

Overall, it seems like we might be on to something, but our results from our first cut at it aren't too exciting. With machine learning though, it's easy to give up too soon – often the issue doesn't lie in the algorithm, but in your choices of hyperparameters, or in the nature of the data you're using to test it with. Maybe 100,000 ratings is just too small of a data set to train an RBM, especially when we're doing Gibbs sampling to further cut down the data we're using to train it with. Maybe we should investigate using something other than softmax when converting the reconstructed visible layer in our RBM to probabilities for rating values, since it seemed to result in rating predictions that were too low across the board. There's a lot to tweak and experiment with to see if you can squeeze more performance out of it .

You can probably guess what your next exercise is going to be, then – let's talk about that next.

Coding Exercise



exercise

Find the best set of hyperparameters for the rbm algorithm.

Sundog sundog-education.com 232

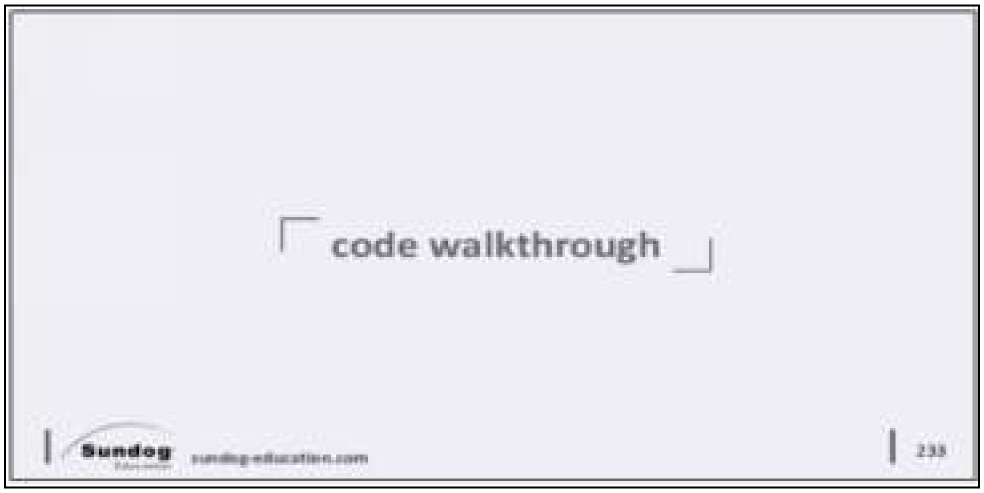
So, here's your challenge: see if we just have the wrong topology and parameters for our RBM. The right number of hidden nodes, the right learning rate, the right batch size, and the right number of epochs needed to converge all depend on the amount and the nature of our training data, so there is no single correct answers for RBM's in general. It's your job to tune the RBM for the data we have.

So give it a go! As we did before, use surpriselib's GridSearchCV class, and pick a couple of hyperparameters to work with. Keep trying sets of different values, using the results each time to converge on a better set of values to try next time, until you have parameters that yield the best results.

I'd encourage you to write this from scratch if you're up for it, but you may have noticed my solution is in your course materials as the RBMTuning.py file. If you choose to cheat and just use my code, it's still a worthwhile exercise, as most of the work here is just the process of converging on the right values by trying different stuff out.

In general, I advise starting with the default values for a given parameter, and try doubling it just to see if you should be increasing or decreasing it. If doubling it helped accuracy, try a point halfway between your original values you tested to see if that works even better. Or if doubling hurt accuracy, try halving the value instead. It's a bit of a slog, but it's something you'll have to do in the real world – and it's as important as anything in getting the best results from your work. Pace yourself, though – running cross validation on multiple permutations, training an RBM every time, takes a lot of time. Be prepared to just run it in the background, do something else for awhile, tweak things, and run it again – it can take hours of wallclock time, but the time it actually requires your attention isn't so bad.

In the next section, we'll review my results, so don't continue until you've had a chance to try it yourself.



code walkthrough

Sundog sundog-education.com 233

As you probably learned the hard way, it takes a really long time to tune this. What I learned from the process however, was a couple of things. First of all, it seemed that as far as accuracy is concerned, reducing the number of hidden units and increasing the learning rate helped – but not by much! Even after days of fiddling with it, I couldn’t get the RMSE below 1.18 or so, and that’s not a significant improvement over the 1.19 we started with.

And, these supposedly better recommendations don’t look subjectively better if you scroll down and look at the results.

So, while hyperparameter tuning might squeeze out some gains in our RBM, it seems like our problems are deeper than the parameters. I suspect it’s really a problem of not having enough data to properly train it. Later in this course, we’ll scale up a sparse neural network similar to a RBM and run it in the cloud, which will allow us to experiment with a much larger data set to see if we get better results that way.

In the meantime though, if you were able to find a better set of parameters, or perhaps a tweak of your own to the code we provided, please tell us about it! Hop on the Facebook group for this course that we introduced you to in the setup lecture, and share your findings there with us and your fellow students.

Deep Neural Networks for Recommendations

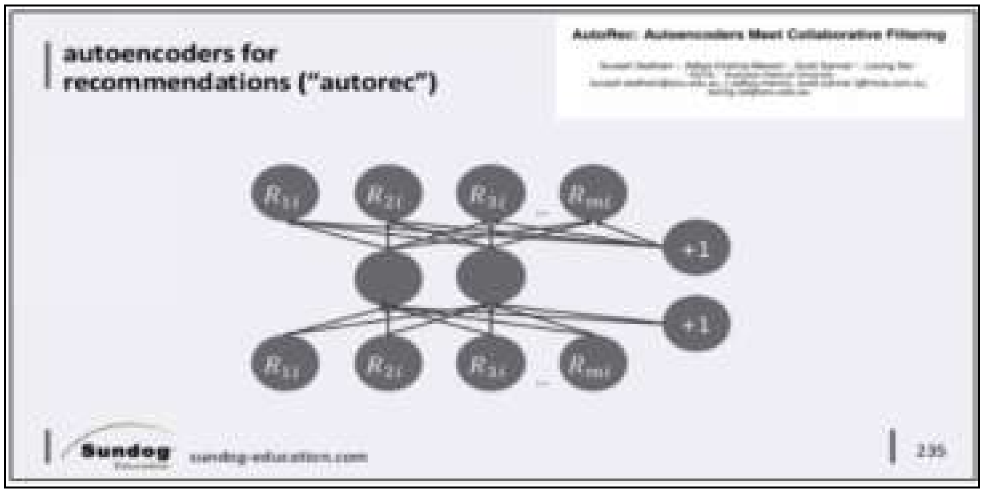
deep neural networks for recommendations

Sundog Education sundog-education.com

234

We mentioned that restricted Boltzmann machines were a very early type of neural network, and the field of deep learning has evolved considerably since then. What happens if we apply a more contemporary neural network to the problem? Well, as we'll see it's possible, but not without its challenges .

Autoencoders



autoencoders for recommendations ("autorec")

AutoRec: Autoencoders Meet Collaborative Filtering

[illegible]

Sundog Education sundog-education.com

235

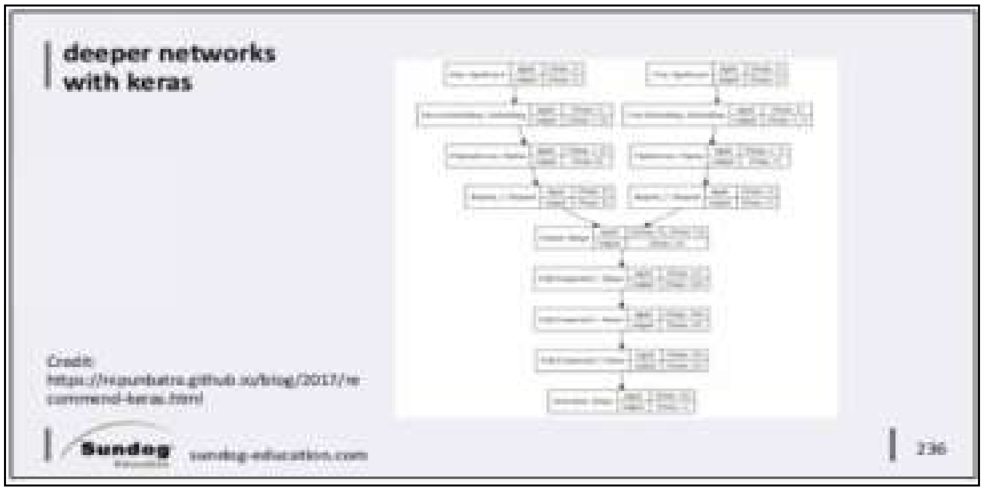
People started using deeper neural networks for recommender systems in 2015, which seems pretty recent - but it's a long time in the context of current AI research!

A group from the Australian National University published a paper called "AutoRec: Autoencoders Meet Collaborative Filtering," and they used the topology you see here. It looks a lot more familiar to the sorts of networks we covered in our introduction to deep learning. You have three layers: an input layer on the bottom that contains individual ratings, a hidden layer, and an output layer that gives us our predictions. A matrix of weights between the layers is maintained across every instance of this network, as well as a bias node for both the hidden and output layers.

In the paper, they trained the network once per item, feeding in ratings from each user for that item in the input layer. A sigmoid activation function was used on the output. All in all, it's a pretty straightforward approach – and they reported slightly better results compared to using an RBM. But the implementation is a bit different – RBM's just had separate bias terms for each pass, while here we have a whole separate set of weights to work with too.

This sort of architecture also has the benefit of being a lot easier to implement in modern frameworks such as Tensorflow or Keras. But there's still one wrinkle – the sparsity of the data we are working with. In the paper, they briefly mention that "we only consider the contribution of observed ratings." So, they were careful to process each path through this neural network individually, only propagating information from ratings that actually exist in the training data, and ignoring the contribution from input nodes that correspond to missing data from user/item pairs that weren't rated at all. This is still a tough thing to do in Tensorflow; while Tensorflow does have "sparse tensors," there's no simple way to restrict the chain of matrix multiplications and additions needed to implement a neural network to just the input nodes with actual data in them. Any implementation you'll find of this using Tensorflow or Keras just ignores that problem, and models missing ratings as zeroes. You can still get decent results with enough effort, but it's a very fundamental problem to applying deep learning to recommender systems.

This architecture is referred to as an "autoencoder." The act of building up the weights and biases between the input and hidden layer is referred to as encoding the input – we are encoding the patterns in the input as a set of weights into that hidden layer. Then, as we reconstruct the output in the weights between the hidden and output layers, we are decoding it. So, the first set of weights is the encoding stage, and the second set is the decoding stage. Conceptually, this isn’t really any different from what we were doing with RBM’s – in an RBM, we encoded on the forward pass, and decoded on the backward pass.



deeper networks
with keras

Credit:
https://nipunbatra.github.io/blog/2017/re
commend-keras.html

Sundog sundog-education.com

236

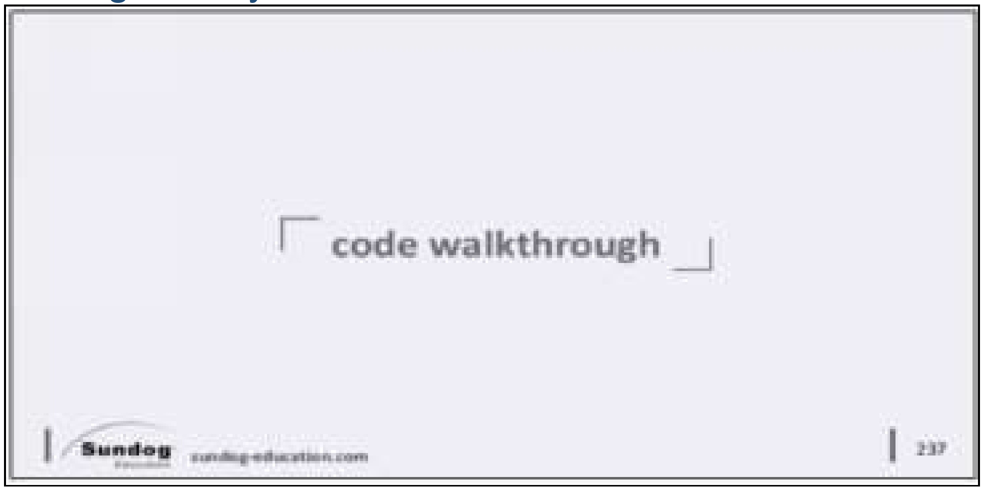
That problem hasn’t stopped people from trying, however. A couple of ideas that I’ve seen used a few times are using deeper neural networks with more hidden layers, and one-hot-encoding the user and item data together into a single input layer. That’s what you see in this architecture printed out from Keras; items are embedded on the left side, and users on the right. Both are flattened, and a dropout layer applied to prevent overfitting, and then they are concatenated together before feeding it all into a deep neural network.

But, it still has the problem of not being able to distinguish between missing ratings and ratings of a value 0. Fundamentally it models missing ratings as a signal that a user really, really hated a given item, and that’s just not an accurate representation of reality. In the end, this particular network was unable to outperform matrix factorization on the ml-100k data set. In part it’s due to the sparsity of the data, and it’s also because 100,000 ratings just isn’t anywhere near enough data to train a network as complex as this.

When we talk later in this course about scaling things up to run in the cloud, we'll talk about Amazon's system which is called DSSTNE, or "Destiny" for short. They actually have solved the problem of properly handling missing ratings, and their results are really good. So don't give up hope on deep neural networks for recommendations yet – they actually can work quite well, once you have the right tools.

It's not hard, however, to implement an autoencoder in Tensorflow that just treats missing ratings as zero values, so let's give it a shot and see for ourselves how well it does.

Coding Activity



code walkthrough

Sundog sundog-education.com 237

So let's go back to Spyder, and if you still have the contents of the DeepLearning folder opened up, you should be able to select the AutoRec.py file.

Like our RBM example, our implementation of using an autoencoder for recommendations uses Tensorflow, and it's structured in much the same way. As a reminder, an autoencoder is just a 3-layer neural network with an input layer, a hidden layer, and an output layer. Learning the weights between the input and hidden layer is called encoding, and reconstructing predictions with the weights between the hidden layer and output layer is called decoding. But fundamentally, it's just a neural network with one hidden layer.

Let’s skip to the MakeGraph function, as that’s where things are fundamentally different from the RBM example.

It’s not too complicated. You can see we’re setting up the weights for the encoding and decoding here, randomly initialized. These weights are learned and shared for every user we train the autoencoder with. We also set up a set of biases for both layers. This is a little bit different from what the original “autorec” paper proposed – they only had a single bias node shared across the entire layer, while we learn biases for each node, which is more in line with modern practices.

Next, we set up the layers themselves. Our input layer receives ratings for each item for a given user. We the construct our hidden layer by multiplying our input layer with the encoding weights and adding in the encoding bias terms, then applying a sigmoid activation function to them. Our output layer applies the learned decoder weights and biases to what’s in the hidden layer, and applies a sigmoid activation function to that final result as well – this makes up our actual predicted ratings for every item for a given user.

To measure error, we need to compare those predicted ratings to the actual ratings, and those are in our original input layer, so we just copy that here for use in our loss function. We define our loss function of MSE between our predicted and actual ratings, and use the RMS optimizer to minimize that error. You might try using more modern optimization functions such as Adam here if you want to tinker a bit.

That’s basically all there is to talk about here; the Train function kicks it all off in pretty much the same way as we did before with RBM .

Let’s shift our attention to the AutoRecAlgorithm file. There are a few things that have changed from the RBM example worth talking about. First of all, we’re not modeling this as a classification problem where each individual rating value between 0 and 5 stars as treated as a different input and output node. Instead, we just normalize our input ratings into the range 0-1 here on line 31, and restore them to their original ranges on line 45. So we don’t have to deal with softmax and expectation values and all that, in this example. As such, our input matrix that we pass into our Train function on line 35 is just a 2D array of ratings between users and items.

By the way, this is different from what was in the original autorec paper – they flipped things, so they trained the network on items instead of users. They would feed in all of the ratings by each user for a given item, while we're feeding in the ratings for every item for each given user. Again, if you want to tinker, you might try doing it the other way to see if you get better results.

Let's kick it off and see how it does. Open up the AutoRecBakeOff file here – there's not much to talk about in what it does, it just uses our framework to compare AutoRec to random recommendations as we've done before. Hit play, and it will take several minutes for it to do all of its work, as we've set this up to do all of the top-N metrics as well as generate some sample top-N recommendations so we can get a more comprehensive picture of how it's doing.

Now that it's done, let's look at the results. Let's scroll up to the table of metrics. The results are… well, pretty disappointing. RMSE and MAE are both worse than random. Although, we've seen that systematically under or over estimating ratings can have this effect without impacting the quality of top-N recommendations, so we shouldn't panic just yet. But, the hit rate is also worse. Average reciprocal hit rate is a little bit higher, so the results pr obably aren't terrible – but certainly they don't seem to have been worth the effort. Diversity and novelty are about the same as random – and our "random" recommendations are actually using a normal distribution to favor popular items a bit more, so at least we're not recommending super-obscure, totally random things to users it seems.

Let's take a qualitative, subjective look at the results as well, and scroll down to our sample top-N recommendations for user 85. These recommendations don't strike me as horrible, but there's certainly not inspiring given what we know about this user. The only result that really seems relevant to this user’s interests is Raiders of the Lost Ark.

So, what went wrong? I spent a lot of time digging into this myself. If you start investigating the actual rating predictions being made, you’ll find that most of them are very close to zero. This tells me that all the zeroes in our training matrix that represent missing ratings are really messing things up. We don’t have enough actual ratings to prevent our neural network from learning that for most items, the best prediction of a rating is zero, which to us means the user never saw that movie to begin with. Since our network can’t distinguish between missing ratings and a zero rating, it just thinks people hate pretty much everything. If you were to throw enough training data at the network it might start to overcome that, but 100,00 ratings is nowhere near enough for that to happen. I’ve seen people get reasonable results from an approach like this with a million or more ratings, but it’s still a fundamental problem that we’re ignoring the requirement to only consider actual ratings while training the network.

I spent a bunch of time tuning the hyperparameters, such as the learning rate and number of hidden nodes, and it didn’t help much. Any improvements to RMSE were purely artificial, because our code discards rating predictions that are too close to zero and those don’t count toward RMSE.

If you want some more hands-on practice with Tensorflow however, it would be a good exercise to try fiddling with those parameters yourself, and also to try deeper neural networks. What happens if you have two hidden layers instead of one? It’s not too hard to wire that extra layer in, so consider that a challenge. Don’t expect the results to be much better though; our fundamental problems here are the sparsity of our training data, and not having enough of it to begin with. A more complex network won’t help if you don’t have enough data to train it with!

Remember though, all is not lost! Dealing with sparse training data, and being able to train with massive amounts of training data, are both solved problems. We just need the tools we'll cover in our section on scaling things up before we can revisit deep learning and get results we can truly be excited about. We'll get there.

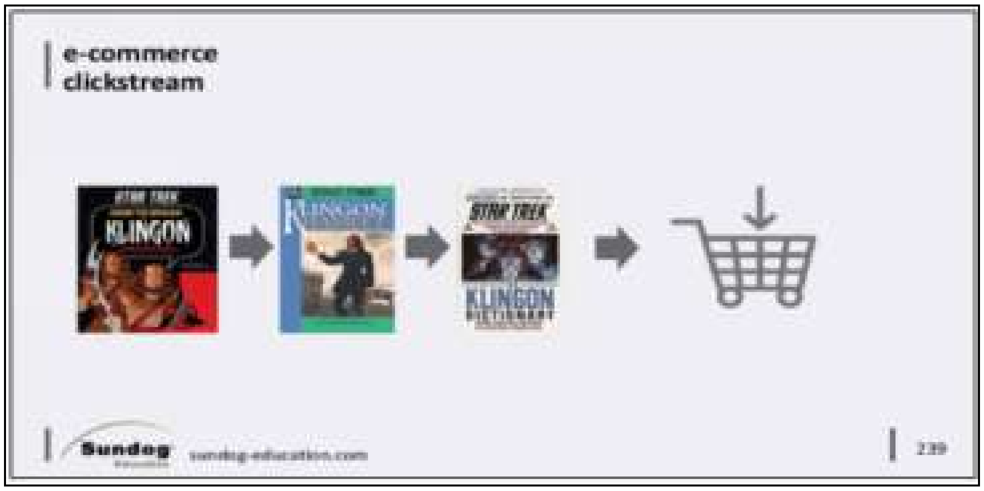
Using RNN's for Session-Based Recommendations

session-based recommendations with rnn's

Sundog sundog-education.com 238

If you remember back to our deep learning introduction, recurrent neural networks, or rnn's, are good at finding patterns in sequences of data, and predicting what comes next.

Well, some recommendation problems can be thought of in those terms. We call them session-based recommendations. Let's give you a couple of examples.



e-commerce clickstream

Sundog sundog-education.com 239

Let's say you're shopping on Amazon or something. If you're not logged in, then making recommendations to you is really hard! We don’t know who you are, so we don’t know what your past history is on the site or what your past interests are. We can keep track of your session though as you browse the site using the same browser window.

Perhaps you’ve decided to learn how to speak Klingon, because for some reason you’ve decided that would be a useful life skill. In fact it’s not, but let’s just make it interesting. Perhaps the first book you look at is “How to Speak Klingon”, so that’s the first thing in your clickstream .

So at this point literally the only thing our recommender system knows about you is that you looked at the book “How to Speak Klingon.” But at least that’s something.

Maybe you didn’t like the reviews on this book or something, so you moved on to another book you found – Shakespeare’s Hamlet, translated into Klingon (yes, this really exists.) So at this point, we can say your clickstream consists of these two items – it is a sequence of clicks, one on “How to Speak Klingon,” and a second on “Klingon Hamlet.”

If you wanted to make a recommendation at this point, a reasonable thing would be to try and predict what someone who just viewed these two items, in this order, might look at next. That is, you’d like to make recommendations based on the sequence of things you’ve looked at previously. Predicting sequences is what RNN’s do, which is why they are a natural fit for this sort of problem.

Perhaps the most likely thing for a user to click on next is “The Klingon Dictionary,” which is of course the definitive reference for the Klingon language. Maybe the user clicked on that because our RNN recommended it, and hopefully they then purchase that item and move on with their life. So this particular session is over, and this session’s clickstream consists of these three books.

There is a dataset called the RecSys Challenge 2015 dataset that contains data exactly like this you can play around with.

Another application of session-based recommendations based on clickstreams is streaming video sites, such as YouTube or Vimeo. Again, if you're not logged in, all we know is the sequence of videos you watch – but we can still use that sequence to recommend what you should watch next. That's the sort of thing RNN's are for.

Maybe you're watching videos from Sundog Education's amazing YouTube channel, because who doesn't? Anyhow, the sequence of videos you watch from our channel can all be considered part of a "clickstream" as well, and the problem of recommending what to watch next based on the videos you watched previously in your session works exactly the same way as the e-commerce clickstream problem.

the paper

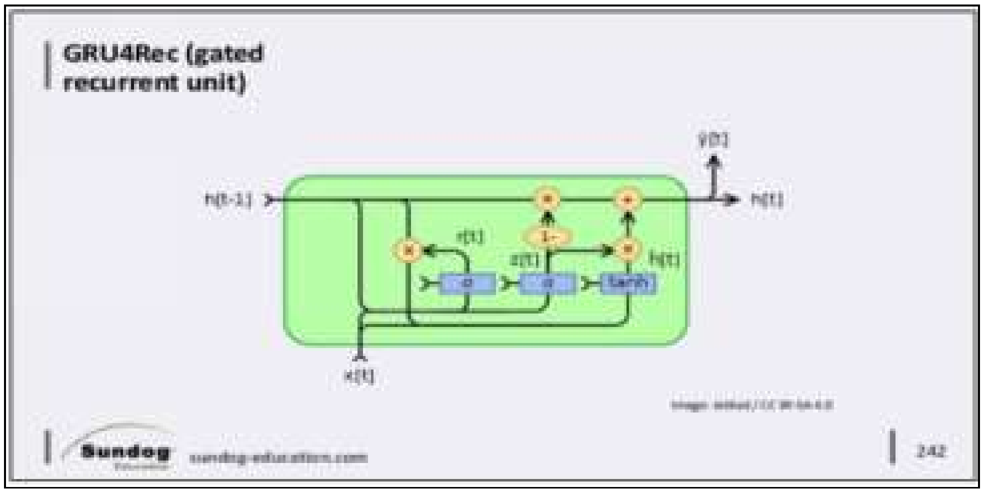
SESSION-BASED RECOMMENDATIONS WITH RECURRENT NEURAL NETWORKS

[illegible]

Sundog sundog-education.com 241

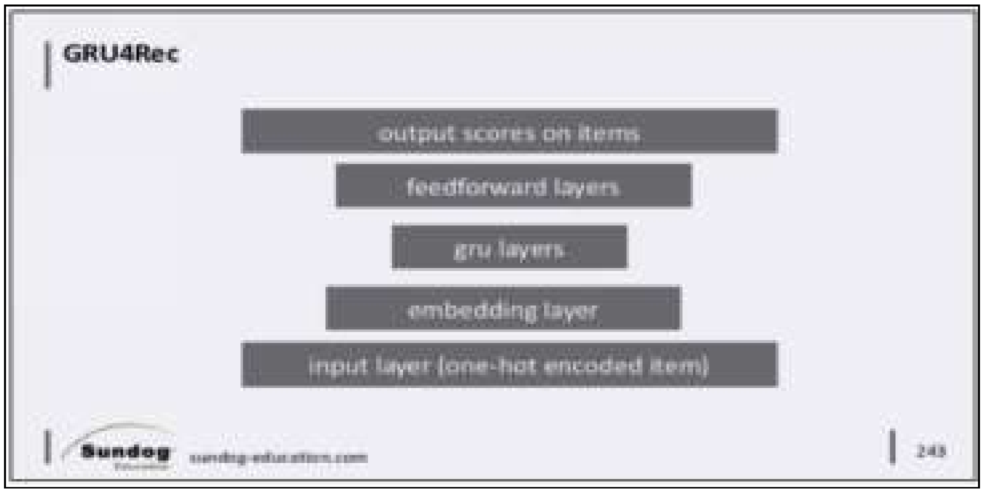
The paper that describes using RNN’s for making session-based recommendations has the creative title of “session-based recommendations with recurrent neural networks.” You might notice Netflix in the list of authors, but the author actually took a job at Netflix in between producing this work and when it was published – Netflix didn’t really have anything to do with it, but it’s a good example of how doing interesting research in recommender systems can land you a job in a company you love. You can find a legitimate free copy of this paper on the Internet if you want to go into more depth.

