Building A Recommender System in Python

Read more about producing engaging videos in the Partner Resource Center.

Core Information:

- Instructor Name: Charles Niswander II
- Project Title: Building A Recommender System in Python

Project Description: What hands-on artifact will learners have developed by the end of this project? Why will they want to know how to do this? Give a 200-300 word description of the project. This will appear on the Coursera platform.	Youtube, Amazon, Google, Netflix.... all of these well-known services are known for their 'magic' algorithms that uncannily predict what videos or movies we would enjoy or what products we might be interested in buying. But how do these recommender systems work? Fortunately, they are simple enough to be understood by the average Python programmer. By the time you've finished "Building A Recommender System in Python", you'll have coded by hand 4 different types of recommender systems that mimic the techniques of Amazon, Netflix, and YouTube.
Project Prerequisites: List any knowledge or skill prerequisites needed to complete the project (e.g. familiarity with matplotlib or Excel spreadsheets)	Python experience, Tensorflow experience is useful
Number of Tasks in Project: Each Task is a Video in a Rhyme project. Each Task should have a clear learning objective, where the learner follows along in a hands-on way.	6
Expected Project length (mins):	~60 min

The whole project, so all Tasks combined, should be <1 hour total. Estimate 150-180 words per minute of video per Task. Suggested length is 4-7 minutes per video.	
Software needed for Rhyme VM instance:	

Project Learning Objectives:

Before you design the Storyboard, list out the project objectives. “By the end of each Task, learners will be able to...”

- LO1: Understand the differences between recommender systems and their use cases
- LO2: Code a simple recommendation engine in Python
- LO3: Code a content-based recommendation engine in Python
- LO4: Code a collaborative-filtering recommendation engine in Python
- LO5: Code a deep-learning based recommendation engine in Python

Tasks:

1. Introducing Recommender Systems
2. Describing Differences Between Recommender Types
    a. Content Filtering vs Collaborative Filtering

    b. Examples of recommender systems

3. Writing a Simple Recommender in Python
4. Writing a Collaborative-Filtering Recommendation Engine in Python
5. Writing a Content-Based Recommender in Python
6. Future Recommenders: Deep Learning Recommenders with Tensorflow

Storyboard

Section/ Scene	**Learning Objective**	**Script**	**What is the visual for this Task? This appears in the VM instance**	**Draft ideas to create quiz questions to test the core concepts for this Task**
Task 1	LO1	Hello there! I'm Charles Niswander, a Python programmer in the area of ML, AI and deep learning. Welcome to "Building A Recommender System in Python". In this project, we'll explore the different types of recommender systems in Python, and by the time	Diagrams (various)	

		we're finished, we will have coded working examples of collaborative-based and content-based recommendation systems, as well as a deep-learning based method using Tensorflow. Over the last few decades, with the advent of Youtube, Amazon, Netflix and many other other web services, recommendation platforms are becoming much more part of our lives. From e-commerce (suggesting to customers articles that could be of interest to them) to web ads (suggesting to consumers the best content, personalized to their preferences), recommendation systems are now commonplace in our everyday online journeys. In a very general way, recommendation systems are algorithms programmed to present related things to users (items being movies to watch, text to read, products to buy or anything else depending on industries). Recommendation systems systems are very important in some industries, because they can produce a large amount of money if they are effective or if they are a way of standing out dramatically from competitors. The aim of the recommendation framework is to produce relevant suggestions for the collection of users of objects or products that may be of interest to them. Suggestions for Amazon books, or Netflix shows, are real-world examples of how industry-leading		

		systems work. The architecture of such recommendation engines depends on the domain and the basic characteristics of the available data. For example, Netflix movie viewers always offer ratings on a scale of 1 (disliked) to 5 (liked). This data source tracks the consistency of experiences between users and objects. A recommendation system generates a compiled list of items that could be of interest to the user in the reciprocity of their existing selection of items (s). It expands the user's recommendations without any interruption or monotony, and does not propose things that the user already recognizes. For example, the Netflix recommendation framework provides suggestions by comparing and searching common user patterns and recommending movies that share characteristics with movies that users scored highly. There are two main kinds of approaches: collaborative filtering methods and content-based methods. Collaborative approaches for recommendation systems are methods that are focused entirely on previous experiences documented between users and items in order to generate new recommendations. These interactions are contained in the "user-item interaction matrix."		