This paper is just from 2016, and I considered making it a “bleeding edge” alert topic because it is so new – but really, it’s just some minor twists on RNN’s which are by now a proven technology. The world of deep learning is moving so quickly, that research from 2016 really can’t be considered bleeding edge, or even leading edge at this point. We’re in exciting times here.

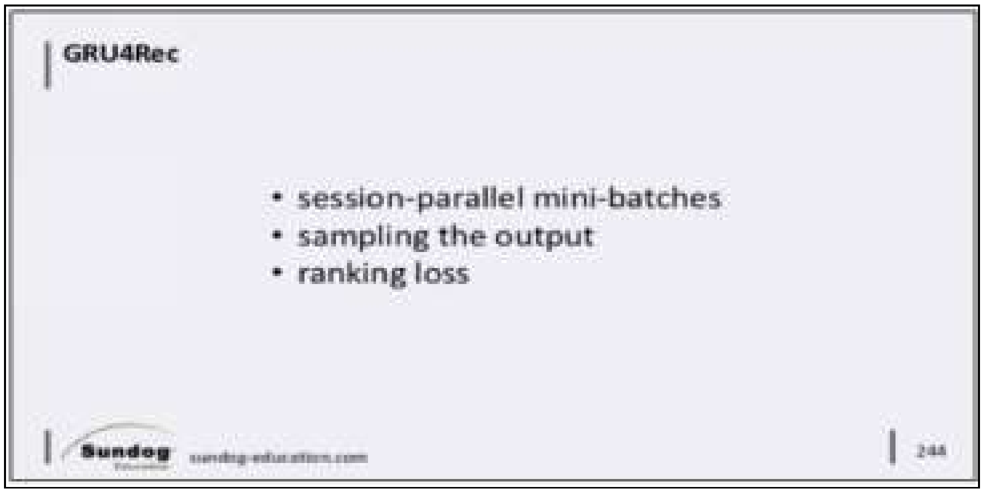


Mostly, the paper concerns itself with how to tweak RNN's to work well with session-based clickstream data. If you remember, RNN's are rather complicated beasts – instead of simple neurons, they depend on more complex structures such as LSTM, or in our case GRU's – gated recurrent units. This gives the network an internal hidden state within each unit that must be maintained. GRU gates learn when and by how much to update that hidden state within each unit.

Because the paper relies on GRU's with some customizations for the recommendation problem, its technique is sometimes referred to as GRU4Rec.



A simplified view of the architecture looks like this – this represents the processing of a single event in the clickstream. We start with the item that was just viewed encoded as 1-of-N, and that goes into an embedding layer, and that in turn leads into multiple GRU layers. We've represented it as a single block here in the interest of space, but in practice there can be many GRU layers in the middle here. There can also be many feed-forward layers after the GRU layers – these are just more traditional neural network layers that don't involve fancy GRU's. And finally we get scores on the all of the items, from which we can select the items the deep network thinks is most likely to be viewed next in the clickstream.



GRU4Rec

- session-parallel mini-batches
- sampling the output
- ranking loss

Sundog sundog-education.com

244

There are a few other twists on RNN's that the authors made for the recommender use case. They feed data into the network in a different way than you would, for example, feed in words from a sentence. Basically, multiple sessions are grouped together, and all of their first items are fed in together, then all of their second items, and so forth. They call this "session-parallel mini-batches."

They also sample the output, because in most recommender systems there are a large number of items, and it's computationally prohibitive to look at every item every time when you want a recommendation right now, based on what the user just clicked on.

Finally, they experiment with specific loss functions for training, and settled on two of them: Bayesian Personalized Ranking, or BPR for short, and TOP1, which is a new loss function the authors developed specifically for this application.

Is it overly complex?

Sundog sundog-education.com 245

Now, one thing I’ve learned repeatedly in my career is that simpler solutions tend to be better. And GRU4Rec is far from simple. Before I would adopt a technology like this, I would really have to be convinced that it offers substantially better results than simpler alternatives do.

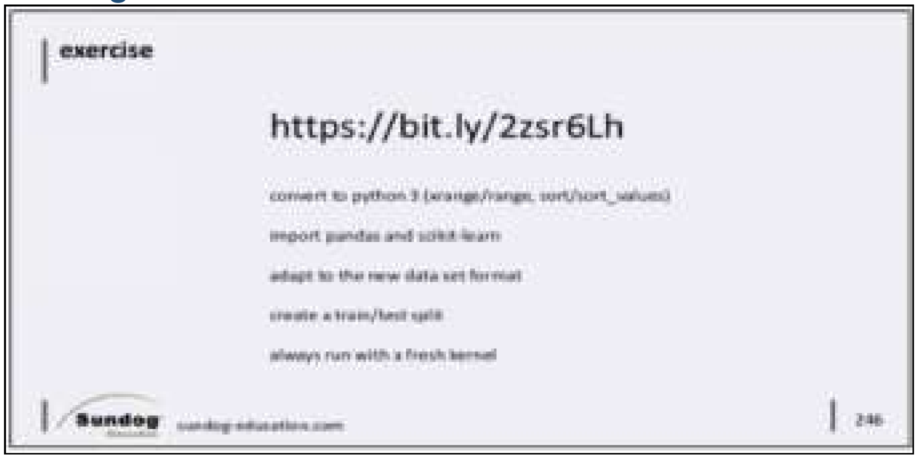
There are other ways of making recommendations based on session data; this problem was around long before RNN’s were. You can just treat the items in someone’s session just like somebody’s set of past movie ratings or whatever, and use item-based KNN or collaborative filtering to recommend things that are similar to the stuff they clicked on already. That doesn’t take the order of the clicks into account, but in practice that information isn’t always that important.

And if you dig into the paper’s results – they were able to get slightly better results from RNN’s than they did from item-based KNN, but it wasn’t a dramatic difference. Nor was it evaluated with online, real people, so more experiments would have to be run to say if it’s really a superior approach or not .

There is a real cost to deploying something like this into a production system. If it starts producing bad recommendations, who will understand the algorithm well enough to be able to tune it and correct it? How much hardware will it take to power a system this large and complex, and what happens when that hardware breaks? How long does it take to produce recommendations for a given session, and how does that affect the page render time for the website it's integrated into? That last concern could be enough to kill the project in itself.

My point is that systems like this are definitely worth learning about and evaluating, especially since deep learning seems to be where all the current research is heading and we want to take advantage of that. But there are practical considerations that must be made, and a complex system should never replace a simpler one that works just as well. You're the one that will be woken up at 2 AM when something goes wrong with it, remember.

Coding Exercise



exercise

https://bit.ly/2zsr6Lh

convert to python 3 (xrange/range, sort/sort_values)

import pandas and scikit-learn

adapt to the new data set format

create a train/test split

always run with a fresh kernel

Sundog sundog-education.com

246

GRU RNN's are complicated beasts, and building a GRU-based recurrent neural network is actually a very complicated task as well. It would be crazy to try and write it from scratch if you don't have to. As with most scientific progress, you want to build upon the work of others whenever you can – just don't cross the line of stealing someone else's work and calling it your own.

So your challenge is to do exactly that - build upon the work of others, and get a GRU RNN running on our MovieLens data. The authors of the paper published code along with it, but it was written in Theano – which is kind of like Tensorflow, but Tensorflow is what ultimately took off, not Theano. Even though the paper is only a couple of years old, a lot has changed in that time! So if you were to adapt the ideas in this paper to Tensorflow, you might have to learn Theano first and port everything over to Tensorflow.

But, it turns out someone else has done that already too – and that's what this bit.ly link will lead you to. It's a GitHub project by a bright Chinese researcher named Dr. Weiping Song. It's very close to what we need to run it within our Anaconda environment with a modern Python and Tensorflow installation – but even though this project is less than a year old, it's already out of date!

So, your challenge is to download a copy of this project from GitHub, and get it running. To do so, you'll need to take care of a few things:

The first problem is that it's written for Python 2.7, but the world has since moved on to Python 3. There will be some minor syntactical things you'll need to address just to get the code to compile, such as replacing calls to xrange with range, and replacing sort with sort_values.

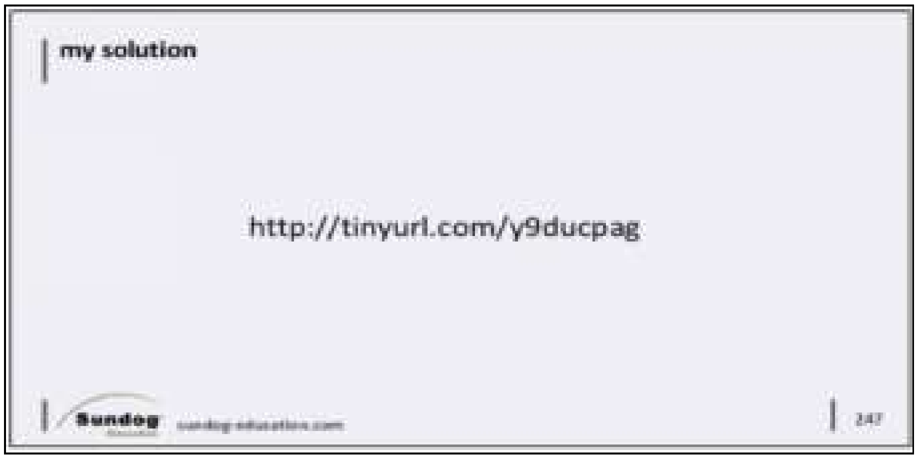
This project also uses the Pandas and scikit-learn libraries, which you might not have installed yet if you skipped the "Intro to Deep Learning" section. So, you may need to import the Pandas and scikit-learn module into your RecSys environment using Anaconda Navigator in order for this code to run.

Next, you need to modify the code that reads in the MovieLens data, because it was written for a different data set with a different format. You'll need to change the column names, paths, and separators. Pay particular attention to the column names, because they crop up in several places within the code.

The code assumes you already have the data split out into training and test sets as separate files, and we don't. So you'll also need to modify this code to create a train/test split on the MovieLens data when it runs.

As you're working, take care to create new kernels each time you run the code. Close out the output console window in Spyder, and use a fresh one every time you run it. Otherwise, you'll encounter some very confusing error messages that are just a result of having state left over from the previous run in memory.

Have a go at it, and see if you can get it running. In the next section, I'll give you a peek at my own results.


my solution

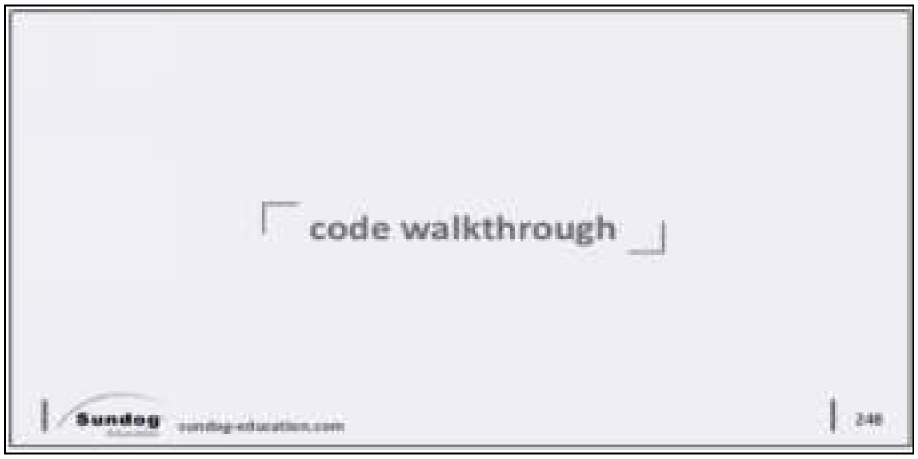
http://tinyurl.com/y9ducpag

Sundog
sundog-education.com
247

Did you get it running? I hope so – really, a lot of what you do in the real world is trying to get algorithms you read about up and running within the confines of your own development environment, so I assure you this exercise wasn't just busy work. It's a taste of what you can expect to be doing within an industry job. And in the real world, this would just be the beginning – the next step would be to integrate it into whatever systems actually provide data and produce recommendations within your organization so you can actually test it in a real-world setting.

I should also remind you that just because code is on GitHub doesn't mean that it's free for you to use however you wish. I wouldn't just take this code and put it into a commercial system without making sure that it was offered under a license that allows for that. If it isn't, then all you can do is experiment with it and learn from it – but your own implementation in a production system would have to be re-written from scratch. Or better yet, offer the guy who wrote the code you want to use a job!

Anyhow, let's take a quick look at the modifications I made and see it run.


code walkthrough

Sundog sundog-education.com

248

So if you decompress the zip file from the URL on the previous slide, you'll get my adaptation of GRU4Rec that works with the MovieLens 100K dataset, and with the latest version of Python.

Let's take a step back, first, and think about what this code is actually doing. GRU4Rec applies a recurrent neural network to session-based clickstream data, and tries to recommend things you're likely to click on next. But, the MovieLens data isn't really clickstream data – we're just pretending it is for our purposes. The ratings data in MovieLens includes timestamps, so we actually can order ratings from a given user into a sequence – and pretend that the ratings are actually click events, and not explicit ratings. As we'll see, the algorithm still works surprisingly well even though the data isn't really what the algorithm envisioned working with. It helps that the MovieLens data set only includes users who rated a large number of items, so every user has a fair amount of fake clickstream data for the algorithm to work with.

Anyhow, the first thing I had to do was install the pandas and scikit-learn packages into my RecSys environment in

Anaconda. I did that by going to Anaconda Navigator, selecting my RecSys environment, selecting uninstalled packages, and adding them from there.

The next order of business was porting everything from Python 2.7. For the most part this was straightforward, but the real lesson here is that Google is your friend. I didn't cover all the specific ways in which Python 3 differs from Python 2.7 in this course, but you simply can't expect to be given step by step instructions on how to solve every problem you face in a technical field. Systems are too complex, and they change too often, for you to have a cookbook to solve every problem. When you run into a weird compilation error, you have to look up whatever command didn't compile under Python 3, and look up what changed from Python 2.7 regarding it. Part of the point of this exercise was to force you to do that, and be resourceful in finding information on your own when you need it. I firmly believe that resourcefulness and perseverance are two qualities that are most important to a good software developer.

For example, if we take a look at model.py here… one specific issue was that the original code used the "xrange" function a lot. I had to search for every occurrence of xrange and replace it with range, for example here on line 113 and down here on line 182. I also had to replace the sort command with sort_values throughout the code, for example here on line 169.

Let's go to main.py. The next order of business was adapting the code to the ratings data we have at hand with the MovieLens 100K data. Starting at line 17, you can see the original code had separate train and test data files that contained retail click data from the Recsys Challenge 2015 data set. We don't have separate train and test files for MovieLens, so I just redirected the training data file to my ratings.csv file from MovieLens instead, and got rid of the test file path. Obviously this path will be different for you, depending on where you stored it on your system. Skipping down to line 62, I load up our training data, telling it about the column name for its item identifiers in the process so we can make sure it's in the type the code expects. Then I split it into train and test data sets by just splitting off all of the ratings after row 90,000 and making that the test, or validation, data.

Skipping back up to line 20, the Args() function also had to be modified with the column names in my data set, so the session key, item key, and time key all had to be changed to reflect the data files we have. Unfortunately these variables aren't always used in the rest of the code, so there are a few other places where the column names had to be changed throughout the code as well.

While testing things out, I learned the hard way that you really need to run it on a fresh kernel every time. Closing out the console window and allowing a new one to appear between each run does the trick. The errors you receive make absolutely no sense if you don't do this. Also, you'll find that the code stores state data in a checkpoint and __pycache__ folder inside your project – when all else fails, you can try deleting these folders to force it to start from a clean slate instead of restoring a previously built model.

Overall though, it wasn't too hard to get it up and running – and it's certainly a quicker way to start experimenting with GRU4Rec than trying to write it yourself from the ground up. Let's run it – it goes pretty quickly.

Unfortunately the output isn't terribly interesting – we don't get to see any examples of actual recommendations, only the measurements of the final results. But to be fair, it would be hard to subjectively evaluate the results given that we're using a data set that wasn't really intended for this sort of thing to begin with.

They give us a result for "Recall@20" – this is basically hit rate looking at the top-20 results. So, I like that they are focusing on hit rate as a metric. This number is small, but at least it's not zero – and given the small size of our data set and the fact that it isn't really session-based data to begin with, I think it's impressive that it's as high as it is. In the paper, scores as high as 0.66 were achieved, which is pretty darn amazing. They were able to accurately predict a click within their top 20 results 2/3 of the time! MRR@20 is what we've called ARHR in this course – it stands for Mean Reciprocal Rank, looking at the top 20 results, which is just another way of saying Average Reciprocal Hit Rank. In the paper, this score was as high as 0.39. It's worth noting, however, that Item-based KNN also produced impressive results, so the difference between what we're seeing in the paper and what we're seeing on our desktop has more to do with the data we're using to train and test the algorithm with than anything.

Bleeding Edge Alert! Deep Factorization Machines

It's time for another bleeding edge alert! This is where we talk about some recent research that has promising results, but hasn't yet made it into the mainstream with recommender systems yet.

deep factorization machines

Sundog sundog-education.com 250

If you remember back to the matrix factorization section of this course, we mentioned factorization machines toward the end of it. They are a more general form of matrix factorization, and can be designed to do the same thing as SVD as an example. But they take in categorical data, and can find latent relationships between any combination of the features they are given. As such, factorization machines are more general purpose than SVD, and can sometimes find relationships between features that SVD wouldn't have considered.

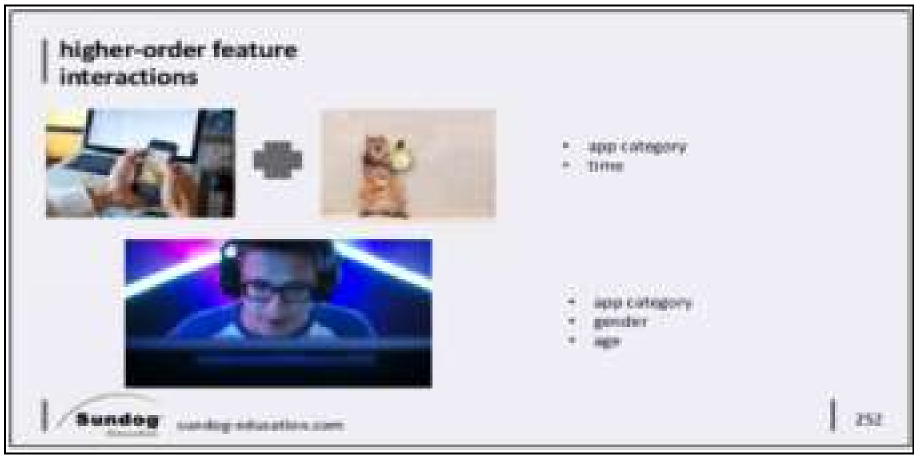
The idea of deep factorization machines is to combine the power of factorization machines, with the power of deep neural networks, to create an even more powerful recommender system. This hybrid approach outperforms factorization machines or neural networks used individually, albeit not by a huge margin. This research is just from 2017, so it hasn't really been proven in large-scale, production use yet that we know of.

the paper

DeepFM: A Factorization-Machine based Neural Network for CTR Prediction

Sundog Education sundog-education.com 251

I’ll give you a brief overview of how it works, but if you want all the details, hunt down the paper – it’s called “DeepFM: A Factorization-Machine based Neural Network for CTR Prediction” from a team at the Harbin Institute of Technology in China. It’s also easy to find legitimate, free copies of the paper online.


higher-order feature interactions

- app category
- time

- app category
- gender
- age

Sundog Education sundog-education.com 252

The motivation behind DeepFM is to combine the strengths of factorization machines and deep neural networks. While factorization machines can uncover what we call higher-order feature interactions, they’re best at lower-order feature interactions – but deep neural networks excel at higher-order interactions. DeepFM promises us the best of both worlds.

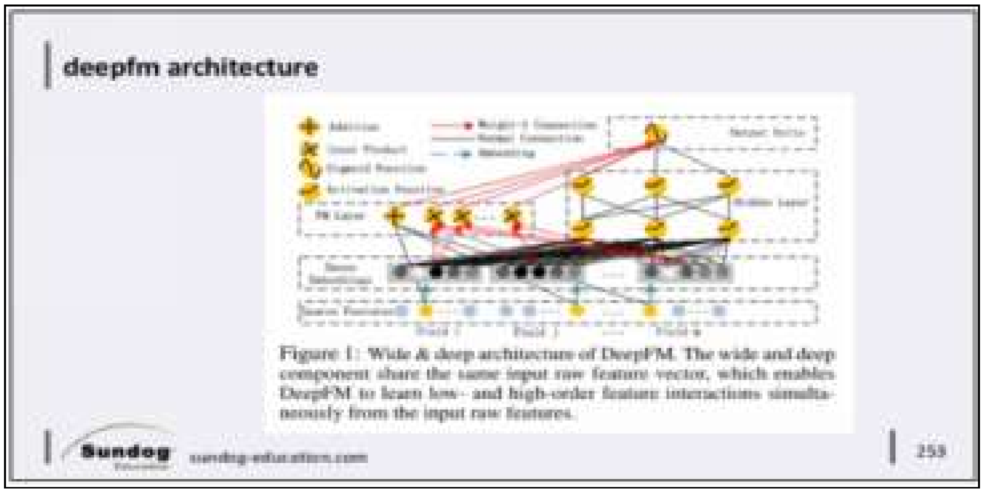
What do we mean by high or low order feature interactions? Let’s look at a couple of examples given in the paper. Imagine we're building a recommender system for an app store that recommends which app a user should download right now.

A food delivery app for example, might be more popular in the early evening, when it's dinner time. In this example, the features involved are order 2: there is a latent relationship between application category (food delivery apps,) and the user's local time of day. Factorizations of the user data that split out these features would work well.

Let's also consider the case of a male teenager. Perhaps the data supports a latent relationship between being male, being a teenager, and liking first-person-shooter games. In this case, we have order-3 feature interactions – the three features involved are the application category (shooter games,) the user's gender, and the user's age.

The food delivery app is an example of where a factorization machine may do well, and the shooter game is a case where a deep learning network might do better. DeepFM tries to combine both approaches, so we can get good results in both low and high order cases.

As an aside, the idea of an algorithm recommending shooter games to you just because you're a male teenager may rub some people the wrong way. And this is in fact an example of how recommender systems can have unintended consequences – this system would continue to perpetuate the stereotype of young males enjoying violent content, and encourage it to continue by putting more violent content in front of young men. That's a much bigger ethical discussion that I'd love to have, but in short it is important to remember that recommender algorithms can perpetuate the same biases that exist in the data they were trained with, and even amplify them. It's something to be aware of and watch out for.



deepfm architecture

Figure 1: Wide & deep architecture of DeepFM. The wide and deep component share the same input raw feature vector, which enables DeepFM to learn low- and high-order feature interactions simultaneously from the input raw features.

Sundog sundog-education.com

253

This figure from the paper gives an overview of their architecture. You can see it's a fairly straightforward hybrid approach; the same feature data from your training set is fed in parallel to both a factorization machine, and to a deep neural network – in this example, with two hidden layers. The outputs from the factorization machine and the deep neural network are combined into the final sigmoid unit that predicts a click, purchase, or whatever action you're trying to predict.

You can see visually here that the factorization machine is "wide", and as such is adept at finding lower-order feature relationships, while the neural network is "deep", and adept at finding higher-order relationships. The idea is that they are better together, but of course how much so depends on the nature of your data and what sort of latent relationships between features really exist in it.

an ensemble approach

Sundog
sundog-education.com
254

DeepFM may strike you as an overly complex approach, but it’s really just an example of an ensemble approach to recommendations. It’s combining together two different algorithms that are powerful on their own: factorization machines, and deep neural networks. Think of them as two different instruments that sound even better when they’re together!

You can create a complex system by combining together many simpler ones, and usually the end result is better than using any individual algorithm in isolation. Ensemble systems, when designed properly, can make that complexity manageable by keeping its component systems largely isolated and running in parallel, and just combining their results in the end. That way, if any one component goes haywire, you can just disable it and the system will keep on running using the other components.