		The core principle that governs collaborative approaches is that these historical user-item encounters are adequate to identify related users and/or similar items and to make forecasts based on these estimated proximities. Collaborative filtering algorithms are classified into two subcategories that are commonly referred to as memory-based and model-based approaches. Memory-based methods function specifically with reported interaction values, assuming no model, and are essentially based on nearest neighbor search (for example, find the closest users in terms of interests and suggest the most popular items among these neighbor's chosen items). However, because it only uses previous experiences to make suggestions, collaborative filtering suffers from a "cold start issue": it is difficult to recommend something to new users or to recommend a new object to any user, because several users or objects have too few interactions to process effectively. This drawback may be tackled in a number of forms, such as recommending random things to new users or new items to random users (random strategy), recommending common items to new users or new items to most successful users (maximum expectation strategy), recommending a collection of different items to new users or a new item to a set of different users (exploratory		

		strategy) or, ultimately, using a non-collaborative method for the early life of the user or the item. Unlike interactive strategies that focus only on user-item experiences, content-based approaches use additional knowledge about users and/or items. If we take the example of a film recommendation system, this additional information could include, for example, age, sex, occupation or some other personal information for users as well as genre, lead stars, length or other characteristics of a movie (items). Then, the concept of content-driven approaches is to try to create a model based on the available "features" that describe the user-item interactions reported. Still considering customers and movies, we might attempt, for example, to model the fact that young women tend to rate some movies better, that young men tend to rate certain other movies better, and so on. If we manage to get such a model, it's pretty easy to make new predictions for the user: we just need to look at this user's profile (age, sex,...) and, based on this data, determine the appropriate movies to recommend. Content-based strategies suffer much less from the cold start dilemma than the collaborative approaches: new users or items can be identified by their characteristics (content) and thus specific recommendations can be made about these new entities Only new		

		users or items with previously unknown features will logically suffer from this downside, but if the system is old enough, there is little or no risk of this happening. In this task, we covered some very basic explanations of recommender systems and what they do. In the next task, we will describe in detail the different forms recommendation systems can take. See you then!		
Task 2		Welcome back! In the last task, we covered some very basic explanations of recommender systems and what they do. In this task, we will describe in detail the different forms recommendation systems can take. First, I'd like to point out that recommenders can take on very simple forms. They often give generalized suggestions to any user, based on popularity of movies and/or genres. The fundamental concept behind this system is that movies that are more popular and critically acclaimed will have a better chance of being liked by the typical viewer. An example of this is the "Trending" section of recommendations, or the IMDB Top 250 (show)	Example lists/web pages of content	

		If you remember from the last task, collaborative filtering methods can take two forms. In memory-based collaborative approaches, no implicit model is assumed. Algorithms deal specifically with user-item interactions: for example, users are represented by their interactions with items, and the nearest neighbors search for these representations are used to generate recommendations. As no implicit model is assumed, these techniques functionally have a low bias but a high variance. Some implicit relational model is assumed in model-based collaborative methods. The model is trained to interpret the values of user-item interactions from its own representation of users and items. Fresh recommendations will then be made based on this model. Users and elements of latent representation derived by the model have a mathematical significance that can be difficult for a human being to interpret. As a (pretty-free) model for user-item interactions is assumed, this approach technically has a higher bias but a lower variance than one assuming no latent model. So, in effect, collaborative filtering identifies how you engage with items, and then identifies other users that behave like you—and then suggests what other users like to you.		

		Some latent interaction model is often assumed for content-based approaches. Here, though, the model is supplied with material that defines the representation of users and/or items: for example, users are represented by specific features and we attempt to model for each item the type of user profile that likes or does not like the item. Here, as with model-based collaboration processes, a user-item interaction model is assumed. However, this model is more confined (because the representation of users and/or items is given) and thus the method tends to have the highest bias but the lowest variance. Essentially, content-based recommendation systems recognize the similarities between items and will suggest items that are comparable to those that the user has used, bought, or engaged with previously. It is important to note that both collaborative filtering and content-based methods can be used in the same system. The integration of each of the two systems in a way that suits a specific business is known as the Hybrid Recommender method. It is a common practice. This is the most sought after Recommender method that many businesses take note of, since it incorporates the benefits of more than two Recommender systems and therefore removes any		

		limitations that occur when only one Recommender system is used. There are many ways in which structures can be mixed, for example: Switching Hybrid Recommender: Switching Hybrid Recommenders move between suggestion techniques dependent on specific criteria. Suppose that if we mix content and collaboratively based advising systems, the switching hybrid advising system will first implement content-based recommendation systems, and if that doesn't work then it can deploy collaboratively based recommendation systems. Mixed Hybrid Recommender: If a sufficient number of suggestions can be made concurrently, we can go for the Mixed Recommendation Systems. Here, suggestions from more than one methodology are provided together, so that the user can select from a wide variety of recommendations. The PTV system, which is primarily a recommendation system to suggest audiences for television viewing, created by Smyth and Cotter, is used by the majority of broadcast and film firms.		