So is DeepFM really bleeding edge? Well, it’s actually just combining two technologies that have been proven, so this gives me more confidence in the idea. Whether it produces meaningful gains in real world scenarios though remains to be shown, but I think it’s worth experimenting with. Remember the Netflix Prize was won with a very similar hybrid approach. Netflix learned that combining SVD with RBM worked really well, and DeepFM is really just taking that up a notch – it replaces SVD with the more general Factorization Machine model, and RBM’s with a more general deep neural network. It’s really the same concept as one that has been proven at Netflix, which makes DeepFM all the more appealing.

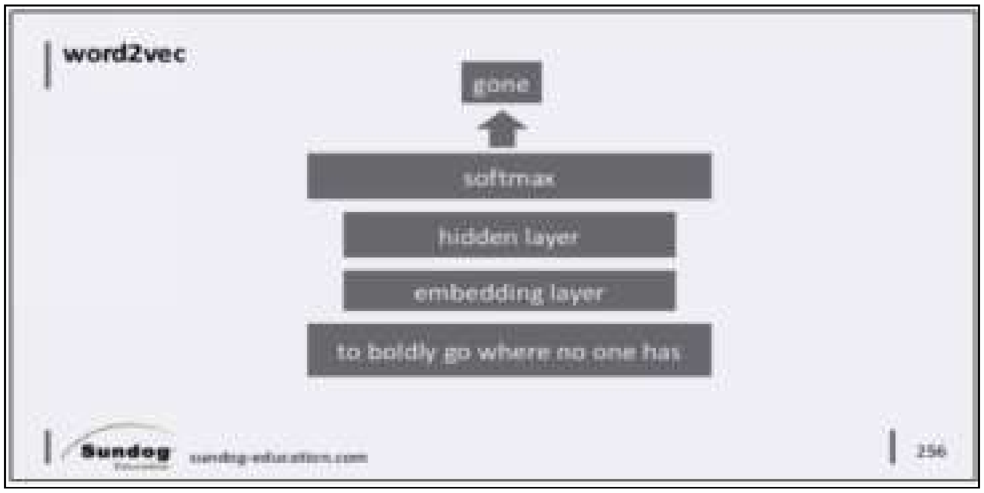
So keep an eye out for DeepFM, and if you’re already an industry researcher, give it a try.

more technologies to watch

Sundog sundog-education.com 255

While we're still operating under our "bleeding edge alert," let's briefly cover a few more recent research topics that are worth keeping an eye on.

Word2Vec


word2vec

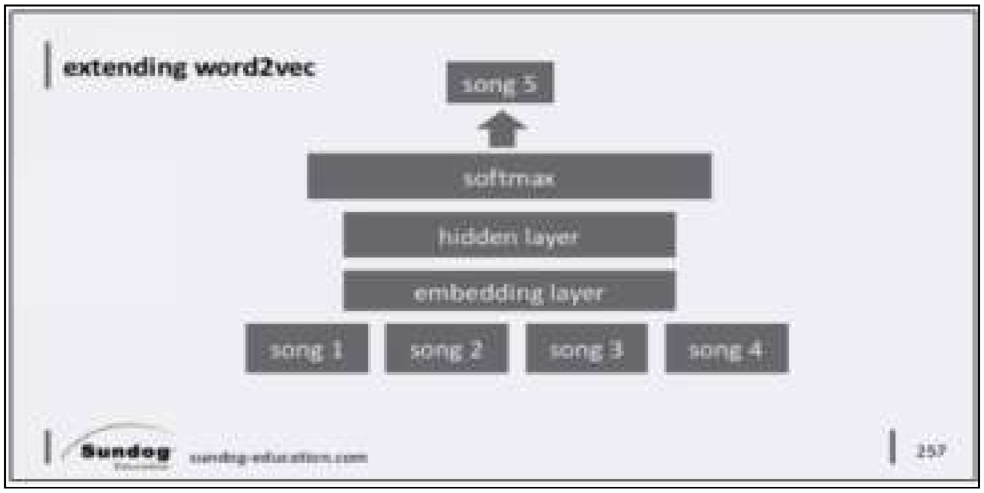
Sundog sundog-education.com 256

Some people have applied a neural network model called word2vec to recommender problems. If you have text data to work with, it can be an interesting tool.

The idea is you start with some words that represent the context or user history you want to use to inform your recommendations, but to make it easier to understand let's just start with the problem of completing someone's sentence. We can encode all of the words in that sentence, feed it through an embedding layer and a hidden layer where the learning takes place, and run a softmax classifier on the output of the hidden layer to find the probabilities of each word that might be associated with the ones you fed in as input. In this example, we feed in signals associated with the words “to boldly go where no one has”, and a properly trained word2vec network might classify the word “gone” as the most likely word associated with those words.

An example of applying this technique to a recommender system would be recommending menu items to restaurant owners, based on the menus of other restaurant owners. Perhaps you could train word2vec with a large number of menus and the words that appear on them. Then, by feeding in the words in a specific restaurant's menu, you could recommend terms associated with other dishes they might want to offer on their menu.

You might observe that this is also the sort of problem recurrent neural networks, or RNN’s, are made for, and you would be correct. There is also research in using RNN’s for text-based recommendations, like the menu item recommender we just mentioned.

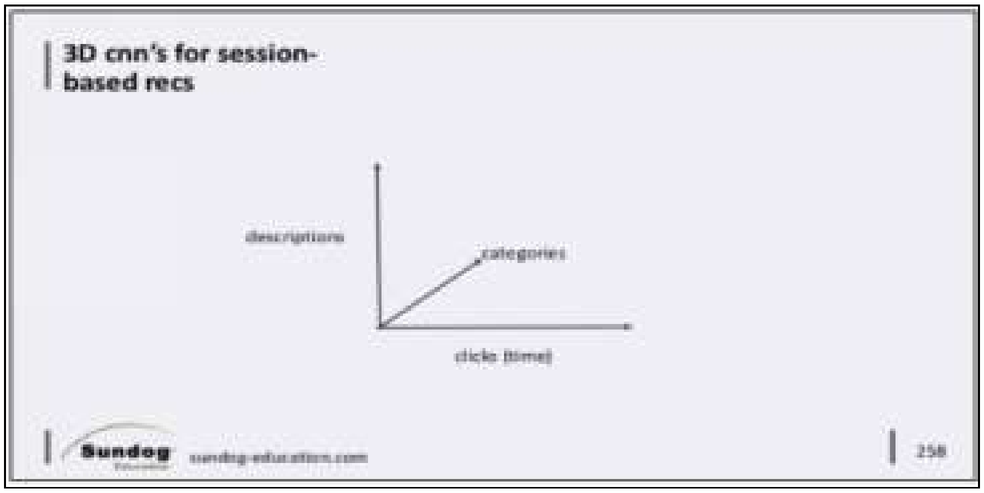


Things get interesting when you use word2vec with things that aren't words at all. In one interesting example, a streaming music service called Anghami used word2vec, but applied it to the problem of recommending songs based on a user's previous choices in their stream.

It works exactly the same way as it does with words in sentences, but instead of working with words in sentences, it works with individual songs that are part of someone's stream. The output isn't a prediction of words, but of other songs that fit into the "sentence" of that user's past stream of songs.

This is a really neat example of taking an established idea, and getting an entirely new system by just questioning the assumptions about what it's for .

3D CNN's


3D cnn's for session-based recs

Sundog sundog-education.com 258

An interesting paper in the 2017 RecSys conference proposed used 3D convolutional neural networks for session-based recommendations. We talked about using recurrent neural networks to produce session-based recommendations; this idea is basically adding two more dimensions that can be used to consider feature data about the items being recommended in addition to the data on how they were consumed in peoples' clickstreams.

All of the feature data is character-encoded, and then we treat it just like we would treat image data when using a CNN and trying to do image recognition.

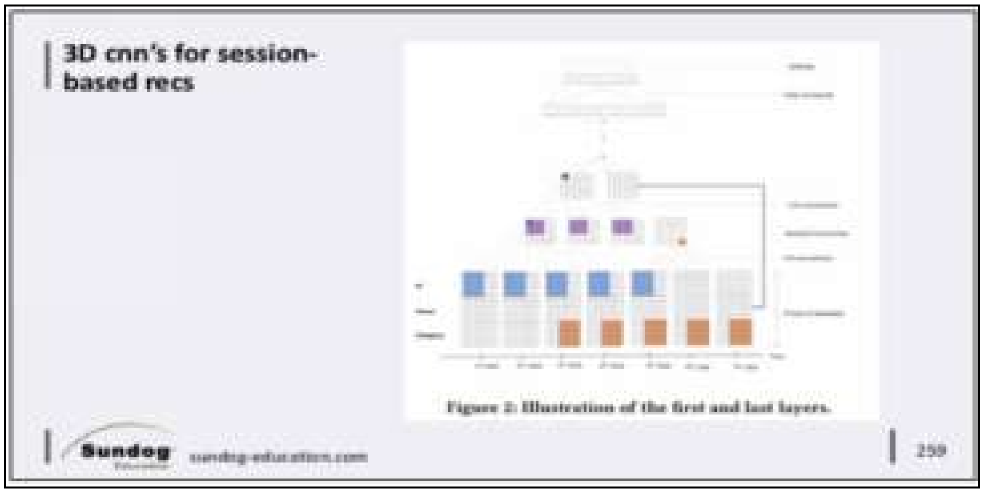
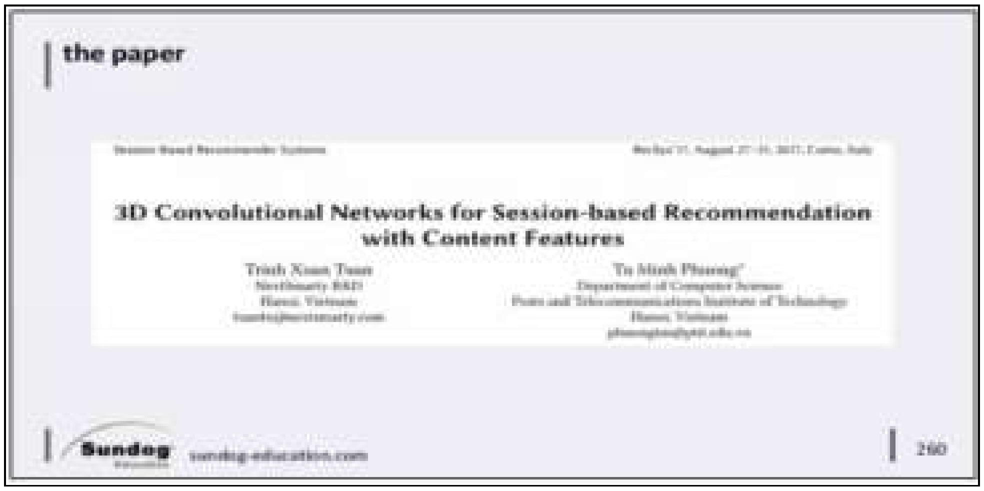

3D cnn's for session-based recs

Figure 2: Illustration of the first and last layers.

Sundog sundog-education.com 259

As you can imagine, this gets very complicated, very quickly. The one saving grace is that CNN's themselves are in widespread use these days in other fields, so the complexity of the CNN itself is sort of a solved problem. There are lots of little twists involved with applying it to clickstream sessions, however, such as how to deal with sessions of different lengths. CNN's want to work with blocks of data that are all the same size, which presents a problem the paper has to address.

Complexity aside, it's hard to argue with their results – they did out-perform RNN's pretty handily in their paper, but we have to remember this is a solution for a very specific kind of problem – recommending items in a clickstream session, taking content attributes into account at the same time.



the paper

Session-based Recommender Systems RecSys'17, August 27–31, 2017, Como, Italy

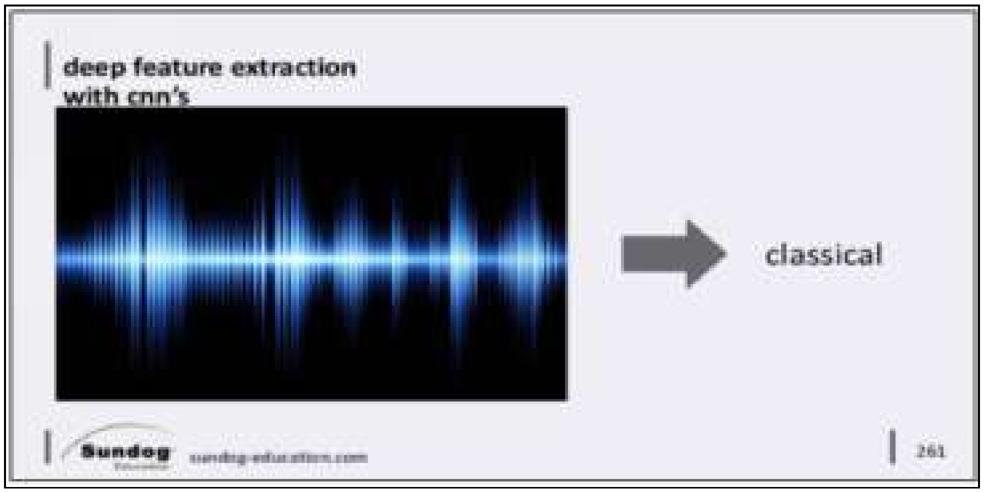
3D Convolutional Networks for Session-based Recommendation with Content Features

Trinh Xuan Tuan
Next[illegible] R&D
Hanoi, Vietnam
[illegible]

Tu Minh Phuong*
Department of Computer Science
Posts and Telecommunications Institute of Technology
Hanoi, Vietnam
[illegible]

Sundog sundog-education.com 260

The paper for this idea is also available for free online if you'd like to learn more. Again, it's just something to keep in the back of your head for now – you're not going to find examples of how to code this up online yet, because it's just too new.



deep feature extraction with cnn's

classical

Sundog

sundog-education.com

261

A more straightforward application of cnn's in the world of recommender systems is using them to extract feature data from the things themselves that you are trying to recommend.

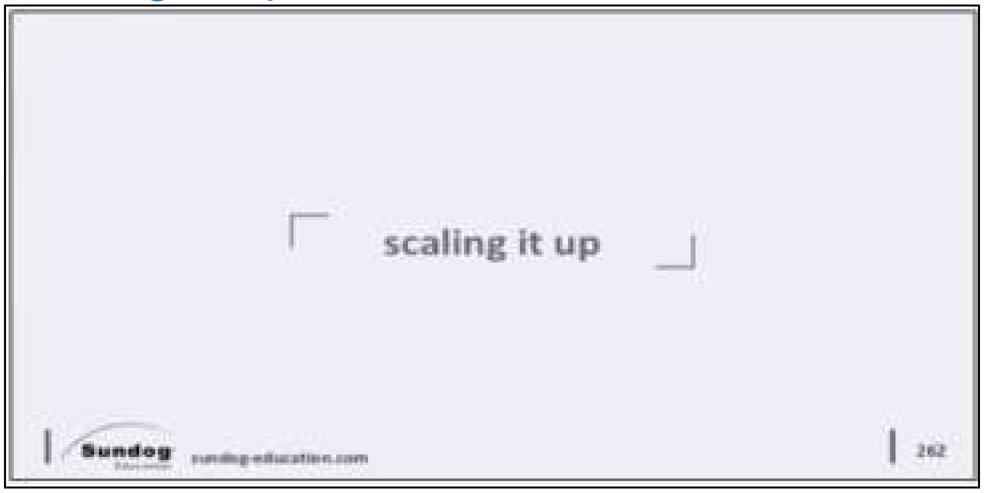
For example, just like you can use a convolutional neural network to classify an image as, say, a duck, you can also use a CNN to classify an audio waveform as a specific musical genre. If you're building a music recommendation system, you might not have good metadata about the songs you need to recommend – but you could use a CNN to extract that metadata, such as the genre, automatically. That genre information could then work its way into a content-based recommendation scheme of some sort, that might augment a collaborative filtering or matrix factorization model.

You could even imagine using a CNN to classify paintings, and recommend paintings to people based on the types of paintings they've liked in the past.

You still need to train your CNN's, so you can't use them to create new classifications it's never seen before – but it can be a useful tool in filling in attribute information for new items, or items with missing data.

That wraps up our bleeding edge alert on deep learning in recommender systems, but research is progressing quickly on this front – by the time you watch this, some of these ideas may already be mainstream! We know YouTube and Amazon already have systems in production using deep learning for recommendations, and are starting to publish their learnings from it. It’s exciting times in the field of recommender systems – but remember, don’t get caught up in the hype just for the sake of it – for many problems, older, simpler solutions are still a better fit.

Scaling it Up


scaling it up

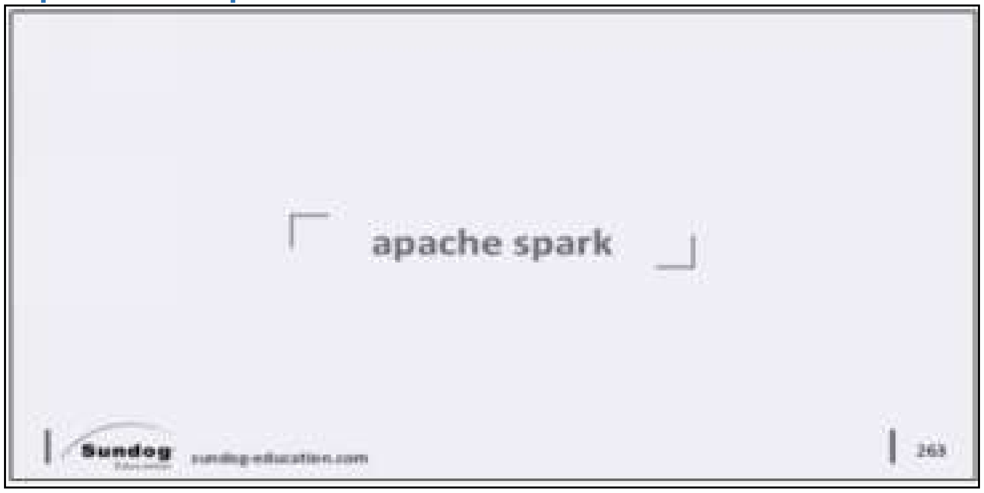
Sundog sundog-education.com 262

So far, we’ve been using the 100,000 ratings data set for MovieLens as we illustrate different recommender systems. That’s good enough for educational purposes, but if you want to build a recommender system for a real company, you will probably be working with much more data – and you certainly won’t be processing it just on your own desktop PC.

This next section is about scaling it up – systems that exist to let you train recommender systems with real “big data”, on a cluster, maybe even in the cloud.

There are lots of options, but we’ll cover a few of my favorites .

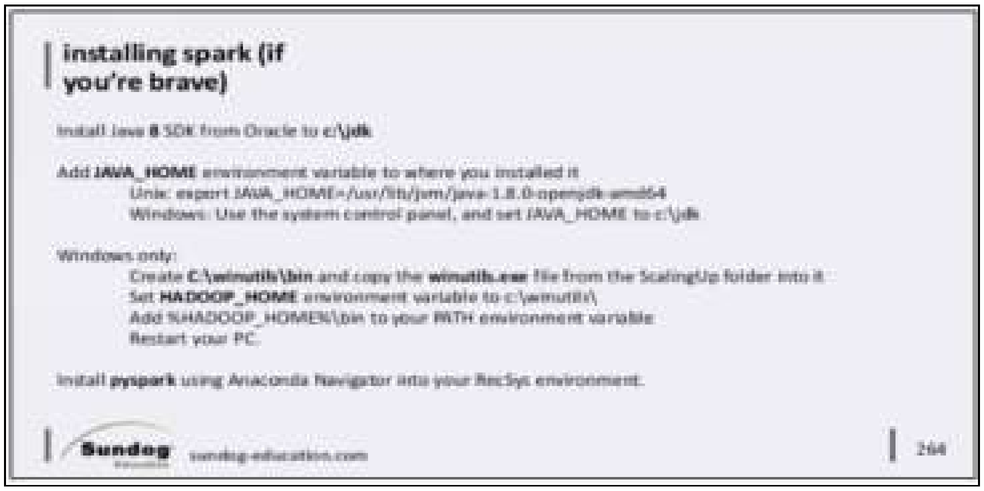
Apache Spark and MLLib


apache spark

Sundog sundog-education.com 263

You may have heard of Apache Spark – it's a very powerful framework for processing massive data sets across a cluster of computers. As your data grows, you just keep adding more computers to your cluster – the sky's pretty much the limit!

I'm not going to go into the details of how Spark works here, but Sundog Education offers entire courses on that topic. For now, we'll just focus on it from a practical standpoint – we'll get it installed, and examine some code to generate recommendations using Spark on a larger data set. Since you probably don't have a cluster of your own handy, we'll do still do this on your own desktop PC – but running this code on a cluster running Apache Spark would work exactly the same way. Think of it as running on a cluster of one computer for now, which is your PC.



installing spark (if you're brave)

Install Java **8** SDK from Oracle to **c:\jdk**

Add **JAVA_HOME** environment variable to where you installed it
- Unix: export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
- Windows: Use the system control panel, and set JAVA_HOME to c:\jdk

Windows only:
- Create **C:\winutils\bin** and copy the **winutils.exe** file from the ScalingUp folder into it
- Set **HADOOP_HOME** environment variable to c:\winutils\
- Add %HADOOP_HOME%\bin to your PATH environment variable
- Restart your PC

Install **pyspark** using Anaconda Navigator into your RecSys environment.

Sundog sundog-education.com 264

If you want to follow along with your own PC in this section, first you need to get Spark installed. And on Windows, that's a bit of a hassle – but here are the steps you need to follow, whatever OS you're using. If you're not comfortable with setting environment variables on your OS, you should probably just watch the next few videos because you'll probably run into trouble otherwise. You need to know what you're doing with environment variables for this to all work properly, and there are a lot of little frustrating things that can go wrong.

The first step is to install the Java 8 SDK from Oracle's website – and make sure it is Java 8, not Java 9 or newer! Spark only works with Java 8 right now. When you install it, you need to be sure to install it to a path that does not include any spaces – so on Windows, that means you can't accept the default location under "Program Files" because there is a space between "Program" and "Files." Use a new path, like c:\jdk or something, that's easy to remember and has no spaces in it.

Then you need to set the JAVA_HOME environment variable on your system to the path you installed the Java 8 JDK into. In Windows, environment variables are set using the System control panel – from there, you select advanced system settings, and then you'll see a button to go into your environment variables.

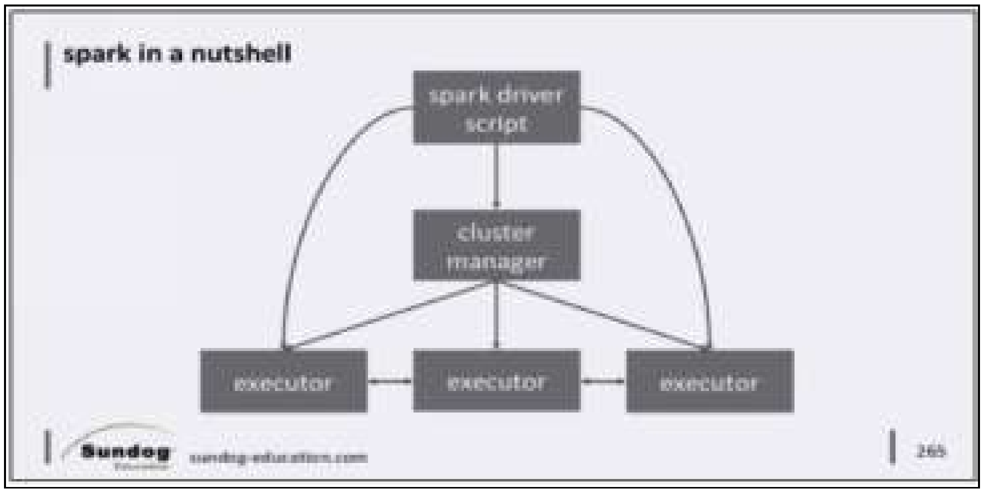
These next steps are specific to Windows – since Spark uses the Hadoop Distributed File System for some things, it won't run unless it thinks Hadoop is installed on your PC. We can fool it by copying the winutils.exe file provided with your course materials in the "ScalingUp" folder into c:\winutils\bin, and then setting the HADOOP_HOME environment variable to c:\winutils.

At this point you should restart your PC to make sure all of those new environment variables have taken effect.

Finally, you need to install the pyspark package into your RecSys environment using Anaconda Navigator.

One more thing – if you already have Spark installed and the SPARK_HOME environment variable set, you need to make sure that you have the same version of Spark installed that the pyspark package installed into Anaconda. Usually this is whatever the latest version is. If you have different Spark versions installed on your machine, you'll get weird errors about the "Java gateway process exiting before sending its port number" that make no sense.

Once you've completed these steps, we can move on and start playing with Apache Spark!



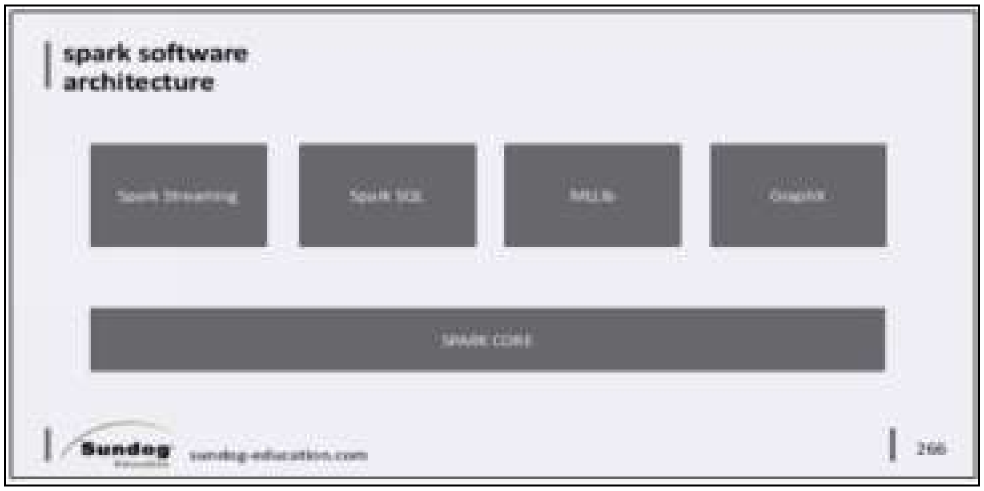
The reason Spark is a big deal is because it's very efficient at distributing the processing of massive data sets across a cluster, and in a reliable manner.

Architecturally it looks like this – you write a driver script, in either Python, Scala, or Java, that defines how you want to process your data, using the API's Spark provides.

Once you launch that driver script, which is usually from the master node of your cluster, it will communicate with your cluster manager to allocate the resources it needs from your cluster. That cluster manager could be Hadoop's YARN cluster manager if you're running Spark on top of a Hadoop cluster, or Spark has its own cluster manager as well if you want to use it instead.

That cluster manager allocates a bunch of executor processes across your cluster that will do the actual work of processing your data. Spark handles all the details of figuring out how to divide your data up and process it in the most efficient manner. It does this by keeping things in memory as much as possible, trying to process data on the actual nodes it is stored on, and using something called a directed acyclic graph to organize the processing in the most efficient way it can.

All of these different components can talk to each other when necessary, which might become necessary if a machine in your cluster goes down or something. It's all rather resilient. For example, even though the cluster manager looks like a single point of failure in this simplified diagram, you can set things up so you have a backup cluster manager ready to take over if necessary.



From a software developer standpoint, Spark consists of a core that manages all the distribution of work that we talked about, and there are other libraries on top of Spark that you generally work with. Spark SQL for example defines something called a DataSet, and this allows you to work with Spark in much the same way as you would with a SQL database. This has become the standard way of using Spark now – DataSets are the future. Spark Streaming allows you to ingest data in real time and process it as it comes in, and it also offers something called Structured Streaming that even lets you treat that real-time data as SQL data. GraphX exists for analyzing and processing data organized in a graph data structure, such as social networks.

What we're most interested in, however, is MLLib – that's Spark's machine learning library. As we'll see, it contains classes that make recommendation generation from massive data sets really simple.

rdd's

resilient

distributed

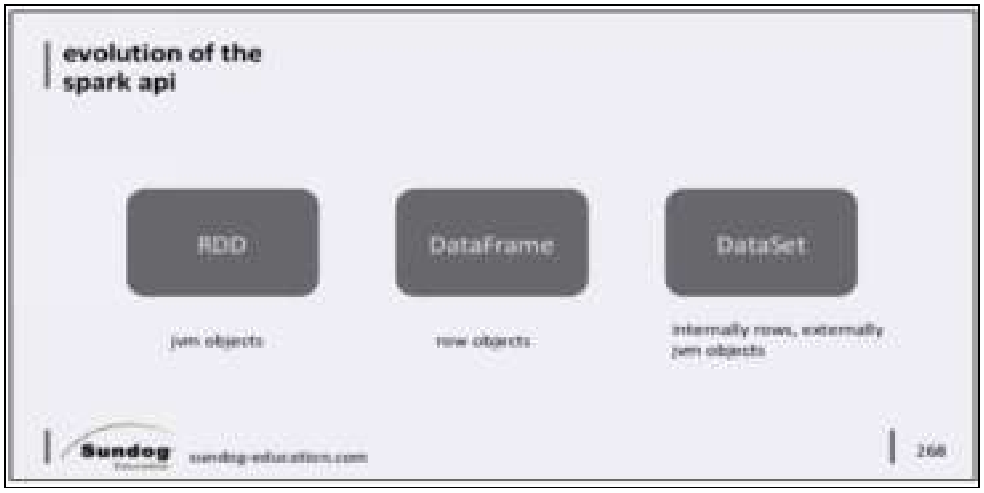
dataset

Sundog sundog-education.com 267

Before we dive into some code, there is one concept you'll need – the RDD. RDD stands for resilient distributed dataset, and it's at the core of Spark if you're not using its SQL interfaces instead.

An RDD is an object that encapsulates the data you want to process. When you write a Spark driver script, what you're really doing is defining operations on RDD's that all tie together somehow, and ultimately lead to some sort of desired output at the end. You define where the RDD should load its data from, what operations and aggregations it should perform on that data, and where to write its output when it's done. Kind of like with Tensorflow, nothing actually happens until you kick things off and request that final output – you're really just allowing Spark to build up a graph of the operations it needs to do to fulfill what you want, and then you start that graph.

From a programming standpoint, this is really nice because you only think about RDD's and the operations you're applying to all of the data within them. The RDD hides all of Spark's complexity in distributing those operations across a cluster, and in handling node failures and things like that. You don't worry about the distributed computing part, you just worry about how you want to transform your data.



evolution of the spark api

RDD — jvm objects

DataFrame — row objects

DataSet — internally rows, externally jvm objects

Sundog sundog-education.com 268

Of course in the world of technology, nothing can stay the same for long – and this is true of Spark as well.

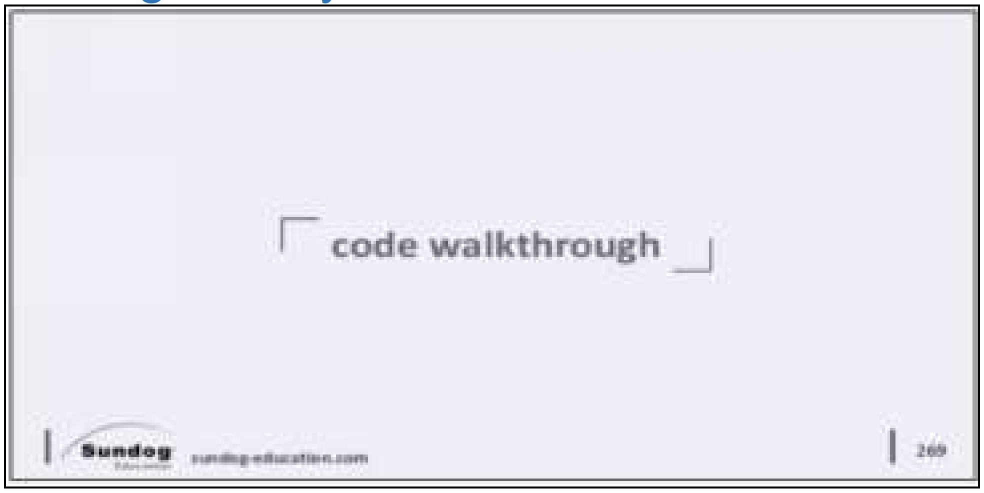
Spark has introduced alternative API’s over time that build upon the RDD. So, while you can still code against RDD’s if you want lower-level control over what your driver script does, DataFrames were introduced that treat the underlying data as Row objects that allow you to code against it like you would a SQL database. You can convert an RDD into a DataFrame if you want to .

Spark 2 introduced the DataSet, which is like a DataFrame except it has better type safety – so you can catch more errors in your driver script at compile time instead of at runtime. Spark 2 uses the DataSet as a common language between all of its components, which also helps to simplify development. You can also convert a DataFrame into a DataSet and vice versa – technically, a DataFrame is exactly the same thing as a DataSet of Row objects. But, there’s a catch – they don’t work in R or Python, so if you’re developing in Python you have to stick with DataFrame objects instead as Spark’s common component. DataSets are also a little bit slower, so some people choose to stick with DataFrames even outside of Python.

This is all kind of confusing, so why am I telling you this? Well, our example in Python will start off using the RDD interface for the initial cleanup and structuring of our input ratings data. But then, we have to convert it to a DataFrame because that’s what Spark’s machine learning library expects as input. DataSets aren’t relevant to us because we’re coding in Python, but it’s a term you’ll hear a lot when people talk about Spark.

So with all that out of the way, let’s take a look at a real Spark driver script and create some real recommendations with it .

Coding Activity



code walkthrough

Sundog sundog-education.com 269

So, open up Spyder inside your RecSys environment, and open the SparkALS.py file in the ScalingUp folder of your course materials.

This is adapted from one of the examples that comes with Apache Spark – they actually use MovieLens as an example as well. But, I made some modifications to get it to work with our specific dataset and to generate top-N recommendations at the end.

It’s surprisingly small, right? One reason Spark is so fun to work with is because you can do really powerful things with very little code. Let’s dive in.

First, we start by importing the packages we need from pyspark itself. As we mentioned, modern Spark scripts use Spark SQL as their primary interface, and that means we need to set up something called a SparkSession for our driver script. It’s similar to a database session in spirit, but we’re not going to use it as a database .

We’re also going to import RegressionEvaluator which will let us measure RMSE on our results, and most importantly ALS from pyspark.ml.recommendation. This allows us to perform matrix factorization using ALS, alternating least squares, as its optimization function. But the amazing thing is that it’s written such that the work can be distributed across a cluster without us even having to think about how it works!

We also import our MovieLens class, just as a convenience to let us print out movie names from movie ID’s.

Now we get into the driver script itself. First, we create a SparkSession object – this contains all of the Spark commands we want to issue. All we’re doing is giving it a name, and letting everything else go with the default settings for the session. But, you could have finer grain control of how Spark distributes the work if you wanted to here.

Next we load up our ratings data into Spark using the read function on our SparkSession, which has handy csv parsing abilities built in. We then call rdd on that, to retrieve a resilient distributed dataset that contains all of that data we can operate on. We’ll store that rdd in a variable called “lines”, since at this point it contains the raw inputs read in directly from ratings.csv.

It’s important to remember that at this point, lines isn’t really a structure that contains all of our data. That would defeat the purpose if we had to read and store everything on a single PC. Creating the lines rdd just creates a placeholder for reading that data in; spark.read didn’t really read in the data right there, it just told the rdd where it will read the data from – but that reading may happen from many different machines in parallel, each reading in their own little piece of the input data that they want to process .

Next, we’ll transform the raw data into something more structured. We call the map function on our lines RDD to transform each individual line using the lambda function provided. This lambda function converts each line into a Row object, that consists of four columns named userId, movieId, rating, and timestamp.

We did this because all modern Spark code operates on DataSets, not RDD's – so we want to convert our RDD into a DataSet. One form of DataSet is a DataFrame, which consists of Row objects. It's a little confusing, but all that's happening here really is that we're converting our data into a form that Spark's ML library can work with – and it wants a DataFrame, not a RDD.

Now that we have our dataframe called ratings, we can split it into a training set and a test set using the randomSplit operation. Again, remember nothing is actually happening yet when this line executes – we're just defining what we want Spark to do, all in a distributed manner, once we request our results.

Now we train our ALS model across a cluster with a whopping two lines of code! As you can see there are a few hyperparameters you can fiddle with if you want to try and improve the results, and you just need to pass in the column names that correspond to the users, items, and ratings. Then it's just a matter of calling fit on it with our training dataframe.

To generate predictions from this trained model, we use the transform function with our test set, and the RegressionEvaluator class lets us easily compute RMSE from the test set by comparing the predicted ratings to the actual ones. We retrieve the RMSE score from it and print it out. As we're finally asking for some output that the driver script needs to retrieve, this is the point where Spark will actually kick off the actual processing now that it can optimize the operations it needs to generate that output.

As a sanity check, we also obtain top-N recommendations since RMSE doesn't tell the whole story. The recommendForAllUsers will generate top-N recommendations for any value of N we want, for everyone – again, in a distributed manner.

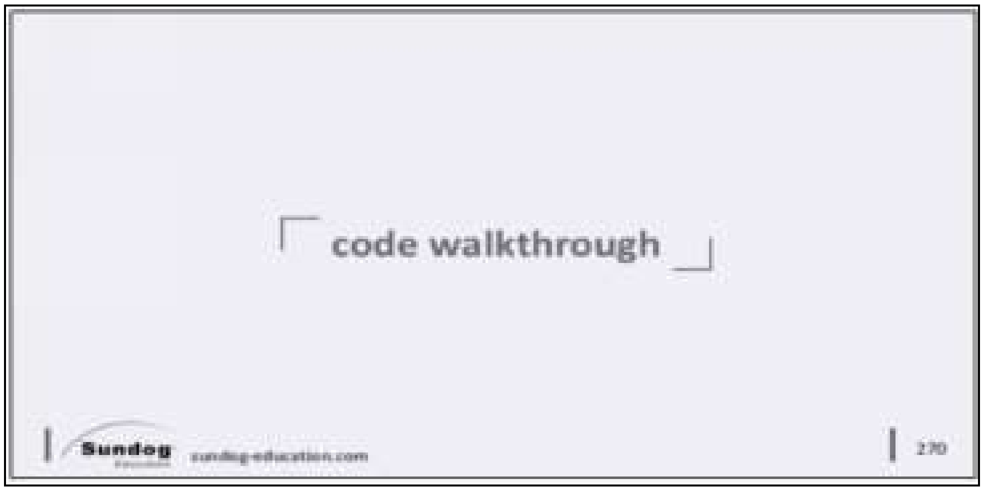
We'll extract the results for our good friend, user number 85, using the filter function. And then we use collect() to retrieve those final results that we're interested in from the cluster and back to our driver script, where we can then print them out.

Let's run it and see what happens! If you installed Spark as instructed, it should work – but there's a lot that can go wrong, so don't sweat it too much if you just have to watch my own results.

OK, so the RMSE score is 1.12, which isn't that great – some hyperparameter tuning may be in order, or the data set may just be too small. I've tinkered with Spark's ALS a lot in the past and I suspect it's the latter. The top N recommendations we got back look pretty weird as well, so maybe this isn't the best algorithm to use for this data. But the important point is that it could be distributed – even though we're running this on a single PC, you could run this same exact script on a cluster and Spark would distribute it all for you automatically. Even on a single PC, it can use the different cores of your CPU to do parallel processing if it's beneficial.

So even if the results aren't too exciting – the capability we've uncovered here is!

Let's see if we can take it farther.



code walkthrough

Sundog sundog-education.com 270

We promised we'd scale things up in this section – so let's see what Apache Spark can do, even on a single PC. We're going to straight from the 100,000 ratings data set we've been using so far, to the 20 million ratings dataset from MovieLens! Let's see if Spark can handle that.

MovieLens's license terms don't allow me to redistribute their data, so you'll have to head over to grouplens.org, select the datasets page, and download the ml-20m.zip file from the 20 million ratings data set there. Once it's downloaded, uncompress it, and place the ml-20m folder inside your course materials folder.

Now, let's go back to Spyder, and open up the SparkALS-20m.py file.

There are only a couple of things we changed here. First, I wrote a new loadMovieNames function instead of relying on our MovieLens module, because the movie ID's in the 20 million dataset are different from the ones in the 100K data set. This is really just copied and pasted from the code in our MovieLens module that did the same thing for the 100K dataset, just using a different file path .

The only other change is on line 31 – you can see here that I've added a configuration setting to our SparkSession specifying that it should use 4 CPU cores. My PC has 4 cores, so I can tell Spark to use them all – and it's sort of like having a cluster of 4 machines at that point. Technically, 4 executor processes will be spun up on my PC when I run this, all running in parallel on different parts of the data. Without that parallelism, 20 million ratings would be too much for a single process to handle. If your PC has a different number of CPU cores, you'll want to change this value accordingly.

When you're running on a real cluster though, things are usually pre-configured such that the default settings Spark uses are already optimized to the hardware available on your cluster. You don't normally need to think about this, or hard-code assumptions about your hardware into the script – this is just a special case because we're running on a single PC and not on a real cluster. In the real world, you'll usually be working on a cluster of some sort, and what I said earlier about using the same exact driver code no matter how big your cluster is would be true.

Let’s kick it off. It will of course take some time to train a matrix factorization model with 20 million ratings, and then generate top-10 recommendations for every single user in that dataset – but it actually finishes more quickly than some scripts we’ve run with just 100,000 ratings! We’ll pause, and come back when it’s done.

OK, you might recall that with the 100,000 ratings dataset, our RMSE was 1.12, which wasn’t great – but now it’s down to 0.81, which is actually pretty awesome. This tells us that the ALS algorithm in Spark’s ML library doesn’t work well with small datasets, or it’s tuned for larger ones with its default hyperparameters, or both. But regardless of the reason, these results are MUCH better at least in terms of accuracy .

We will also sample the recommendations of a given user, but you have to keep in mind user 85 in this dataset isn’t the same user 85 that was in our 100K dataset. Also, the results you see on your PC will be different because there is a random component to how the model is trained. But, after running it a couple of times, it would seem that this user’s tastes are pretty obscure, but somewhat consistent. Old movies from the 40’s and 50’s keep coming up for him or her, as well as more modern, foreign films or films about foreign lands. Even if we don’t have a deep subjective understanding of this user and the movies being recommended, we can still see that there seem to be some coherent patterns in what’s being recommended that aren’t just random – so that’s a good sign that this system is doing something useful.

But, the really impressive thing is that we just built a really good recommender system in just a few minutes from 20 million ratings! That is truly big data, and thanks to the efficiency of Spark, we didn’t even need more than one computer to do it!

Apache Spark is pretty cool, and it’s worth learning about in more depth. But it is a bit limited in the tools it offers for producing recommendations – let’s look at some other systems that can produce recommendations at massive scale.

Amazon DSSTNE

amazon dsstne

Sundog sundog-education.com 271

Next we're going to talk about a system from Amazon called DSSTNE. That stands for Deep Scalable Sparse Tensor Neural Engine.

Back when I worked in Amazon's Personalization team, our algorithms and systems were some of the most closely guarded secrets in the company – but in 2016, they open-sourced DSSTNE and published an article on exactly how to use it to produce recommendations at Amazon scale.

DSSTNE makes it possible to set up deep neural networks using sparse data, like the ratings data we have when training recommender systems – without writing a single line of code. All you have to do is convert your training data into the format DSSTNE expects, write a short configuration file that defines the topology of the neural network you want and how it's optimized, and it does the rest. DSSTNE also runs on a GPU, and can process massive amounts of data very quickly. If your PC has multiple GPU's, it can even distribute the load across them and compute recommendations in parallel .

And, if you need to scale up even further, you can integrate DSSTNE with Apache Spark to run it across a larger cluster. Let's start by digging into an example of using DSSTNE on a single machine, and then we'll talk about how to scale it up to a cluster.

a sample config file

```
[illegible]
```

Sundog sundog-education.com 272

Just to make it hit home, here is the entirety of the config file that trains a 3-layer neural network on the MovieLens data set, and it produces some really amazing results. Amazon has really made it as simple as it can get.

Let’s walk through this a bit. We start off with some information about the neural network at a high level – we are creating a feed-forward autoencoder. Specifically it’s something called a “sparse autoencoder.” In this context, sparsity refers to placing constraints on the hidden layer to force them to find interesting patterns, even if there are a large number of hidden nodes. The values under SparsenessPenalty define those constraints. The details of what these values represent would require you to find the lecture notes from Andrew Ng’s CS294A course at Stanford about sparse autoencoders, but those are available online – however, usually the defaults are a decent starting place .

Next, we specify that we don’t want to bother shuffling the indices, but we are applying “denoising” which will randomly flip 1 inputs to 0 for the probability specified – in this case, 20%. This forces the hidden layer to discover more robust features instead of just converging on 1:1 relationships, by making it reconstruct the input from a corrupted version of it. It’s kind of weird, but we talked about this a bit in our deep learning tutorial – you get better results by forcing your neural network to work harder, be it from hiding information from it, or in this case, straight up messing with its inputs on each iteration during training.

Next we define the parameters for our loss function. These are actually a little bit different from the defaults, which are 0.9, 0.1, 1.0, and 1.0 – so I think some hyperparameter tuning has already been done for this example. It's not far off, so the default values are probably a good starting place still.

Finally we get to the meat of it – the Layers, where we define the layers of our neural network. Ours defines three layers.

The first is the input layer – note that we specify it is "Sparse." This is what makes DSSTNE special – it's built for the case where most of the input data is missing. The input case here is a lot like the RBM example we looked at earlier – for each user we train our network with, we have an input node for every possible item, which is set to 1 if the user rated that item – but the vast majority of items were not rated by any given user. This means our input data is very sparse, and by specifying this layer as sparse, DSSTNE knows this and can deal with it efficiently. We set N to "auto", meaning it will set up as many input nodes as we have inputs in the data file, and that data file is given by the file named gl_input.

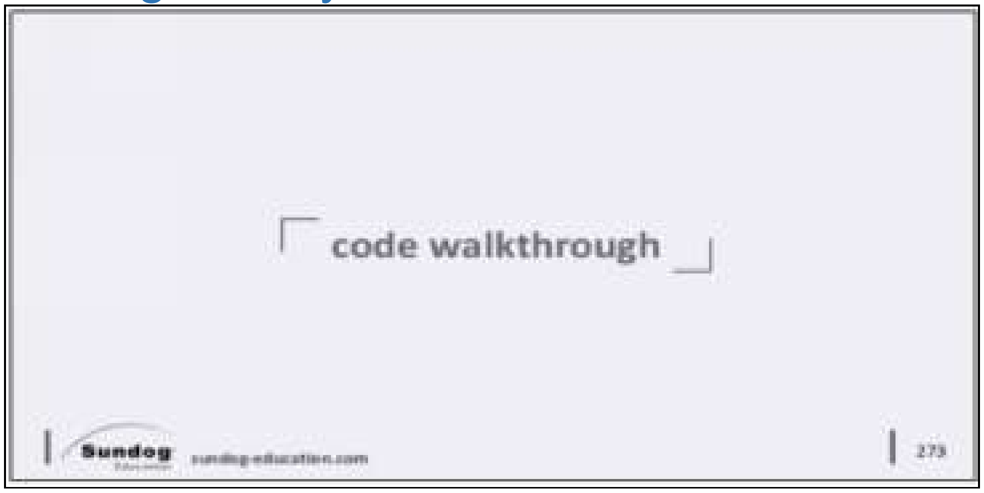
Our second layer is the hidden layer, specified as such. It is fully connected, with 128 nodes, and a sigmoid activation function. It too is sparse .

And finally we have our output layer. We give it the "answers" for training purposes in the gl_output file, and set N to auto so it can adapt to the structure of the features in gl_output we are trying to predict. It too has a sigmoid activation function, which is what we generally use for binary output.

To take a step back, you may have noticed that this architecture seems to assume binary inputs and outputs, but the MovieLens rating data exists on a 1-5 scale. It turns out that in this example, they discard the actual rating values themselves – they take any rating as an implicit, binary indication of interest in the movie. And it works surprisingly well. However, you could do the same trick we did with RBM's and model each rating value as an individual binary classification to get finer grained results and predictions.

Let me show you this in action.

Coding Activity



code walkthrough

Sundog sundog-education.com 273

DSSTNE is open source, and you'll find it on GitHub. It's made for Ubuntu systems that have one or more GPU's attached (a GPU is the core of a 3D video card, which can also be used to run neural networks on.) I don't have a system like that handy, so I'm going to use Amazon's EC2 service to rent time on one instead.

The kind of machine you need for this isn't cheap, so I'm not going to ask you to follow along here as it would cost you real money. What we're doing does not fall under Amazon's free usage tier for people who are just starting and experimenting. If you want to see all the steps involved in getting the machine itself set up with all of the necessary software, it's detailed on the setup page for the project in GitHub, here (https://github.com/amzn/amazondsstne/blob/master/docs/getting_started/setup.md) But I'm going to skip past the steps for actually procuring a machine and installing the necessary software, and just jump straight to playing around with this example.

Instructions for running the example are also here, in GitHub. (https://github.com/amzn/amazon-dsstne/blob/master/docs/getting_started/examples.md) They point out that generally, using DSSTNE involves three steps: converting your data, training, and prediction.

First, let’s go into the directory that contains the MovieLens example:

```
cd /opt/amazon/dsstne/samples/movielens
```

We’ll start by downloading the MovieLens 20-million rating data set from GroupLens.

```
 wget http://files.grouplens.org/datasets/movielens/ml-20m.zi p
```

All we actually need is the ratings data, so we will extract that from the zip archive, and save it as ml-20m_ratings.csv.

```
unzip -p ml-20m.zip ml-20m/ratings.csv > ml-20m_ratings.csv
```

DSSTNE however can’t deal with csv files, nor does it want data that’s organized as one rating per line. Ultimately DSSTNE requires file formats in NetCDF format; they provide a tool that can convert files into it, but that conversion tool also has its own requirements. So, we need to convert our ratings data into a format that can then be converted into NetCDF.

This first step requires arranging our data so that every row represents a single user, followed by a list of all the rating inputs for that user. What we want is each line to contain a user ID, followed by a tab, followed by a colon-separated list of each item ID that user rated. If you think about it, that’s basically a sparse representation of the input layer for each individual user, one user per line. So this input format gets us very close to what we need to train our neural network with.

Amazon provides an awk script to convert our csv file into this intermediate format. Let’s take a quick look with cat convert_ratings.awk.

```
cat convert_ratings.awk
```

It takes advantage of the fact that all of the ratings for a given user are grouped together in the raw MovieLens data; it just collects all the ratings for each user, then spits out a line for each user when it's done. Let's go ahead and run that conversion script:

```
awk -f convert_ratings.awk ml-20m_ratings.csv > ml-20m_rating s
```

And if we take a peek at the output:

```
head ml-20m_ratings
```

We can see that it does contain training data, one user per line, with the items that user rated in a sparse format.