		There is one more option to base a recommendation engine on. Deep learning platforms, such as Tensorflow, can be used to train a model to encode interactions between users and items in what is known as an 'embedding', a low-dimensional space used to encode and transform high-dimensional vectors. This is a common technique in NLP (natural language processing) and other machine learning methods with large and/or sparse inputs. This model will utilize this embedding space to learn the similarity between items, and we can then leverage this combined with further information such as ratings. The input to the neural network in this case is the learned embedding of item-user interactions. A basic encoder-decoder model is used. Our final project for this course will be a Tensorflow-based neural network recommender system. In this task, we reviewed the major conceptual differences between different types of recommendation systems. In the next task, we'll get started with our first simple example system written in Python. See you then!		

Task 3		Welcome back! Charles here. In the last task, we reviewed the major conceptual differences between different types of recommendation systems. In this task, we'll get started with our first simple example system written in Python. As I described in the previous task, simple recommenders are basic recommendation systems that suggest items based on particular metrics, ratings or scores. In this task, I will help you build a simplified clone of IMDB Top 250 Movies using metadata collected from IMDB. The following are the steps involved: 1. Decide on the metric or score to rate movies on. 2. Calculate the score for every movie. 3. Sort the movies based on the score and output the top results. The dataset files contain metadata for all 45,000 movies listed in the Full MovieLens Dataset. The dataset consists of movies released on or before July 2017. This dataset captures feature points like cast, crew, plot keywords, budget, revenue, posters, release dates,	coding in IDLE	

		languages, production companies, countries, TMDB vote counts, and vote averages. These feature points could be potentially used to train your machine learning models for content and collaborative filtering. This dataset consists of the following files: • movies_metadata.csv: This file contains information on ~45,000 movies featured in the Full MovieLens dataset. Features include posters, backdrops, budget, genre, revenue, release dates, languages, production countries, and companies. • keywords.csv: Contains the movie plot keywords for our MovieLens movies. Available in the form of a stringified JSON Object. • credits.csv: Consists of Cast and Crew Information for all the movies. Available in the form of a stringified JSON Object. • links.csv: This file contains the TMDB and IMDB IDs of all the movies featured in the Full MovieLens dataset. • links_small.csv: Contains the TMDB and IMDB IDs of a small subset of 9,000 movies of the Full Dataset. • ratings_small.csv: The subset of 100,000 ratings from 700 users on 9,000 movies.		

		The Full MovieLens Dataset comprises of 26 million ratings and 750,000 tag applications, from 270,000 users on all the 45,000 movies in this dataset. It can be accessed from the official GroupLens website.
		
		On your virtual desktop, you will find an empty script already open. Please navigate to it and we will begin.
		
		You will use the DataFrame pandas library to load the dataset. The pandas library is mostly used for data processing and analysis. This represents the data in a row-column format. The Pandas library is backed up by the NumPy library for the implementation of pandas data objects. Pandas delivers off-the-shelf data models and operations for the manipulation of numerical tables, time-series, imagery and natural language processing datasets. Basically, pandas is useful for datasets that can be easily expressed in tabular form.
		
		let's load your movies metadata dataset into a pandas DataFrame:
		
		# Import Pandas
		import pandas as pd
		
		# Load Movies Metadata
		metadata = pd.read_csv('movies_metadata.csv', low_memory=False)		

```
# Print the first three rows
metadata.head(3)
```

One of the most basic metrics you can think of is the ranking to determine which of the top 250 movies is based on their respective ratings.

However, using a rating as a metric has a few caveats:

For one thing, it does not take into account the popularity of a film. A film with a rating of 9 out of 10 voters would also be rated 'higher' than a film with a rating of 8.9 out of 10,000 voters.

For example, suppose that you want to order Chinese food, you have a few choices, one restaurant has a 5-star rating of only 5 people, while the other restaurant has 4.5 ratings per 1000 people. What restaurant would you like to have? The second, wouldn't it be?

There may be an exception, of course, to the fact that the first restaurant opened only a few days ago; thus, less voters voted for it

		while, on the contrary, the second restaurant has been in business for a year. On a related note, this metric would also tend to favor films with a lower number of voters with biased and/or exceptionally high scores. If the number of voters increases, the ranking of a film regularizes and shifts towards a score that represents the quality of the film and gives the viewer a much better sense of which film he/she should select. Although it is tough to determine the quality of a movie with very few voters, you may have to consider analyzing external sources. Taking these limitations into account, you must apply a weighted rating that takes into account the average rating and the number of votes it has received. Such a system would ensure that a film with a rating of 9 out of 100,000 voters gets a (far) higher score than a movie with the same rating but a mere few hundred voters. Since you're attempting to develop a Top 250 IMDB clone, let's use the weighted rating formula as a metric/score. In this project, you will cut off at the 90th percentile. In other words, for a movie to be featured in the charts, it must have more votes than at least 90% of the movies on the list. (On the other hand, if you had chosen the 75th percentile, you would have considered the		

		top 25% of the movies in terms of the number of votes garnered. As percentile decreases, the number of movies considered will increase).
		