Now we need to convert this to the NetCDF format DSSTNE requires:

```
generateNetCDF -d gl_input -i ml-20m_ratings -o gl_input.nc -f features_input -s samples_input –c
```

This actually generates three files; the NetCDF file itself, an index file with the indices of each neuron, and an index file with the indices of all samples.

We also need to generate the output layer's data file, using the same sample indices we generated for the input file:

```
generateNetCDF -d gl_output -i ml-20m_ratings -o gl_output.nc -f features_output -s samples_input –c
```

Now we're ready to rock. Let's take a quick peek at config.json again, which defines the topology of our neural network and how it is trained:

```
cat config.json
```

It's the same file we looked at in the slides and already covered, but again, there at the end is the meat of the 3 layers in our neural network. It's pretty amazing how simple this looks, but remember there is a lot of complexity that's being hidden from you here, involving sparse autoencoders, a tuned loss function, and a denoising feature – as well as all the complexity that comes with the sparse data, and distributing the training of this data across a neural network that's not running on our CPU, but on the GPU – and possible even multiple GPU's! It's pretty amazing what's going on here.

Let's kick it off, with parameters that indicate a batch size of 256, and 10 epochs:

```
train -c config.json -i gl_input.nc -o gl_output.nc -n gl.nc -b 256 -e 10
```

That was actually amazingly quick, for 20 million ratings! If you had more than one GPU, it would be even faster – in that case, there is an mpirun command you would use instead of the train command. The output of this is the gl.nc file, which contains the trained model itself.

So now that we have our trained model, let's use it to create top-10 recommendations for every user.

```
predict -b 256 -d gl -i features_input -o features_output -k 10 -n gl.nc
-f ml20m_ratings -s recs -r ml-20m_ratings
```

The parameters there are pretty self explanatory. –k 10 means we want top 10 recommendations, and we are passing in our model that lives in the gl.nc file. One thing worth mentioning however is that the predict command will automatically doing filtering as well, which is why we're passing in the ml-10m_ratings file again – this allows it to filter out items a user has already rated.

Let's kick it off; it doesn't take very long.

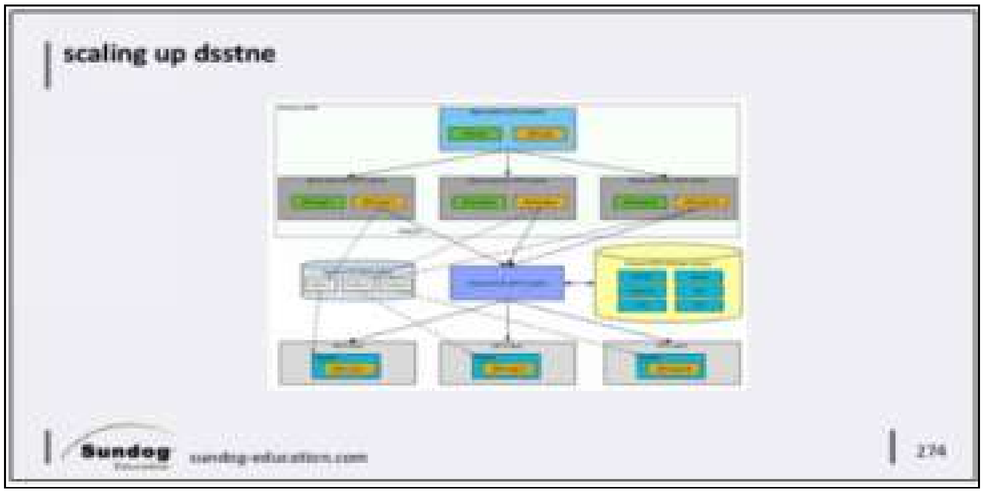
So, the results are in the "recs" file. Let's take a peek at it:

```
head rec s
```

Each line is a user ID, followed by a list of their recommended items and scores. Unfortunately it's not human readable, but let's spot check a few. I’ve opened up the movies.csv file in Excel so we can look up a few of these recommended item ID’s and see what movies they correspond to.

Let’s check out the first few movies for user 0. They turn out to be The Empire Strikes Back, Star Wars, and Alien. That’s… pretty exciting! It seems to have nailed user 0 as a science fiction fan, and recommended some of the greatest science fiction movies of all time.

Let’s reflect again on what just happened here. We trained a feedforward 3-layer neural network with sparse data from 20 million ratings, in just a few minutes by using the power of a GPU to accelerate that neural network. There was no need to deal with all the complexity of RBM’s in order to apply neural networks to recommender systems; DSSTNE has made it so we can apply deep learning directly to this problem, and think of it as any other classification problem a deep neural network might excel at. It’s purpose built for applying deep learning to recommender systems, and that in itself is pretty darn exciting.



So, we’ve shown that even a single Ubuntu system with a single GPU can crunch massive data sets when generating recommendations using deep learning in a short amount of time. But what if you need to scale up even more, to the point where a single GPU can’t handle it?

Well, Amazon has told us how to do that, as well. On the AWS Blog, you'll find an article called "Generating Recommendations at Amazon Scale with Apache Spark and Amazon DSSTNE." (https://aws.amazon.com/blogs/big-data/generatingrecommendations-at-amazon-scale-with-apache-spark-and-amazon-dsstne/) Just like it sounds, they are combining the power of DSSTNE with the power of Apache Spark, using the same Amazon Web Services you or I can use.

The basic idea is to use Spark to distribute the job of data analytics and processing, such as the conversion of data into NetCDF format that we just did, on the CPU from Spark Executor nodes. Once the data is partitioned by Spark, it's handed off to another cluster dedicated to GPU computing, where DSSTNE can train its neural networks on that data.

To take a step back, Spark can't manage clusters of GPU's – so they use two different clusters: a Spark cluster to wrangle their data and partition it up, and then another GPU cluster that's actually managed by Amazon's container management service,

ECS. They use Amazon S3 to transfer data between these two clusters. So Spark is our CPU cluster, and ECS is our GPU cluster. Spark orchestrates the whole thing, using S3 to transfer data back and forth.

On a specific GPU slave, Amazon DSSTNE is just one of several technologies that might run to actually process the data. What's even nicer is that Amazon Web Services has auto-scaling features that will automatically allocate and de-allocate these nodes as they are needed.

learning more

https://amzn.to/2l69kAw

Sundog sundog-education.com 275

Amazon published all the details you could possibly want to know about how they scaled up DSSTNE to run on a cluster; here's a short link to that article. You obviously can't run it yourself without first setting up a very large, complex, and expensive cluster to work with, but they even show you Python notebooks where they use this cluster to generate movie recommendations. They can run the whole thing from a notebook in their browsers.

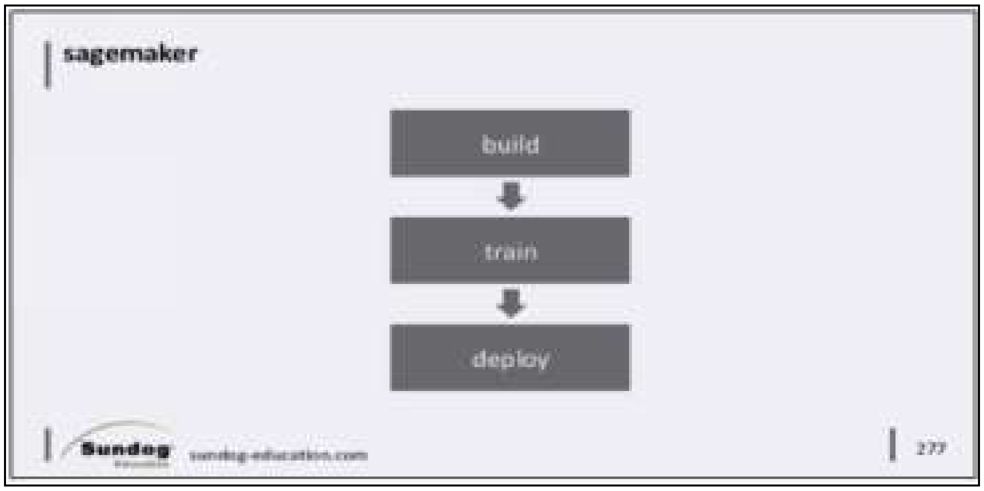
I think this is a very exciting moment in this course; we've seen how to apply real deep learning networks to massive recommender system problems, and scale it up pretty much infinitely .

AWS SageMaker

amazon sagemaker

Sundog sundog-education.com 276

Also worth a mention is Amazon's SageMaker services. SageMaker is a component of Amazon Web Services, and it allows you to create notebooks hosted on AWS that can train large-scale models in the cloud, and then vend predictions from that model from the cloud as well. It's an easy way to get some serious computing horsepower behind your recommender system in an on-demand manner, and it comes with some useful algorithms for recommender systems too.



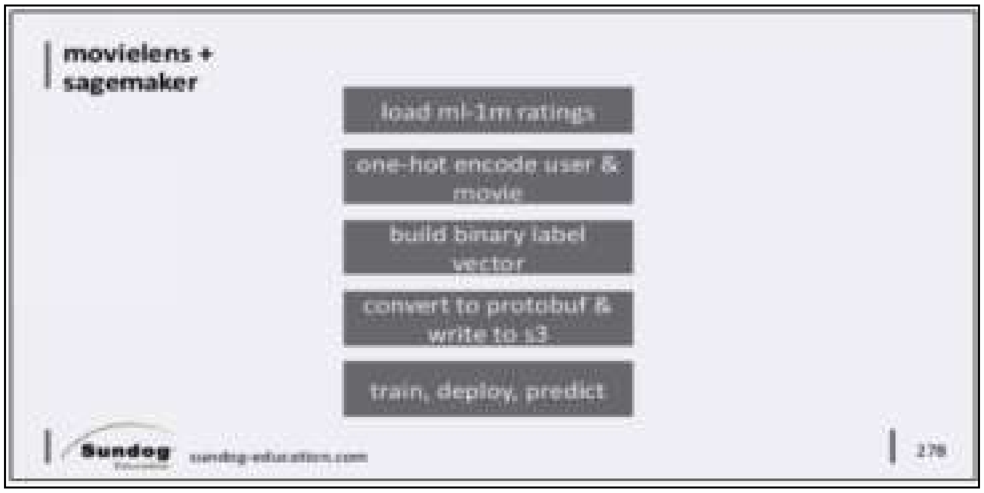
Using SageMaker involves 3 steps: building your model, training your model, and deploying your model. Let's start with building.

It's pretty easy to start using SageMaker – you just push a button in the AWS console to start a new notebook instance, and a hosted Jupyter notebook environment will be spun up for you, with access to all of SageMaker's built-in algorithms available to you. You can spin up environments that include most any deep learning framework that you want to use as well, such as Tensorflow or Apache's MXNet. You use that notebook to build your model, in our case a recommender system algorithm.

You can kick off the training of your model from that same notebook, or from the AWS console by setting up a training job. Your training and test data need to be in a specific format called "protobuf," but Amazon provides utilities that make it easy to convert your data into this, much like it was easy to prepare your data for use in DSSTNE. One neat feature of SageMaker is that you can also set up “hyperparameter tuning” jobs, which will automatically run a series of cross-validation tests to converge on the best parameters for your model. And, this training can be distributed – AWS will spin up as many machines as you want, and train your model in parallel across them. This isn’t free, of course – that computer time costs real money.

Finally, you place your trained and tuned model into production with the deploy step. This also spins up one or more servers in AWS that will make predictions on demand using your model. Again, this too is not free – you’ll be charged for computer time for as long as your model is deployed, so you need to remember to shut down your model’s deployment when you’re done with it, or you’ll have a nasty surprise on your credit card bill at the end of the month.

I won’t say using SageMaker is simple, but we can walk through an example that will at least help make more sense of it.



movielens + sagemaker

- load ml-1m ratings
- one-hot encode user & movie
- build binary label vector
- convert to protobuf & write to s3
- train, deploy, predict

Sundog sundog-education.com 278

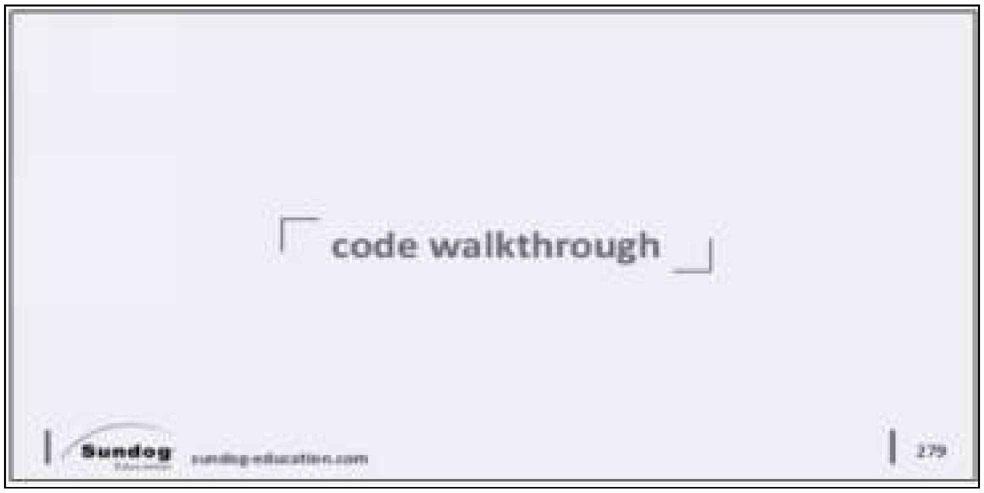
If you remember back to our matrix factorization section, we talked about an algorithm called factorization machines, which is a more general purpose means of factorizing sparse matrices such as the user/movie ratings matrix we have with our MovieLens data. SageMaker includes an implementation of factorization machines, so this is our chance to finally try them out – and since we’re on the cloud with AWS, we can do this at large scale. Let’s walk through how we might use SageMaker to train a factorization machine model on the MovieLens 1-million-rating dataset.

Most of the work is actually in getting the MovieLens data into a format that SageMaker can work with. Factorization machines want to work with high-dimension data sets, so we need to “one hot encode” each rating. Let’s say we have 500 users and 1000 movies in our data set – the idea is to encode each rating as 1,500 binary values, where the first 500 values would represent each user, and the next 1000 values would represent each movie. We’d set only two values in those 1500 values to 1 – one for the user, and one for the movie that was rated. The rest are set to zero. All those zeros are a huge waste of space, so we use a sparse tensor to store each one-hot-encoded rating in a much more compact format.

We also need a vector of labels to train with, indicating whether this specific rating for a given user/movie pair indicated that the user liked the movie or not. Somewhat arbitrarily, we’ll say any rating of 4 or higher gets a label of “1”, and 3 or below gets a “0”. The algorithm just works best with binary values like this.

Once we have our sparse ratings vectors and label vectors ready for both our training and test data sets, we convert them to the protobuf format SageMaker expects, and write them to S3 where SageMaker can access them.

Once our training and test data are in the expected format and in the expected place, the rest is relatively simple. We tell SageMaker to spin up some hardware to train our factorization machine model, deploy the trained model into production, and then we can query it for specific predictions for given user/movie combinations. Let’s have a look at the code.


code walkthrough

Sundog sundog-education.com 279

For this activity, just watch – doing this yourself would involve spending real money, and it's a little too easy to forget to shut things down when you're done and end up with a huge AWS bill you didn't expect.

I've already created a SageMaker notebook, so I'll start that in the SageMaker console in AWS and we can take a look at it. It will take a minute or so for the notebook environment to spin up before we can launch it.

Once the notebook environment is up, we can create new notebooks or open up ones we created earlier. Let's open up my MovieLens-1m notebook here.

This is loosely based on a similar example in the AWS blog, but I've simplified it a bit and modified it to work with the one-million-rating data set. Let's restart the kernel so we have a clean slate to run with.

The first thing we need to do is download the MovieLens data set and decompress it, so we'll use a little trick in notebooks where you can use the exclamation mark to execute shell commands. We'll use wget to download the data, and unzip to decompress it into an ml-1m folder .

Let's change into that folder, and take a peek at the ratings.dat file to remind ourselves what format it is in. Looks like it's delimited by double-colons, and there is no header row of column names. So that will tell us how to import this data.

I’m going to use the Pandas library to make life a little bit easier in manipulating this data. We use the read_csv command to load it up into a Pandas DataFrame object, with given column names and using that double-colon separator. We then peek at the beginning of the resulting dataframe to make sure it looks OK, and it does.

Next we need to create a train/test split on the data so we can train and evaluate our model. Scikit_learn’s train_test_split function makes that easy; we’ll reserve 20% for the test set, and save the resulting dataframes with the names “train” and “test.” Let’s peek at the train dataframe just to make sure it still looks OK. It does.

Now, in order to set up the sparse one-hot matrix we discussed earlier, we need to know how many features exist in each row. Remember we’ll have one feature per user ID, and one per movie ID. So, we need to get the maximum user ID and item ID, and add those together to figure out how many features we need in our training data. Fortunately Pandas makes those numbers easy to come by – and we end up with a total of 6040 users and 3952 movies in this particular data set.

Now we need to build up the feature and label tensors for each rating. Since the feature data is mostly 0’s, we’re going to use a sparse matrix called lil_matrix here. Let’s walk through this code. It’s a function that takes in either our training or test dataframe, and converts it into a feature matrix called X, and a label matrix called Y. We iterate through every row of the dataframe. First we set a 1 in the column of the feature matrix that corresponds to this rating’s user ID, and we also set a 1 in the column to correspond to this rating’s movie ID. If the rating is 4 or better, we set the corresponding label to 1 indicating that the user liked this movie, otherwise we leave it as zero.

Finally we convert the labels to float32 format, since that’s what SageMaker requires.

Let’s go ahead and call this function for both our train and test dataframes. This may take a few minutes… and it’s worth noting that this particular part of our notebook doesn’t really scale very well. It requires us to keep all of our training data in memory on the one host that’s running this notebook. SageMaker makes it easy to distribute the training and predictions of your machine learning models, but distributing the pre-processing of your data is still kind of up to you. Maybe it would make more sense to use Spark or something to do pre-processing at a large scale – as long as your data ends up in S3 in the format SageMaker expects, it doesn’t really matter how it gets there.

So, now we need to convert this data into protobuf format, and write it into S3. That’s what this writeDatasetToProtobuf function does; you can see it’s pretty simple. I created an S3 bucket of my own called sundog-sagemaker for this; obviously you would substitute in your own S3 bucket that you have access to if you were doing this yourself. We’ll call this for the feature and label data for both the training and test dataset, and once this runs, SageMaker will have all it needs to work with.

So now we will create a factorization machine model in SageMaker. First we need to tell it where to find the container for the factorization machine model in our AWS region, and we’ll set up a key in Amazon S3 under which to store the resulting model .

We create our factorization machine model by creating an Estimator with the factorization machine container, and because we’re cheap we’ll train it on a single c4.xlarge machine.

We also need to set the hyperparameters – we’ll jut pick some arbitrarily here, but again SageMaker does have the capability to tune these for you with hyperparameter tuning jobs. They take a lot of time and resources though, so we’re going to skip that step here.

Finally we call “fit” on our model, passing in the S3 keys for our training data and test data. At this point, SageMaker will spin up the hardware we requested and start training our model – and again, this will take a few minutes.

Once our model has been trained and stored, we can deploy it so we can start querying our model to make predictions. Let’s spin up a single instance to host this model, since we’re not going to be calling it at massive transaction rates or anything. This will take a few minutes for the requested hardware to be allocated and set up.

Now that our model is deployed, we can use it to make predictions on whether a given user will like a give movie. The factorization machine model produces output in json format, so we need a little bit of code to unpack that json data and print out the results inside of it here. We need to explicitly tell the predictor we deployed that it is in json format, and how to serialize and deserialize that json data. But with that out of the way, we can just call the “predict” function on our factorization machine predictor for some set of data from our test feature data, and then compare it to the labels in our test data.

If you walk through each prediction and actual value, the results are pretty hit and miss. I would hope for something better given a million ratings to work with, so this doesn’t make me super excited about factorization machines. Still, they’ve been shown to work as good or better than SVD in many situations – and you’ve seen how you can use Amazon SageMaker to train a factorization-machine-based recommender system and predict user ratings at large scale. SageMaker is just another tool in your toolchest for doing machine learning at scale, and I’m sure its capabilities will only expand over time.

Challenges of Recommender Systems

The Cold-Start Problem

the cold-start problem

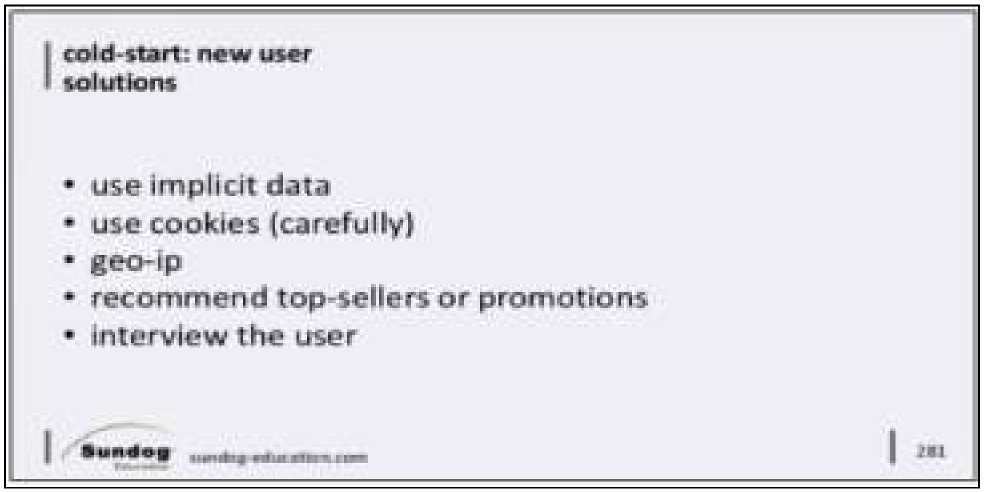
Sundog Education sundog-education.com

280

One of the better-known issues with recommender systems is what is known as the “cold start” problem.

If a brand-new user arrives at your site, what do you recommend to them when you know nothing about them yet?

And new users aren’t the only problem – what about new items in your catalog? How do they get recommended when there is no data on them yet to pair them with other items?


cold-start: new user solutions

- use implicit data
- use cookies (carefully)
- geo-ip
- recommend top-sellers or promotions
- interview the user

Sundog Education sundog-education.com

281

When faced with a new user, your options are limited. The one saving grace is that a new user isn't new for long – assuming they stay at your site at all, you'll soon have some information to work with that indicates their interests.

As soon as this new user looks at a new item, you'll have at least some implicit information about this user's interests – even if it's just that "this user looked at this product." If you're lucky, they landed on your site on an actual product page, and you can just recommend other items similar to the item they are looking at. As it turns out, the thing a person is looking at right now is actually the strongest indication of what they are interested in right now. That's why "people who bought this also bought" is such a successful feature on Amazon – and it works even for people who aren't logged in, or are brand-new to Amazon. All they need to know is what you're looking at right now, and recommending products similar to that is a really safe bet.

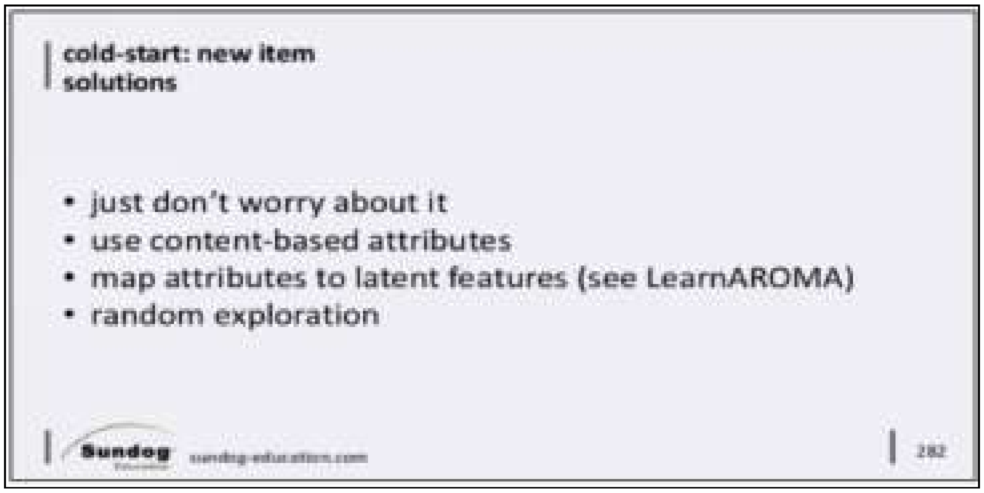
But, if a new user just lands on your home page, you really don't have much to work with. A fair question to ask, however, is whether this really is a new user or not .

Perhaps they are actually an existing customer who just hasn't logged in yet. Sometimes you can use browser cookies to help identify users even when they are logged out, and tie the user session to a user account for use in making recommendations. You have to be really careful about this, however. The world is enacting stricter and stricter laws on how websites use cookies, and if you get a users' identity wrong, you can end up in a real mess if you recommend items to someone based on somebody else's interests. That would be a real violation of user privacy.

There is one thing you know about anybody who lands on your website, and that's the IP address they are connecting from. Many IP addresses can be translated into geographical locations, or at least a good guess of where the user is connecting from. So you could imagine building up a database of "people from this region liked these items" using a similarity metric between places and items, instead of people and items. You could then recommend items that are uniquely popular in the area you think this user is located. The connection between a person's location and their interests is tenuous at best, but it may be better than nothing.

A safer bet is probably to just recommend top-selling items when you have nothing else to work with. Top-sellers are by definition the things a person is most likely to buy if you know nothing else about their unique interests. And it's a pretty easy thing to work with – you're probably already computing top-sellers on your platform anyhow; all you have to do is show them to users you don't recognize. You might also take this as an opportunity to promote items that your business wants to push, or that they are receiving advertising dollars for. Since you can't provide personalized recommendations, think about what the next best use of that space on your website would be .

Finally, you can straight up ask the new user what their interests are. If you remember back to the very early days of Netflix, when you signed up they would present a bunch of popular movies to you and ask you to rate them if you had seen them, in order to get some information to work with on your personal interests. If your site really revolves around personalized recommendations, this might be a reasonable thing to do – but the reality is most people these days aren't going to be willing to devote that kind of time to you. A lot of people will just abandon your site when you start to demand that sort of work from them, so you don't see this approach being used much these days.



cold-start: new item
solutions

- just don't worry about it
- use content-based attributes
- map attributes to latent features (see LearnAROMA)
- random exploration

Sundog sundog-education.com 282

The other aspect of the cold start problem is how to deal with new items. When a new item is introduced into your catalog, how will it get recommended to users when there is no data to associate it with other items or other users yet?

Well, one option is to just not worry about it. Presumably, there are other ways for users to discover new items on your website than recommendations. They will show up in search results, they might appear in promotions, and the maker of the item is probably advertising it in hopes that people will search for it and buy it on your site. As soon as people start indicating their interest in this new item through these other items, then your recommender system will start to pick up on it. In practice I've found this to be a perfectly reasonable approach. But, some researchers do consider this a real problem and have proposed other solutions.

If you remember way back to the beginning of this course, we talked about content-based recommenders. Even if you have no activity data on this new item, you have data on its attributes – its title, its description, its category, and probably other stuff as well. You can use that to recommend it alongside items that have similar attributes. Making recommendations based on content attributes alone is never a good idea, but if you augment your behavior-based recommendations with content-based ones, that solves the item cold start problem right there. Let's say you're generating top-10 recommendations for a user based on ratings data, but you can only come up with 8 good recommendations. Maybe those last 2 slots can be filled by content-based recommendations, which might surface new items.

Taking that idea even further is merging content attributes with latent features learned from ratings behavior. If you learn latent features through matrix factorization or deep learning, perhaps you can associate those latent features with content attributes, and use the behavior patterns you learned from existing items to inform relationships with new items. It's a pretty complicated idea, but if you want to learn more, look up a paper called LearnAROMA that details this approach. It's definitely bleeding-edge territory, but it's a neat idea.

Finally, there's the idea of random exploration. We talked about using extra slots in your top-N recommendations to surface content-based recommendations, but those content-based recommendations could be from new or old items. If you really want to tackle this problem directly, you could dedicate those extra slots to just randomly showing new items to users, in an attempt to gather more data on them and surface them to users more quickly. I'm not really sold on this approach, because a brand new item isn't likely to be something a user recognizes – so I don't think they're likely to click on them. But if you truly have nothing better to show to a user in that real estate on your website, it might be a reasonable thing to do. It's pretty easy, so let's go ahead and give it a try.

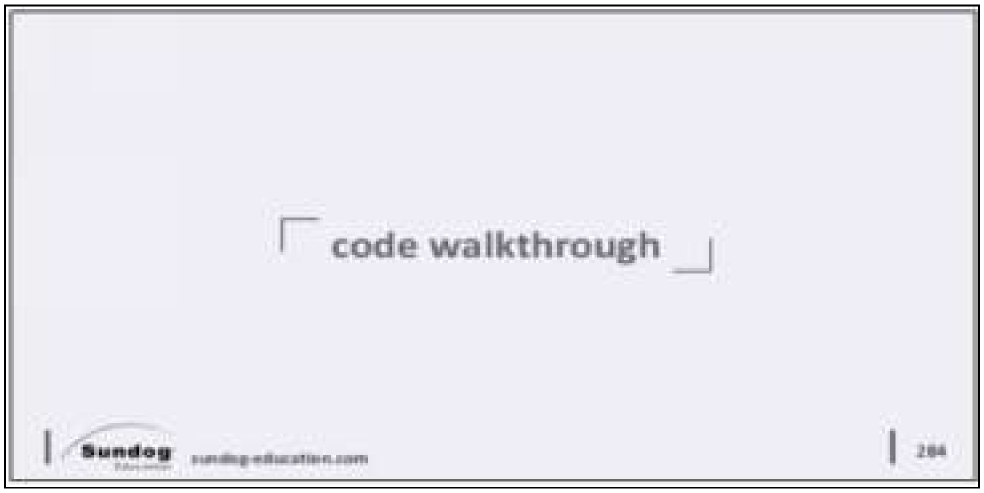
Exercise: Random Exploration

exercise: random exploration

Sundog sundog-education.com 283

As an exercise, try your hand at implementing random exploration. Go back to the EvaluateUserCF script way back in the collaborative filtering section of the course; it generated top-10 recommendations for every user with user-based KNN, and measure the hit rate.

Your challenge is to modify this code such that the last slot in your top-10 recommendations is always used to surface a random movie that was released in the most current year in the MovieLens data set. To do this, you'll have to modify our MovieLens module to identify and extract movies in the most current year it has, and modify the EvaluateUserCF script to replace the tenth slot in each users' recommendations with one of these movies, chosen at random. See what impact it actually has on the hit rate. If it doesn't impact hit rate much, then it might be a reasonable thing to do if you have a real problem with new movies not being surfaced to users quickly enough. Up next, I'll review my solution to this exercise.


code walkthrough

Sundog sundog-education.com 284

Hopefully you managed to get that running, but if you’re new to Python it may have been a little bit challenging. If you’d like to look at my solution, open up Spyder and close out everything you no longer want to look at with control-shift-W. Now open up everything in the “Challenges” folder of the course materials.

Select the “EvaluateUserCF-Exploration” script. There are only a few changes here – first, we’re importing MovieLens2 instead of MovieLens, which is my modified MovieLens module that we’ll look at shortly. It’s been modified to identify movies from the most recent year. If we skip down to line 41, we can see where I’m retrieving that list of new movies, and defining which slot in my top-10 recommendations will be used to surface those new movies at random.

Skipping down to line 78, this is where we are constructing the top-10 recommendations for each user. If we’re in the final slot we defined, we choose one of those new movies at random – otherwise, we select the movie with the next-highest recommendation score for this user. There’s more than one way to code this up, by the way. An arguably better approach would be to just generate the top 9 recommendations normally, and then tack on that random new movie at the end outside of this loop. That way we’d get that new movie even if we didn’t have 10 recommendations to show for the user.

Let’s peek at my MovieLens2.py file. What’s new here is the getNewMovies function, which isn’t too bad. We just use the max() function to find the most recent year in our data. Then, we iterate through every movie, appending movies with a release date that matches that year to the newMovies list that we then return.

Let’s kick it off and see if it works. Go back to EvaluateUserCF-Exploration and hit play, and it will come back pretty quickly.

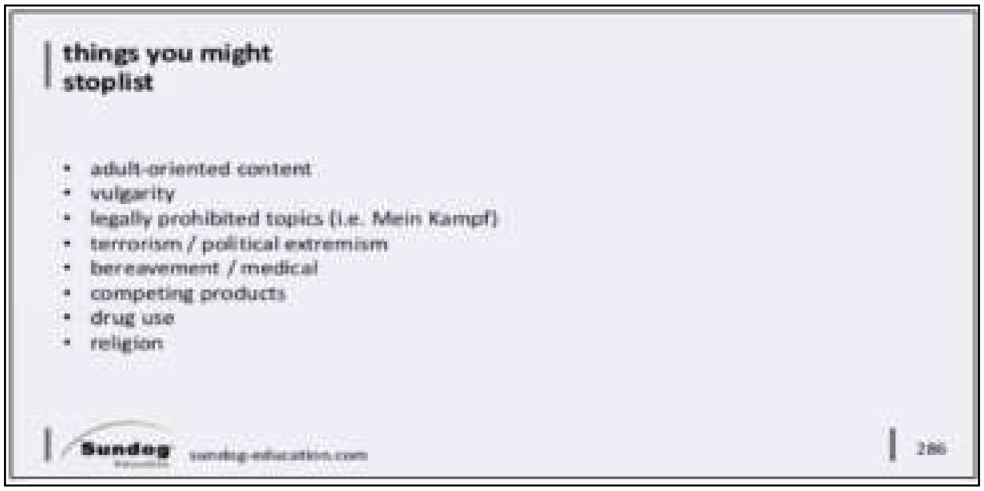
We can see the most recent movie in this data set is from 2016, and recommending random movies from 2016 really didn’t hurt our hit rate much – we’re still at over 5%, which is pretty good. With movies, new releases are usually of general interest, so it’s arguably not a bad thing to recommend – and if you have an explicit goal of surfacing new items to users as soon as possible, this is a reasonable way to do it .

Stoplists

Another issue you’ll deal with in the real world is that it’s easy to offend people, without ever intending to do so. I don’t know what sort of creature this is, but he looks offended – and you better hope your recommender system isn’t what offended him.

Let’s go back to the year 2006, when Wal-Mart’s web site was starting to experiment with making recommendations. As part of a manual rule someone added into it to promote DVD’s about the black civil rights leader Martin Luther King, their recommendations started to pair movies about Martin Luther King with a lot of different things. Including, unfortunately, movies such as “Planet of the Apes” and “Charlie and the Chocolate Factory.” You can imagine that people being told they might enjoy a movie about Martin Luther King because they liked Planet of the Apes didn’t go over well. And you’d be right – the story got picked up by national news, Wal-Mart had to issue an apology and donate money to civil rights causes, and they scrapped their entire recommendation system. Presumably, people lost their jobs due to this .

There are some topics that are just too touchy for you to deal with, and that’s where stoplists come in. There are certain terms and keywords where if they are present in the titles, descriptions, or categories of the items in your catalog, your recommender system shouldn’t even know that they exist. If titles associated with race were omitted from the training of Wal-Mart’s recommender system entirely, then they would have no potential to cause offensive pairings with other items. That’s all a stoplist is – checking to see if an item might cause unwanted controversy before you let it into your recommender system.



things you might stoplist

- adult-oriented content
- vulgarity
- legally prohibited topics (i.e. Mein Kampf)
- terrorism / political extremism
- bereavement / medical
- competing products
- drug use
- religion

Sundog sundog-education.com 286

There are a lot of topics that might cause a crisis if they are paired with the wrong items, and it’s best to err on the side of caution. I don’t care what your own moral or political beliefs are – if you’re working for a business, you have a responsibility to uphold the reputation of that business. If you don't agree with your customers' beliefs, frankly that doesn't matter. You don't want your recommender system to end up on the front page of the New York Times, and to be dragged in front of the CEO of your company to explain why you recommended Adolf Hitler's memoirs to someone looking for books about their favorite politician .

Some things are no-brainers. Content that's about sex is best avoided – you don't want to end up recommending sex toys alongside children's books, for example. Items with vulgarity in their titles or descriptions will set a lot of people off. Some topics are even illegal in certain countries, such as Naziism in Germany – and really, recommending Nazi content probably isn't a good idea anywhere. But many countries also ban explicit lists of political terms that you are required to honor if you're doing business there. If you want to do business in China for example, you're going to need to stoplist items that contain topics the Chinese government doesn't like. Terrorism and political extremism will certainly tick some customers off, as will religious topics – especially more fundamentalist religions. It's not about making value judgements, it's just the universal adage that you shouldn't talk about politics or religion with strangers.

Some things are more subtle, however. Items associated with bereavement or medical care comes to mind. Imagine you're caring for a loved one who is suffering from terminal cancer, and you bought some books about cancer. Once that loved one passes away, you certainly don't want to be reminded of it be seeing recommendations for more books about cancer every time you come back to a website. It's best to just leave books like that out of your training entirely.

Or maybe you're recommending baby diapers to people who bought the book "What to Expect when you're Expecting." Seems reasonable at first, but would you want to see recommendations for diapers if your pregnancy actually ended in a tragic miscarriage? Perhaps books about prenatal care are best stoplisted as well.

Sometimes, stoplists might be used for other purposes entirely. Perhaps you don't want to promote certain items due to business concerns, because they compete with items you sell yourself. As much as you might want your recommender system to be a completely neutral, level playing field – that's just not the reality of business. There may be items you don't want to recommend strictly for business reasons and nothing more.

You can't anticipate every PR disaster though, so it's important that your stoplist can be updated and applied quickly should the need arise. You might even have an emergency stoplist that prevents titles from being displayed on the front end, while the stoplist on your backend training system has time to work its way through.

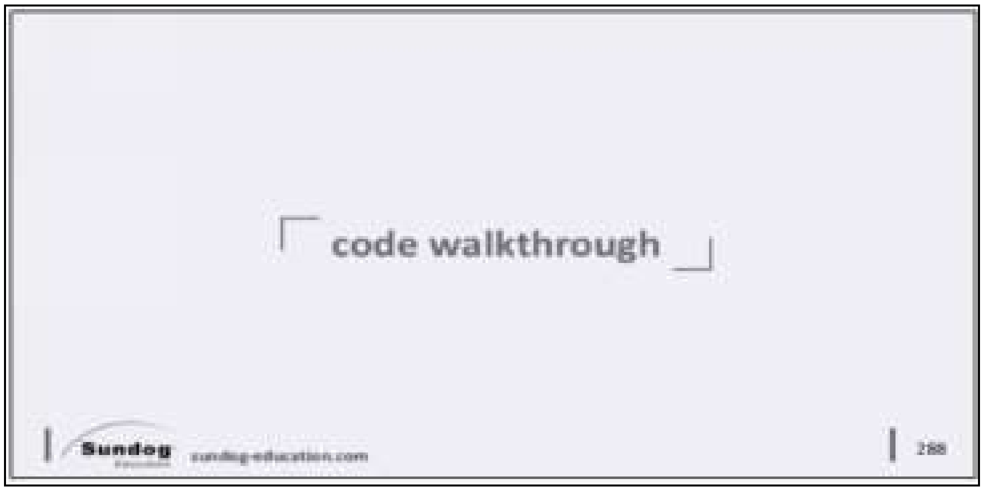
Not every offensive recommendation pairing is an accident, either. If you are using implicit ratings information, for example user clicks or search terms, it's possible for people to game your recommender system and generate fake associations between items by simulating enough traffic on your website to create those associations. For example, right now if I do a Google image search for the word "idiot," pictures of the current US President will appear due to a coordinated campaign from his opponents. Part of the role of your stoplist is to have a quick way to respond to attacks like that, because even though half of Google's customers might find that amusing, they're risking a boycott from the other half who do not.

exercise: implement a stoplist

Sundog sundog-education.com 287

Given how important a good stoplist is in the real world, it’s worth your time to actually go and implement one. So, that’s your next exercise. Try modifying our RBM example from earlier in the course to filter out any movies that contain words in a stoplist from the training stage entirely. Your goal is to make sure that our recommender system doesn’t even know these titles exist.

To do so, you’ll have to modify the RBMAlgorithm class to apply this stoplist as it’s building up the training matrix. That’s probably the only hint you need. Give it a go, and when we come back, I’ll show you how I did it.


code walkthrough

Sundog sundog-education.com 288

OK, so if you open up my RBMAlgorithm.py file inside the Challenges folder of the course materials, you can see how I went about implementing a stoplist.

In the init function, I’ve loaded up the MovieLens module because we need it for looking up movie titles from movie ID’s. We’ve also defined a very small stoplist here, containing just the terms “sex”, “drugs”, and “rock n roll.” This obviously isn’t a real stoplist – you should put a lot more thought into yours, and make sure it includes the roots of words and not specific tenses or variations of them. For example, in a real stoplist, I’d stoplist “drug” and not “drugs” to catch as many drug-related words as possible.

Next we have the buildStoplist function on line 26. All we’re doing here is building up a dictionary that lets us quickly look up if a given item ID is banned or not. We iterate through every item in our training set, convert it to a raw item ID, and look up its title. If it has a title, we first convert that title to lowercase. Since the terms in our stoplist are also lowercase, this makes our comparison case-insensitive, which is very important. If we find that term in our stoplist, we set a true on that item ID's entry in the stoplistLookup dictionary.

Now, if we skip down to line 53, we can see where the stoplist is applied. As we're building up our training matrix, we check every internal item ID to see if it has been flagged in our stoplist lookup. Only if it's not do we proceed to add it into our training data. In this way, our recommender system gets zero data about potentially offensive items, which prevents it from creating any pairings with them.

We can kick this off if you'd like by opening up the RBMBakeoff.py file and running it. Our code that builds up the stoplist lookup prints out any titles it encounters that get flagged, and you can see that even our little joke of a stoplist picked up some titles – and some of them look, well, shall we say "interesting." I'm sure you can think of quite a few items you wouldn't to see recommended alongside the movie "Sex Tape" for example.

So it seems like our stoplist is doing its job. Again, in the real world, you'd want to put a lot more thought into the actual terms you are stoplisting, and you'd want to make sure you can update and apply your stoplist very quickly should the need arise .

Filter Bubbles

The next real-world challenge is a tough one. It’s called “filter bubbles,” and it refers to societal problems that arise when all you show people are things that appeal to their existing interests.

Let’s say you’re building a recommender system for a book store. If someone buys a book about a topic associated with right-wing politics, your recommender system will probably pick up on that, and start recommending more right-wing books to that person. If they respond to those recommendations, that person gets more and more immersed in right-wing ideology. The same happens for someone who bought a book about left-wing politics. The end result is that the recommender systems we developed to try and show people interesting things creates a more polarized society. This isn’t something we ever anticipated in the early days of building recommender systems, but as the same techniques have been applied to social networks and advertising, it really has caused people to be exposed almost exclusively to things that reinforce their existing beliefs when they are online .

This is called a “filter bubble,” because the content you’re presenting your users is filtered in such a way that it keeps them within a bubble of their pre-existing interests. The term arose in 2014 from Eli Pariser, and it proved to be a very prophetic observation. He gave a TED talk on it that you can look up if you want to hear more about it.

The tough thing is that this isn’t necessarily bad from a business standpoint, but it’s arguably very bad from an ethical standpoint.

You’ve seen how to measure diversity in your recommendations, so you can actually quantify just how “bubbly” your recommendations are. A little extra diversity in your recommendations can give people exposure to items that might help them break out of their bubble. There’s no easy solution, but making sure your business as whole doesn’t rely too heavily on personalized content is a good start. Exposing items or content with broader, universal, objective appeal that resonate with the population as a whole instead of certain subgroups can help keep society unified. It’s a little hard to imagine that your recommender system might contribute to the downfall of civilization, but honestly, that’s a question I’ve had to grapple with myself as an early practitioner in this field. And it’s not fun to think about .

Trust

transparency and trust

We're recommending Science Fiction & Fantasy Books because

Sundog sundog-education.com

290

We’ve talked a little bit in the past about user trust in your recommender system. It’s a good idea to make sure there are several recommendations that they recognize in their top-N results, to ensure they have some confidence that your system is producing relevant recommendations for them. That way, they’ll be more likely to explore the items you’re recommending that they haven’t heard of.

There’s a flipside to the trust problem as well, and that’s what happens when a user that sees a recommendation that just makes them go “huh? Why are you recommending THAT to me?” No matter how good your algorithm is, spurious results can creep in, and sometimes this can undermine a users’ trust in your system as a whole.

A good antidote to this is transparency – allow the user to see exactly why you recommended this item to them, and ideally, let them fix the root cause themselves. It’s a much better outcome if a user understands why you recommended something they found confusing and actually fixes the issue, than if they start posting on social media about how horrible your company’s recommendations are.

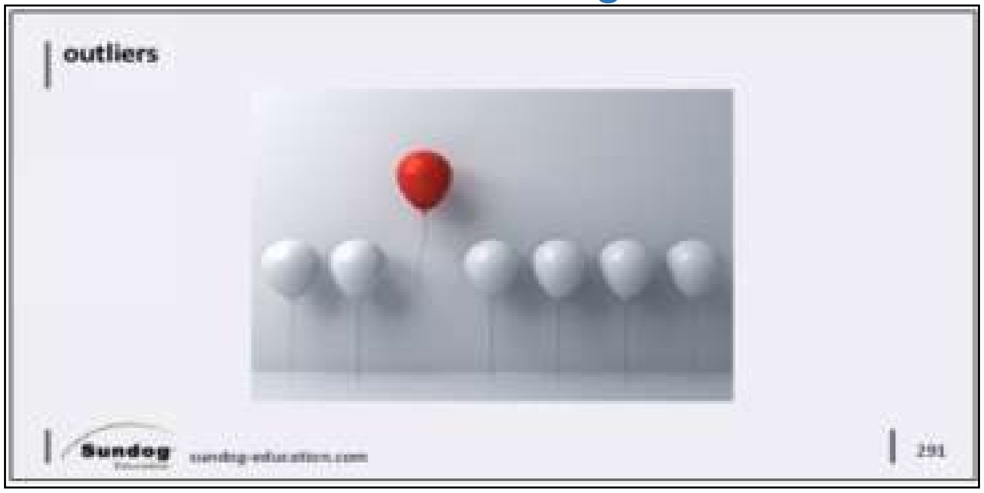
This example comes from Amazon’s recommendations, where they recommend products from certain categories to me based on what they think my interests are. I can click into the reasons why a given category was recommended, and see the specific items from my purchase history that led to the conclusion that I’m interested in this topic. If, for example, this category recommendation was based on items I had purchased as a gift for somebody else, I could exclude those items from my recommendations right here and now.

So if I weren’t actually a science fiction fan, instead of posting on Facebook about how wrong Amazon is about me, maybe I’d see that sci-fi related gift I bought for my cousin here and understand why that recommendation happened – and I can fix the issue as well.

Of course, it’s not always this easy. When you’re dealing with more advanced algorithms such as deep learning or matrix factorization, you’re basing recommendations on latent factors that can’t easily be communicated to the user – or even to you as a researcher, for that matter. That’s part of why you don’t see this sort of thing very often any more – also, it’s a lot of front-end work to develop the user interfaces for tools like this, and it can be hard to justify the expense of developing it compared to all the other stuff your company needs to build.

Still, all things being equal – transparency is a good thing for trust in your recommender system .

Outliers and Data Cleaning



outliers

Another issue you'll deal with in the real world is outliers. Really, this is a concern for any machine learning system, not just recommender systems. Your results are only as good as the data you feed into it, and oftentimes, a lot of the work is in cleaning that input data, and filtering out outliers that might skew your results in unnatural ways.

For example, imagine building a collaborative filtering system based on user rating behavior. What if some of the users in your data aren't actually real people at all, but are bots that are rating things in an unnatural manner? A bot might also generate an excessively large number of ratings, and end up having a very large influence on your final recommendations. It's not always malicious behavior that you're up against, either – it might just be a web crawler that's polluting your data, or maybe even your own internal tools!

Even among real people, you might want to filter some of them out. People who review items for a living will generate a large number of reviews, but their opinion shouldn't necessarily be that much more powerful than everyone else's. Another example is institutional buyers, who purchase large quantities of different items on behalf of companies – their behavior doesn't really reflect anyone's real interests, but they can have a huge influence on your recommendations if they aren’t identified and filtered out. In most algorithms, a person with a large number of reviews will have an outsized influence on the recommendations your system makes.

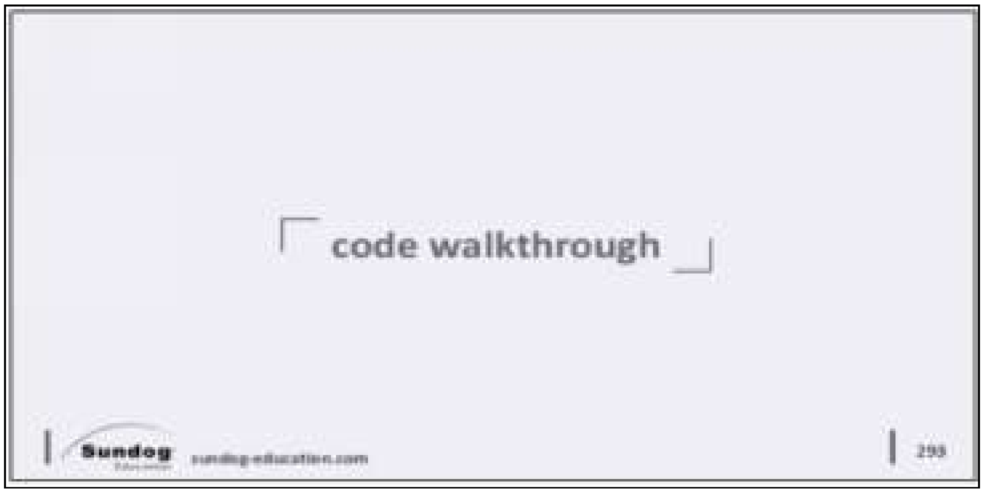
So, as with any machine learning system, you want to identify the outliers in your training data, understand them, and filter them out when appropriate.

exercise: filtering outliers

Sundog sundog-education.com 292

Let’s pause for another hands-on exercise. Revisit our EvaluateUserCF script again, and this time modify the recommender such that users that have a rating count more than 3 standard deviations from the mean are excluded from consideration. This will eliminate so-called “super-users” who have an outsized impact on your results. Let’s measure the effect of filtering them out.

To do this, you’ll probably want to focus your attention on the MovieLens module again – the best place to filter out these outliers is in the function that actually loads the MovieLens data set itself. Doing this easily will require some familiarity with the Pandas module, so if you’re new to Pandas, you might want to just skip to my solution and learn from it. But if you are starting to feel comfortable with Pandas, give it a shot yourself – and compare your results to mine, up next.


code walkthrough

Sundog sundog-education.com 293

So to see how I went about filtering outliers in the MovieLens data set, open up the MovieLens3.py file in the Challenges folder of the course materials.

The changes are in the loadMovieLensSmall function. You can see I've re-implemented it such that it uses Pandas to load up the raw ratings data, and then I use Pandas to filter out those outliers. The resulting dataset for our recommender framework is then built up from the resulting Pandas dataframe, instead of directly from the csv ratings file.

We start by loading up the ratings data file into Pandas, into a dataframe called ratings. We print out the start of it, and the shape, so we can see what the data we're starting with looks like, and how much of it there is.

Next we need to identify our outlier users, which first means counting up how many ratings each user has. We use the groupby command on line 34, together with the aggregate command, to build up a new dataframe called ratingsByUser that maps user ID's to their rating counts – and we print it out so we can see what we're working with as we develop and debug all of this .

Now we're going to add a column to that ratingsByUser dataframe that indicates whether or not this user is considered an outlier. Pandas is pretty neat in that we can just define a new column by assigning it to a set of operations on the existing dataframe. Here, we’re taking the absolute value of the difference between each user’s rating count and the mean rating count across all users, and comparing that to the standard deviation of the rating counts multiplied by the value we choose, which is 3 by default. So, this line adds a new “outlier” column that is true if the rating count for this user is more than three standard deviations from the mean, or false otherwise. We don’t need the actual rating column any longer, so we drop that, and print out what we have at this stage.

Next, we merge in this dataframe that indicates who are outliers and who aren’t, with our original ratings dataframe, based on the user ID’s. This gives us a handy outlier column on the raw individual ratings data. Again we print out what we have so far on line 45.

Next, we’ll get rid of all the outlier data on line 47 by creating a new dataframe called “filtered” that is just the merged dataframe, but only including rows where the outlier column is false. We can then drop the outlier and timestamp columns from the resulting dataframe, as we don’t need that data any more.

At this point we have what we want – a dataframe of individual user ID’s, movie ID’s, and ratings that excludes ratings from users that we consider an outlier. On lines 49-51 we print out the resulting dataframe and its shape, so we can get a sense of how many ratings got dropped as a result of all this.

From there, we’re fortunate that supriselib has a convenience function for creating a dataset from a Pandas dataframe, which is what we’re doing on line 54 .

So, let’s open up EvaluateUserCF-Outliers.py, and kick it off to see what happens.

You can see all of our debugging output that illustrates what happens at each stage within our loadMovieLensLatestSmall function, as we build up the outlier data and exclude ratings from those outliers. An important take-away is that we started off with 100,004 ratings, and ended up with 80,398. So even with the fairly conservative definition of an outlier as being beyond 3 standard deviations, we ended up losing quite a few ratings.

The resulting hit rate was 4.4%, and we saw hit rates from this same algorithm over 5% earlier. So it would seem that in this case, eliminating outliers did more harm than good. That's just specific to this data set, however – it turns out that the MovieLens data set we're working with has already been filtered and cleaned, and we know all of this data represents real, individual people. So by filtering out outliers, we're just filtering out valid information. But in a real world situation, things will probably be quite different, and you'll find that removing outlier users or even outlier items can have a measurable improvement on your results .

Malicious User Behavior

Another real-world problem is people trying to game your system. If items your recommender system promotes to your users leads to those items being purchased more, the makers of those items have a financial incentive to find ways to game your system into recommending their items more often. Or, people with certain ideological agendas might purposely try to make your system recommend items that promote their ideology, or to not recommend items that run counter to it. Some hacker might even be bored, and try to create humorous pairings in your recommender system just for their own amusement. "Google bombs" are an example of this.

Fighting people like this is generally a never-ending arms race, but there is one technique that works remarkably well: make sure that recommendations are only generated from people who actually spent money on the item. Recommendations based on implicit ratings from purchasing data are almost impervious to these sorts of attacks, because it would be prohibitively expensive for someone to buy enough of an item to artificially inflate its presence in your recommendations. And, when people " vote with their wallets", it's a very strong and reliable indication of interest that leads to better recommendations overall.

Sometimes, however, you don't have enough purchase data to work with. There are still precautions you can take, however. For example, if your recommender system is based on star reviews, you can make sure that you only allow reviews from people you know actually purchased or consumed the content in question. If you allow people to rate items they haven't actually seen or used, you're opening yourself up to all sorts of attacks. And using click data should always be a last resort – it's very easy to fake click data using bots, and even if it's not a bot, click data has its own set of problems.

The Trouble with Click Data

implicit data, explicit problems.

Sundog sundog-education.com 295

Using implicit clickstream data, such as images people click on, is fraught with problems. You should always be extremely skeptical about building a recommender system that relies only on things people click on, such as ads. Not only are these sorts of systems highly susceptible to gaming, they're susceptible to quirks of human behavior that aren't useful for recommendations. I've learned this the hard way a couple of times.

If you ever build a system that recommends products based on product images that people click on when they see them in an online ad, I promise you that what you build will end up as a pornography detection system. The reality is, people instinctively click on images that contain images that appear sexual in nature. Your recommender system will end up converging on products that feature pictures that include a lot of flesh, and there won't be anything you can do about it. Even if you explicitly filter out items that are sexual in nature, you'll end up discovering products that just vaguely look like sex toys or various pieces of sexual anatomy.

I'm not making this up – I'm probably not at liberty to talk about the details, but I've seen this happen more than once. Never, ever build a recommender system based on image clicks! And implicit data in general tends to be very low quality, unless it's backed by a purchase or actual consumption. Clickstream data is a very unreliable signal of interest. What people click on and what they buy can be very different things .

International Considerations

international markets
and laws
Sundog
sundog-education.com
296

We're not done yet! There are just a lot of little "gotchas" when it comes to using recommender systems in the real world, and I don't want you to have the learn them the hard way.

Another consideration is dealing with International markets. If you recommender system spans customers in different countries, there may be specific challenges to that you need to consider.

For example, do you pool international customer data together when training a recommender system, or keep everything separated by country? In most cases, you'll want to keep things separate, since you don't want to recommend items in a foreign language to people who don't speak that language, and there may be cultural differences that influence peoples' tastes in different countries as well.

There is also the problem of availability and content restrictions. With movies in particular, movies will often be released on different schedules in different countries, and may have different licensing agreements depending on which country you're in. You may need to filter certain movies out based on what country the user is in before presenting them as a recommendation. Some countries have legal restrictions on what sort of content can be consumed as well, which must be taken into consideration. You can't promote content about Nazi Germany within Germany for example, nor can you promote a long list of political topics within China.

Since your recommender system depends on collecting data on individual interests, there are also privacy laws to take into consideration, and these too vary by country. I am not a lawyer, and these laws are changing all the time, but you'll want to consult with your company's legal department and IT security departments to ensure that any personal information you are collecting in the course of building your recommender system is being collected in accordance with international laws.

The Effects of Time

dealing with time

Sundog sundog-education.com 297

A topic that is really under-represented in recommender system research is dealing with the effects of time .

One example is seasonality. Some items, like Christmas decorations, only make for good recommendations just before Christmas. Recommending bikinis in the dead of winter is also a bad idea. Picking up on annual patterns like this is hard to do, and most recommender systems won't do it automatically. As far as I know, this is still an open problem in the research arena. Those of you looking for a master's thesis topic – there you go.

But, something you can do more easily and more generally is taking the recency of a rating into account. Netflix in particular found that these sorts of temporal dynamics are important; your tastes change quickly, and a rating you made yesterday is a much stronger indication of your interest than a rating you made a year ago. By just weighting ratings by their age, using some sort exponential decay, you can improve the quality of your recommendations in many cases – or you can use rating recency as a training feature of its own, in addition to the rating itself.

As we mentioned in the context of Amazon, the product you're looking at right now is the most powerful indicator of your current interest. Time plays a big part there, as well.

Whenever you train a recommender system using historical ratings data, you are giving your system a bias toward the past. A Netflix that didn't take time into account would end up recommending a lot of old shows to you instead of the hot, newer ones you want to see. If the things you're recommending are time-sensitive in any way, it makes sense to use rating recency as a training feature of its own .

Optimizing for Profit

value-aware recommendations

Sundog sundog-education.com 298

I could keep on going with real-world lessons, but I'll end with this one.

In the real world, a recommender system you're building for a company will ultimately exist for the purpose of driving their profit. You may find yourself being asked to optimize your recommender system for profit, instead of pure relevance.

What does that mean? Well, normally you'd test different recommender systems based on what drives the most purchases, video views, or some other concrete measure of whether a customer liked your recommendation enough to act on it. If you're trying to build a recommender system that introduces people to new stuff they want, then that's the right thing to do. You'd want to just optimize on the quantity of items your recommender system drove the consumption of, not how much money those items made for the company .

But in the real world, some items are more profitable than others. Some items might even be offered at below cost as loss leaders to attract people to your site, and recommending them actually costs your company money.

This presents a bit of a moral quandary for developers of recommender systems. Do we keep our algorithms “pure” and optimize only for users’ interests, or do we work profit into the equation by rewarding algorithms that generate more profit? The euphemism for this is “value-aware recommendations,” where the contribution to the company’s bottom line plays a role in whether something is recommended.

If you’re ever faced with this dilemma, a reasonable compromise is to only use profitability as a tie-breaker. If you’re showing top-10 recommendations to someone, and the underlying scores for the top 2 items in that list are about the same, it wouldn’t do any harm to show the more profitable item first. Items higher in a list are more likely to be clicked on (we call this “position bias”,) and if you’re really not sure which item should come first, you may as well go with the one that will earn more money for your company.

Optimizing too much for profit can backfire, though. You don’t want to end up only recommending expensive items to your customers, because they’ll be less likely to purchase them. It might make more sense to look at profit margin, and not actual profit, so the cost of the item isn’t really a factor in what you recommend.

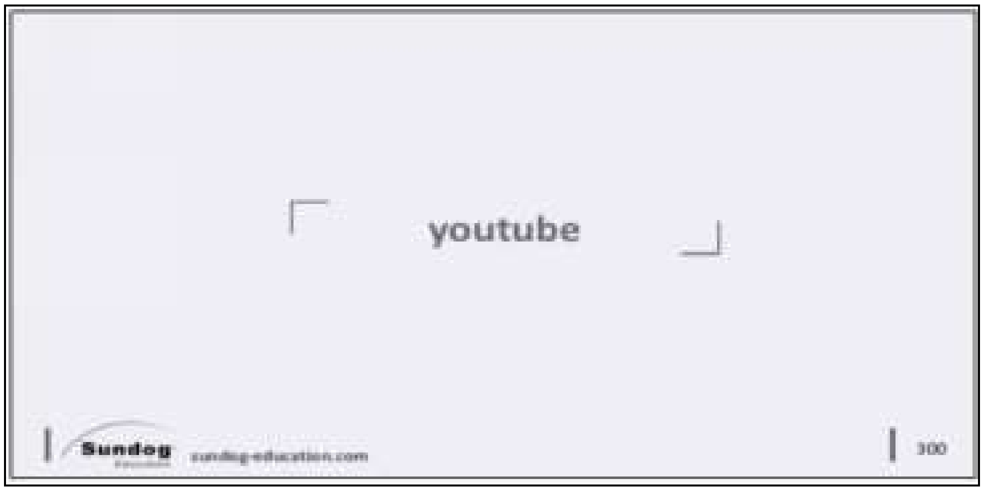
Case Studies

case studies

Sundog sundog-education.com 299

Keeping with the theme of exploring real-world lessons in recommender systems, let’s do a deeper dive into a couple of case studies from two of the best known recommender systems out there: YouTube and Netflix.

YouTube



youtube

Sundog sundog-education.com 300

Let’s start with diving into how recommendations work with YouTube. YouTube is now owned by Google, and given Google’s obsession with deep learning, it shouldn’t surprise you that YouTube has gone all-in with deep learning to power its recommendations.

the paper

Deep Neural Networks for YouTube Recommendations

ABSTRACT

Sundog Education sundog-education.com

301

A nice thing about Google is that they publish a lot of their research, and open-source a lot of their internal tools. As a business person I'd criticize them for enabling their competitors, but as an educator I think this is a wonderful thing.

They've made the paper "Deep Neural Networks for YouTube Recommendations" freely available, and it goes into quite a bit of detail on how YouTube recommendations work. Not only is this a treasure trove of insight for researchers in recommender systems, but it also contains valuable information for people trying to optimize their own YouTube channels. Let's go over some of the more interesting points in the paper.

youtube's challenges

- scale
- freshness
- noise

Sundog Education sundog-education.com

302

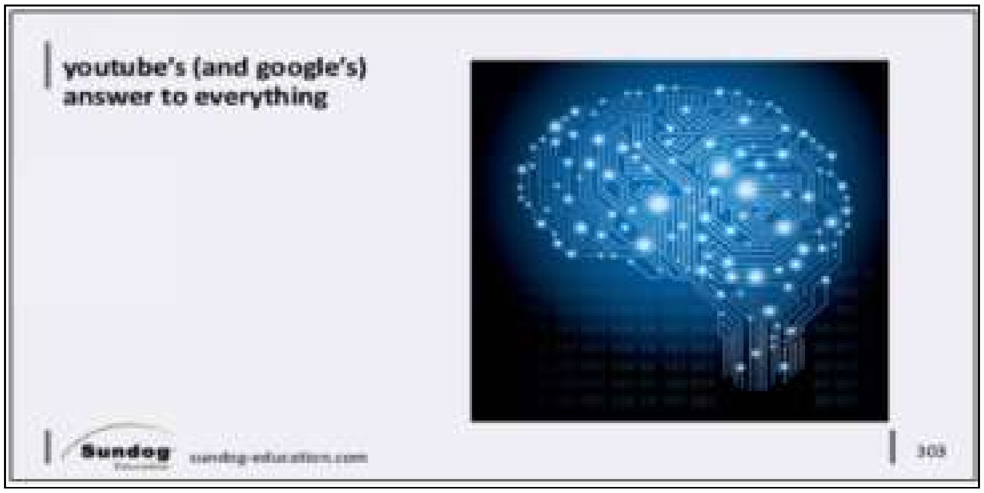
YouTube has some specific challenges, so keep these in mind before you decide to apply what they did to your own problems. If you don’t have the same problems, the same solution may not be the appropriate one for you.

The first issue is scale; YouTube has well over one billion users, who watch 5 billion videos every day. That’s just mind boggling when you think of how many users and items they have to compute recommendations for. Anything they do must be horizontally scaled and distributed on a massive cluster, and must be vended to the users extremely efficiently.

The next is freshness; over 300 hours of video are uploaded to YouTube every minute, and their recommendations must take into account both newly uploaded content, and up-to-the-second data on each users’ individual actions that might indicate their interests right now. They must strike a deliberate balance between recommended well-established videos, and new content – but they clearly favor new content.

The last big issue is noise; most of their interest data consists of implicit ratings, not explicit ratings. As we talked about in the previous section, implicit ratings data is fraught with peril – just basing recommendations on which videos people click on creates a system that’s subject to gaming and, in the words of the paper, “unobservable external factors.” Someone clicking on a viral video isn’t necessarily an indication of the ground truth of that user’s interests. Their data is also extremely sparse, and the content attributes associated with their videos is “poorly structured without a well-defined ontology.” That’s YouTube’s polite way of saying that the data they’re working with is a real mess.

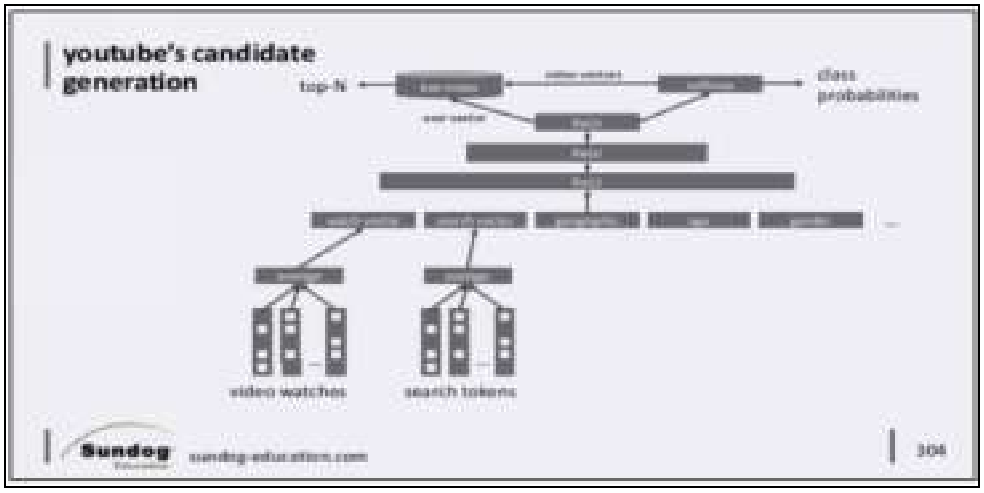
Based on their results, it seems they’ve done a good job of overcoming these challenges. When I visit the YouTube homepage, it comes up instantly and includes recommendations that do an excellent job of capturing my current interests.



youtube's (and google's) answer to everything

Sundog sundog-education.com 303

Google has gone all-in with deep learning, with a mandate to use it for nearly every machine learning problem they encounter. I can understand why they did this; it allows them to focus all of their engineering effort on deep learning frameworks, and use that as a general purpose tool for use by the rest of Google and its child companies, such as YouTube. So, in 2016 YouTube moved its recommendations to a deep learning framework, powered by "Google Brain", which today is called Tensorflow! Previously, their approach was based on matrix factorization, which we've also covered. As we've mentioned in the course, you can implement matrix factorization with a neural network, and at first that's exactly what they did. But over time, their approach evolved beyond matrix factorization.



youtube's candidate generation

Sundog sundog-education.com 304

YouTube was kind enough to publish their deep learning architecture, at least as it was in 2016. Let’s start at the bottom, where we have the user behavior data that is used to train the system. It’s interesting that although YouTube has explicit ratings in the form of thumbs up / thumbs down ratings, they don’t use them at all for generating recommendations because that data is too sparse. Not enough users rate videos explicitly for the data to be useful. Instead, they rely on implicit signals, such as which videos you actually watched, and what you searched for.

This implicit view and search data however is in itself sparse, and as we learned when covering deep learning recommenders, dealing with that sparsity is a huge issue when trying to apply deep learning to recommender systems. Their solution was to break up the sparse representation of video ID’s and search tokens for each user into a variable-length sequence of sparse data, mapped to a dense layer of a fixed length suitable for input to a neural network in an embedding layer. They simply average each chunk of the sparse input data. To say it another way, they split up each user’s sparse behavior data into chunks of a fixed length, and take the average of each chunk to reduce that data into fixed-length embedding layer that can be used as input to their neural network. They restrict this embedding to the most popular videos or search terms in order to keep it manageable for the scale they are dealing with; any obscure video you watch ends up getting mapped to the value zero.

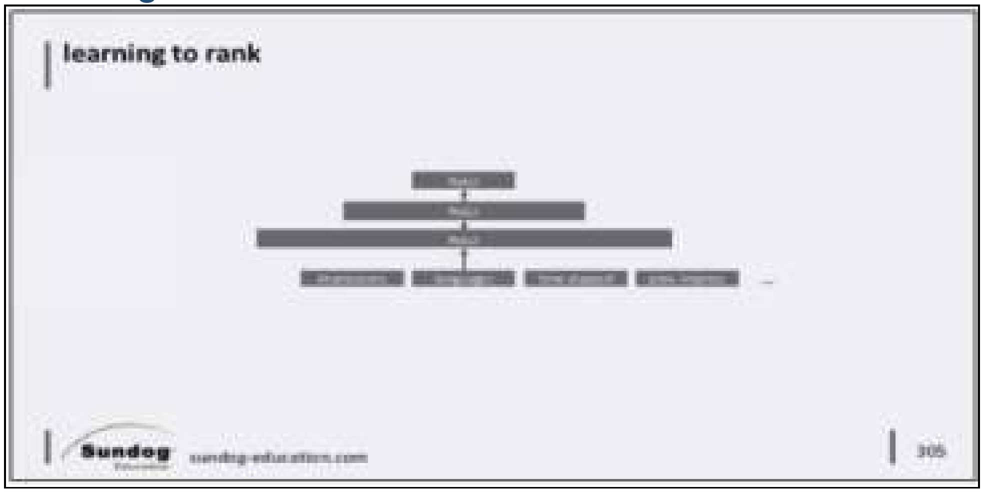
This may seem like an arbitrary choice, but they experimented with other ways of doing it, including summing and taking the max value for each component. And the way the embeddings work is itself learned through gradient descent backpropagation, so their system is actually learning the best way to reduce the dimensionality of their sparse data. In this way, they’ve avoided the problem of only training the system on videos or search tokens that were actually invoked by any individual user.

At the next layer, they combine together those averaged video watch vector and search token vectors with any other training features they want to incorporate into their model, such as the user’s geographic location, their age, their gender, and presumably other features not shown here.

All of those concatenated features are then fed as input into a deep neural network, trained with softmax. They settled on three layers; the first has a width of 1024 units, followed by 512, and then 256 at the top. This gave them a good balance of accuracy while staying within their budget for computing hardware.

The output of all this is fed into a database of nearest neighbors for each video, to generate more recommendation candidates based on what the deep neural network found. The problem is then to optimally rank all of these candidates .

Learning to Rank

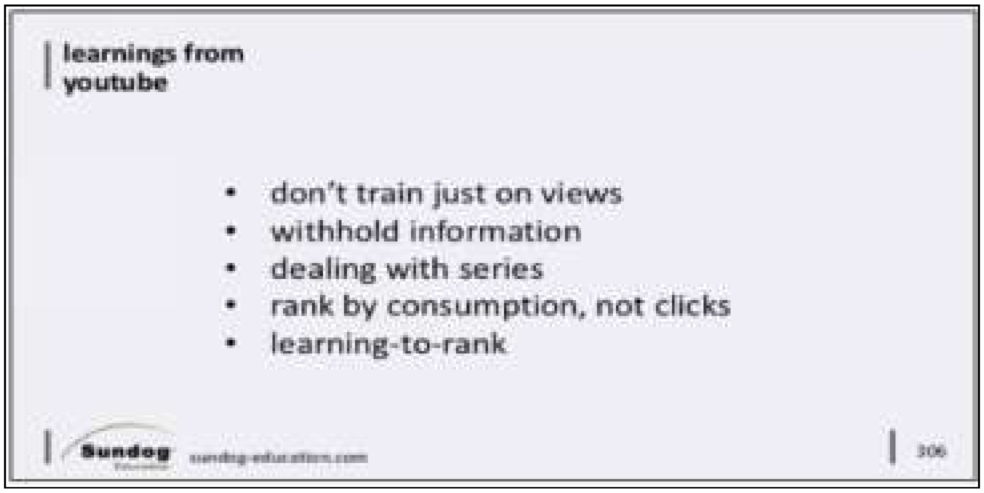


Given their mandate to use deep learning for everything, YouTube also uses deep learning to rank their recommendation candidates to produce a final top-N set of recommendations. This isn't a new idea; it's called "learning to rank" and there's a fair amount of research behind it as well.

They throw as many features into this neural network as they can. They take the entire impression history for the user – that is, every video that was showed to them, prior to viewing the video that generated this recommendation. These are embedded and averaged in a similar way to how the sparse view data was encoded to generate recommendation candidates. The language of the user and the language of the video are also embedded into another feature set. They also look at the time elapsed since the user last watched a video on this topic, and they look at the number of previous impressions this user had – which is actually used as a training feature in several different ways – they take the square root of this value, the square of it, and the value itself, all as separate input features. This allows the system to discover super and sublinear functions, which is a neat trick. Everything is normalized before being fed into the neural network of course, to make all of these features have equal initial weight.

The output of all this is actually used to predict the expected watch time of each video, which is what ultimately the ranking depends on. They don't want to optimize on predicting just clicks on a video, because this tends to surface "clickbait" videos that people aren't actually interested in. If a user actually watches a video all the way through however, that's a stronger indication of it satisfying that individual's interests. This is an important point – YouTube is optimizing for minutes watched, not for clicks – and this decision alone has motivated professional YouTubers to upload longer and longer videos in an effort to increase that watch time metric. They've sort of inadvertently introduced a new way to game the system.

They also refine the actual objective function they use to produce those final results through continual A/B testing, to try and find the function that best maximizes minutes watched. I think it's interesting that the need to combat the noisiness of video click data with minutes watched data has led to YouTube building a system that tries to drive watch time over all else. Fighting that noise has resulted in people spending more and more time on YouTube, and YouTube spending more and more resources to deliver all that video content. It's an example of a recommender system having consequences that are perhaps unintended.



learnings from youtube

- don't train just on views
- withhold information
- dealing with series
- rank by consumption, not clicks
- learning-to-rank

Sundog sundog-education.com 306

Here are some of the key take-aways from the paper as a whole.

First, don't rely on just view data when training a recommender system. The folks at YouTube came up with as many signals as they could from the user's past behavior and the user's attributes, and for some features they also fed these in as squares and square roots in case there were non-linear relationships to be found. View data alone is too easily gamed, and rewards "clickbait" videos that don't actually reflect a user's interests.

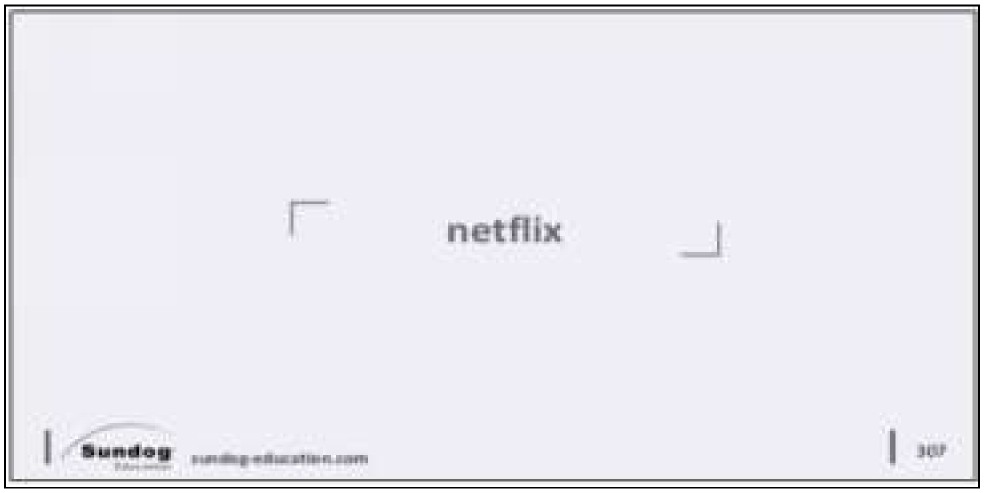
They also withhold some information from their recommender system in order to prevent overfitting. For example, predicting a user will watch a Taylor Swift music video because they just searched for Taylor Swift isn't really helpful – they would have found that video anyhow from the search results, and the relationship between that search and that video is a little too direct. As a result, they discard sequence information and obscure the search data fed into their system to try and prevent this sort of overfitting.

It's very common for YouTube users to watch videos that are part of a series; an episodic series of videos will generally be watched in order. So, they need to try and predict specifically the next video a user will watch, and not some randomly held-out future view. This means that leave-one-out cross validation as we've shown earlier in the course doesn't meet their needs; they aren't trying to predict a randomly held-out view, they must focus specifically on the next view.

But the main take-away is that they are ranking by actual consumption – minutes watched – and not just clicks on videos. Clickstream data is fraught with peril, and they've learned this the hard way.

Another important thing you've learned is that you can apply machine learning not only to generating recommendation candidates, but also to how these candidates are ranked. The systems we've covered in this course produce some sort of final recommendation score for each candidate that you can sort to produce top-N recommendations, but sometimes there is additional information you can incorporate into that ranking. And how that information is applied can itself be learned.

Netflix

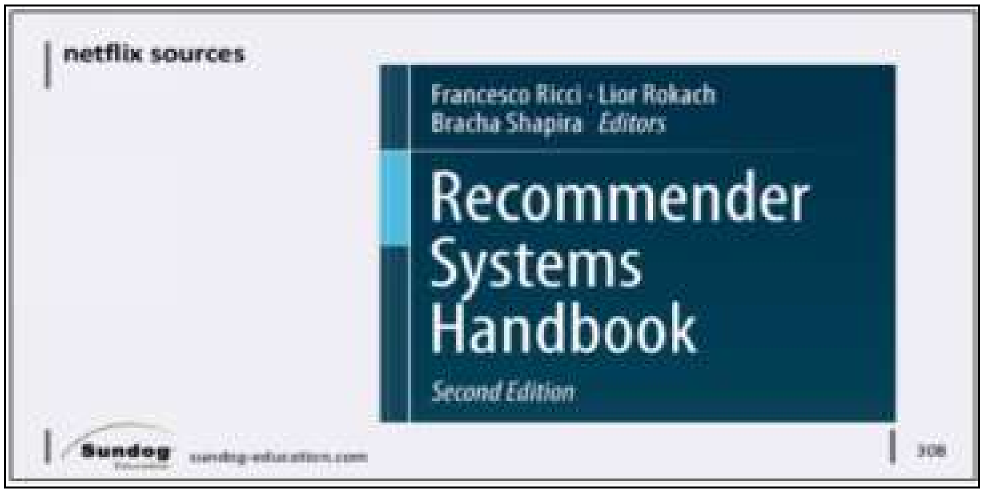


netflix

Sundog Education sundog-education.com

307

Netflix has also published how they're producing recommendations, so let's dive into that next. They're not quite as open as Google is, so the details of how their algorithms work aren't available unless you work there. But, they have shared a lot of higher-level lessons they've learned and the general approach they take.



netflix sources

Francesco Ricci · Lior Rokach
Bracha Shapira *Editors*

Recommender Systems Handbook

Second Edition

Sundog Education sundog-education.com

308

Our main source for this section comes from the book "Recommender Systems

Handbook," specifically chapter 11 which is called "Recommender Systems in Inudstry: A Netflix Case Study." This paper dates from 2015, so I'm sure their approaches have evolved since then – perhaps to include deep learning, but we don't know for sure. More recently, they have also presented at industry conferences such as ACM's RecSys conference – but the actual papers published are very high level in nature.

what model does netflix use?

all of them!

Sundog sundog-education.com

309

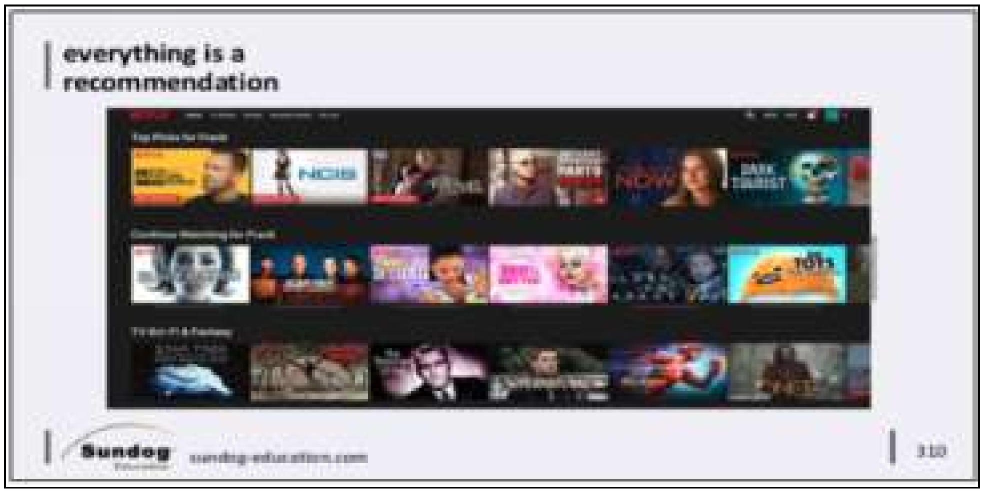
Unlike YouTube, Netflix has no mandate – at least at the time of their writing – to use deep learning for every machine learning problem they encounter. Instead, they bank on a hybrid approach that combines together the results from many different algorithms. The ones they've said they used in the past include RBM and SVD++, which they learned were worthwhile from the Netflix Prize. They've also said that they combine nearest-neighbor approaches with matrix factorization approaches to try and get the best of both worlds .

That's all they've said definitively; although they sort of give us a wink and a nod when they present an "incomplete list of methods that are useful to know" in their paper. This list includes linear regression, logistic regression, elastic nets, SVD, matrix factorization, RBM's, Markov Chains, Latent Dirichlet Allocation, Association Rules, Factorization Machines, Random Forests, Gradient Boosted Decision Trees, k-means, Affinity Propagation, and Dirichlet Processes.

I think this is Netflix's way of saying they basically try every algorithm out there, and let them all fight it out amongst each other when faced with the problem of generating recommendations for a given user.

It’s probably safe to say they’re dabbling in deep learning by now as well.

So while Netflix seems intent on keeping the details of its candidate generation as a trade secret, they do share some higher level learnings, and details on their ranking approach.



One phrase that Netflix uses repeatedly in their publications is “everything is a recommendation.” The Netflix home page is organized into many rows, each row containing its own top-N list of recommendations for a given type of recommendation. For example, my home page includes “top picks,” which presumably are my actual top-N results for me personally. “Continue watching” is recommendations based on videos I’ve already watched – or at least, the people in my family who share my account. I also see a list of recommendations restricted to the genre I seem to be most interested in, which is TV sci-fi and fantasy.

If I scroll down, there are countless more rows associated with recommendations for other categories I may be interested in, recommendations for new releases, for popular items, and for Netflix original series.

Basically, everything is a recommendation – their homepage is just a series of recommender engines tuned for some specific purpose. Netflix has gone all-in with relying on recommender engines to introduce their members to new content.

Having all of these different recommenders on one page introduces its own set of challenges. How do you make sure you don’t repeat the same recommendation on the same page? How do you choose the best order to present these different recommendations in? This means that Netflix not only has to personalize movie and TV recommendations to you, they also have to personalize the order in which these recommendations are presented to you.

This is called whole-page optimization, and using machine learning to optimize the selection of individual rating widgets on slots on a page is a technique that can be powerful for any website. If you look at Amazon.com’s home page, it has a similar structure, and they face the same problem of page optimization to put the right features in the right slots on that page for you.

Another thing Netflix says repeatedly, which we've also said repeatedly in this course, is to not focus on accurately predicting rating values. This is a little bit ironic, given that the focus on rating predictions was really driven by the Netflix Prize itself. Today however, Netflix has done away with star ratings entirely, so they couldn't train on them even if they wanted to. They talk about how they have evolved by initially focusing on rating prediction, to focusing on rankings instead, and finally to whole-page optimization. Although they don't call it out specifically, YouTube isn't trying to predict ratings either – they are optimizing on minutes watched, and their objective function is learned over time from live A/B tests, not on any offline accuracy metric. Netflix also relies on live A/B tests online to tune its systems; they use what they call an "offline-online testing procedure" to use offline metrics to get an initial cut at which new ideas might perform well online, and then use live online A/B tests to validate those ideas on real people. That's basically the approach we've promoted throughout this course.

Netflix also notes that ratings data is inherently noisy, which limits the usefulness of RMSE as a metric for your recommender system. Remember YouTube abandoned the use of explicit ratings entirely, so it seems they both reached the same conclusion here.

Also similar to YouTube, Netflix has separated out the problems of candidate generation with ranking them, and has invested a lot in their own "personalized learning to rank" approach. Learning-to-rank basically treats the problem of generating a top-N list as a classification problem, which each slot in the top N is a classification you are trying to learn. Although they are cagey on the details of how they do this, again they name-drop a few “examples” of how you might do it, including RankSVM, RankBoost, RankNet, and BPR.

They don’t just use the results of their personalized learning to rank system as-is, though. They intentionally factor in the popularity of the results as well at the end, to try and balance predicted rankings with popularity. As we’ve mentioned, showing some popular items is essential to producing user trust in the system, and getting them to engage with it .

As with YouTube, recommendation candidates come along with scores or rating predictions that you could just sort to get a top-N list, but they’ve taken this a step further and applied machine learning to finding the best order to present those final results in, incorporating more information than just those predicted ratings.

Netflix, like YouTube, has found that the more features you train your system with, the better. It’s not just about which videos you watched in the past, it’s also about how you watched them. And the circumstances you’re in right now might affect what sorts of content are best to show to you right now.

For example, are you likely to view different kinds of content depending on which device you’re viewing it on? If you’re accessing

Netflix from your big-screen TV, perhaps that’s the device that you use personally, and recommending longer-form content aimed at adults is what makes sense. If your kid uses your Netflix account on a tablet, making recommendations based on what you’ve watched from that tablet would surface the shorter, kid-focused content that makes sense in that context .

The time might make a difference as well. You might be open to more mature content late at night, than you would in the afternoon when the kids are around. Even the day of the week might matter; you may be looking to kill more time on the weekends, and so you might prefer movies on weekends and TV shows during the week.

These are all examples of the context in which you are seeking recommendations, and a good recommender system will use all of this information to refine the results it returns.

Those are the main points Netflix has shared with us, but they are insightful. It’s also interesting that even though Netflix and YouTube are both in the business of recommending videos, they are doing so in slightly different settings and with slightly different requirements that have informed their choices of algorithms and technology.

Hybrid Recommenders

hybrid approaches

Sundog sundog-education.com

315

As we’ve seen with Netflix, there’s no need to choose a single algorithm for your recommender system. Each algorithm has its own strengths and weaknesses, and you may end up with a better system by combining many algorithms together in what we call a hybrid approach.

ensemble approaches

Sundog sundog-education.com

316

We touched on it earlier, but these hybrid, or ensemble, approaches can make a real difference. The winner of the Netflix Prize came from a group called KorBell, and they won by creating an ensemble of 107 different algorithms that worked together! The top 2 were SVD++ and RBM, both of which ended up in use by Netflix itself.

Another example of where an ensemble approach has been shown to work is with session-based recommendations. Earlier, we covered using recurrent neural networks, or RNN's, to the problem of recommending items as part of a sequence of session data – this was called GRU4Rec. Its results were OK, but actually not as good as a simpler k-nearest-neighbor approach applied to the items in a user's session. However, by combining GRU4Rec with a KNN approach, the resulting system performed better than either approach did individually.

combining behavior
and semantic data

Hybrid approaches can also be useful in solving the "cold start problem." If we can augment recommender systems based on real user behavior such as clicks, views, or ratings with semantic data about the items themselves, we can produce a recommender system that uses behavior data when it's available, but can fall back on content attributes when it needs to.

Coding Exercise

exercise: build a hybrid recommender

Sundog sundog-education.com

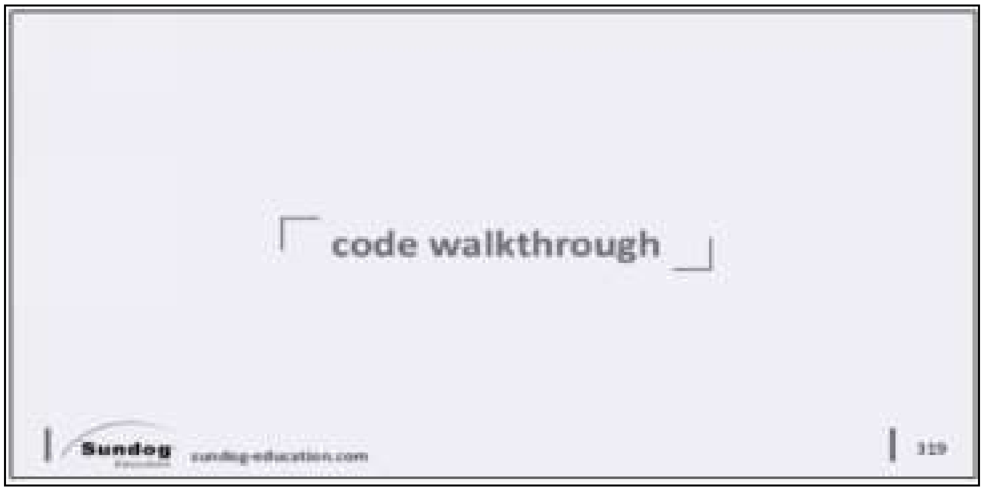
318

For your final exercise in this course, I'll challenge you to build your own hybrid recommender. As with most things, there's more than one way to do it – perhaps you could reserve some slots in your top-N results for certain recommenders. Perhaps you use one recommender primarily, with various fall-backs to use in case the primary recommender could not fill all of your top-N slots. Or perhaps you generate rating predictions from many recommenders in parallel, and add or average their scores together before ranking them.

The latter approach is what I'm going to challenge you to do. Write a new algorithm in the recommender framework we've created for this course, called "HybridAlgorithm." All it should do is take in a list of other algorithms that are compatible with surpriselib and a weight you wish to assign to each, and train each of those algorithms when calling fit() on your HybridAlgorithm. When estimate() is called to generate a rating prediction for a specific user/item pair, compute a weighted average based on each algorithm in the list, and the weights associated with each.

Then, try it out – and see how your HybridAlgorithm performs compared to using its component algorithms individually. Which recommenders you choose to combine is up to you; personally, I chose to combine our RBM algorithm with the Content-based KNN algorithm we developed earlier, in the spirit of combining behavior-based and semantic-based information together into a single system.

It’s easier than it sounds, so try and give it a shot yourself. Up next, I’ll review my solution and results to this problem.


code walkthrough

Sundog sundog-education.com

319

I hope you were able to implement a hybrid recommender on your own! It’s a way to tie together a lot of different things we’ve learned under one umbrella, which is a fitting way to finish this course .

If you’d like to take a look at my solution, open up Spyder and control-shift-W to close out anything else you may have open. Then, open up the contents of the Hybrid folder in the course materials.

Start by looking at the HybridAlgorithm.py file. It’s rather surprisingly small, which is a testament to the architecture of the recommender library we’ve built here. It allows us to easily combine different algorithms that might do totally different things, thanks to having a common interface on every recommender.

The ini t function just takes in a list of algorithms that derive from AlgoBase, and an associated list of weights you want to assign to each one when producing final rating estimates.

All the fit( ) function does is iterate through each algorithm in our list, training each one with our training data set by calling their fit( ) functions in turn.

And there’s not much going on in the estimate( ) function either – all this code does is iterate through each algorithm, calling estimate( ) on each one and combining the resulting estimates in a weighted average, which makes up the final rating estimate that we return. That’s all there is to the heart of this hybrid recommender. It’s a very simple approach, and as we said, there’s more than one way to do it. You could have a hierarchy of recommenders that only fill in individual top-N slots if a better recommender was unable to produce results. Or, you could imagine a system that actually learns the correct weights between different recommenders based on the results on online experiments – or you could even create a system to personalize those weights based on user attributes or contextual information. But for now, this is a good starting point .

Let’s shift our attention to the HybridTest.py file. Nothing much new here – we’re using the same Evaluator module we’ve used throughout the course, only this time we’re comparing the RBMAlgorithm, the ContentKNNAlgorithm, and a HybridAlgorithm that combines the two. Line 40 is where the HybridAlgorithm is created, and you can see we’ve arbitrarily chosen equal weights for both of its component algorithms. Again, in a real system, you’d want to run experiments to find the optimal weights to use – and maybe even personalize those weights, or work in contextual information to them.

Both the RBM and ContentKNNAlgorithm approaches take a fair amount of time to run, and we need to run each one twice for this experiment, so hit the play button to run this, and just let your computer do its thing for an hour or so. We’ll come back when it’s done.

Let’s scroll up to where accuracy from all three are measured – we can see that RBM has a rather poor score of 1.1891. Can we make it better by applying content information to the results as well? Well, the answer is yes! Our hybrid approach outperformed RBM substantially, since our accuracy metric on content-based recs exceeds RBM by quite a bit. But, we know RBM’s score is bad mostly because it under-estimates everything pretty consistently – it has its strengths, when given enough training data. But it really struggles in low-data situations such as ours.

Content-based recommendations don’t care about how much behavior data you have, but of course they have their limits – you want to take individual behavior data into account when you can. Our hybrid approach gives us the best of both worlds, with an accuracy measure that’s in between the two.

As we said, there are examples where a hybrid recommender can outperform either of its constituent algorithms when run independently. Session-based recommendations are an example, as is the ensemble approach that won the Netflix prize. A good hybrid recommender won’t just average out the performance of each one, it will leverage the strengths of some recommenders to augment the weaknesses of others – and leave you with a system that’s more than just the sum of its parts.

We’ve certainly covered a lot in this course so far, and now you’ve seen how to tie these different algorithms together. Developing recommender systems sometimes feels like more of an art than a science, and choosing how to combine different algorithms can feel like choosing different colors in a painting that complement each other. At this point, I hope your palette is filled with lots of different algorithms and ideas surrounding recommender systems, and you’re ready to go apply what you’ve learned to real problems!

More to Explore

learning more

Sundog sundog-education.com

320

Congratulations for making it to the end! Recommender systems are a broad area that sometimes involves some challenging math, algorithms, and concepts. But remember – it's the simpler approaches that tend to work best, in my experience.

With the advent of deep learning and the resurgence in AI, recommender systems will continue to evolve quickly, and you need to stay on top of the latest developments.

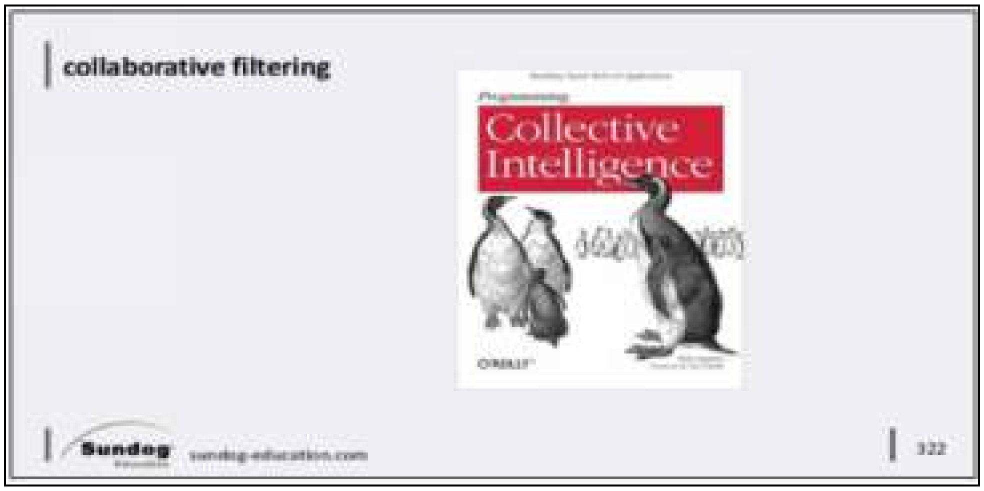
current research:
acm sigkdd

Sundog sundog-education.com

321

ACM's RecSys conference is a great way to stay on top of the latest research in the field. Even if you can't afford to attend in person, it's pretty cheap to join the Association for Computing Machinery and just purchase access to the publications of the SIGKDD group –

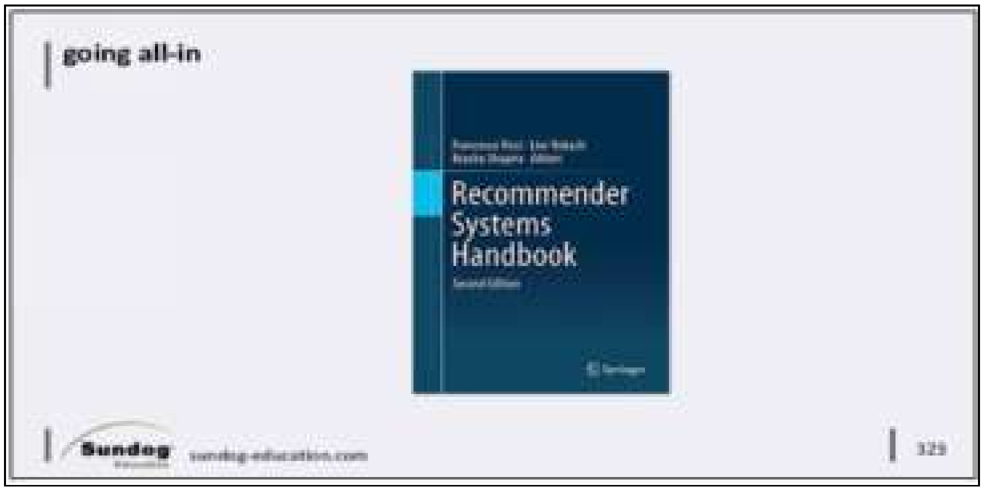
that's the special interest group for Knowledge Discovery and Data Mining, and that gives you access to publications related to recommendation systems online, including the proceedings from every RecSys conference that ever happened. It's only $25 per year as of now.


collaborative filtering

sundog-education.com

322

If you'd like to learn more about the "oldie but goodie" techniques of collaborative filtering, a classic title is "Programming Collective Intelligence" by Toby Segaran, published by O'Reilly.


going all-in

sundog-education.com

329

Books specifically about recommender systems are very few and far between, but another option is the "Recommender Systems Handbook." It's not cheap - $240 for a hardcover copy – but it's over 1,000 pages of papers that cover every aspect of recommender systems in great depth. It's structured as a collection of papers written by a variety of practitioners in the field, but it’s well organized and contains a lot of valuable information.

The only problem is that books like this are obsolete before they even get published; you won’t find any insight in here on applying neural networks or modern tools such as Tensorflow, Amazon DSSTNE, or SageMaker. Many of the topics are timeless, but you’ll want to augment what you learn in this book with more current research from the RecSys conference or by keeping an eye on what deep learning technologies are being offered by the industry giants such as Amazon and Google.

Anyhow, remember this is only the beginning! It’s a fascinating field, and one that’s always evolving. While you’ve reached the end of this course, don’t think of it as the end of your learning about recommender systems – think of it as getting caught up on the current state of the art. And now, you can join the excitement in the field as it discovers new ways to better understand individuals and the things that they love.

You’ve learned a huge amount of valuable knowledge, and you should be proud.

Thank you for sticking with me until the end, and don’t forge t to leave a review for this book! Your written reviews are really essential to my success. Now go forth, and help people discover wonderful things!