		As a first step, let's calculate the value of the mean rating across all movies using the pandas .mean() function:
		
		# Calculate mean of vote average column
		C = metadata['vote_average'].mean()
		print(C)
		5.618207215133889
		
		From the above output, you can observe that the average rating of a movie on IMDB is around 5.6 on a scale of 10.
		
		Next, let's calculate the number of votes, m, received by a movie in the 90th percentile. The pandas library makes this task extremely trivial using the .quantile() method of pandas:
		
		# Calculate the minimum number of votes required to be in the chart, m
		m = metadata['vote_count'].quantile(0.90)
		print(m)
		160.0		

		you can simply use a greater than equal to condition to filter out movies having greater than equal to 160 vote counts:
		
		You can use the .copy() method to ensure that the new q_movies DataFrame created is independent of your original metadata DataFrame. In other words, any changes made to the q_movies DataFrame will not affect the original metadata data frame.
		
		# Filter out all qualified movies into a new DataFrame
		q_movies = metadata.copy().loc[metadata['vote_count'] >= m]
		q_movies.shape
		(4555, 24)
		metadata.shape
		(45466, 24)
		From the above output, it is clear that there are around 10% movies with vote count more than 160 and qualify to be on this list.
		
		Next and the most important step is to calculate the weighted rating for each qualified movie. To do this, you will:
		
		Define a function, weighted_rating();
		Since you already have calculated m and C you will simply pass them as an argument to the function;
		Then you will select the vote_count(v) and vote_average(R) column from the q_movies data frame;		

		Finally, you will compute the weighted average and return the result.
		You will define a new feature score, of which you'll calculate the value by applying this function to your DataFrame of qualified movies:
		
		# Function that computes the weighted rating of each movie
		def weighted_rating(x, m=m, C=C):
		    v = x['vote_count']
		    R = x['vote_average']
		    # Calculation based on the IMDB formula
		    return (v/(v+m) * R) + (m/(m+v) * C)
		# Define a new feature 'score' and calculate its value with \`weighted_rating()\`
		q_movies['score'] = q_movies.apply(weighted_rating, axis=1)
		Finally, let's sort the DataFrame in descending order based on the score feature column and output the title, vote count, vote average, and weighted rating (score) of the top 20 movies.
		
		#Sort movies based on score calculated above
		q_movies = q_movies.sort_values('score', ascending=False)
		
		#Print the top 15 movies
		q_movies[['title', 'vote_count', 'vote_average', 'score']].head(20)		

		Since the chart has a lot of movies in common with the IMDB Top 250 chart: for example, the top two movies, "Shawshank Redemption" and "The Godfather" are the same as IMDB, and we all know they're pretty great movies, in truth, all the top 20 movies deserve to be on that list, right? Unfortunately, this doesn't allow for much variation or personalization between users. Collaborative filtering based systems are preferable for that. In this task, you built a very simple movie recommender system. In the next task, I will show you a simple collaborative filtering method. See you then!		
Task 4		Welcome back! In the last task, you built a very simple movie recommender system. In this task, I will show you a simple collaborative filtering method. Recall that there are two potential categories of collaborative-filtering techniques. They are memory-based and model-based.	coding in IDLE	

		Memory-based approaches can be focused on user-user relationships or item-item relationships. The main characteristics of user-user and item-item approaches it that they use only information from the user-item interaction matrix and they assume no model to produce new recommendations. In order to make a new suggestion to the user, the user-user approach aims to classify users with the most closely related "interaction profile" (nearest neighbors) in order to recommend items that are the most popular among these neighbors (and that are "new" to our end user). This approach is said to be "user-centered" since it analyzes users on the basis of their interactions with items, and calculates distances between users. Suppose we try to make a suggestion for a specific user. In the first place, each user can be represented by his/her vector of interactions with the various items ("its line" in the interaction matrix). Then we can measure some kind of "similarity" between our user of interest and all other users. This measure of similarity is such that two users with comparable interactions on the same items can be considered to be near to one another. After the similarities to each user have been computed, we will retain the k nearest neighbors to our user and then recommend the most popular items among them (only looking at the items that our reference user has not interacted with yet).		

		To make a new suggestion to the customer, the idea of the item-item approach is to pinpoint items similar to those that the user has previously "positively" engaged with. Two items are deemed to be similar if the majority of users who interacted with both of them did so in a similar way. This approach is considered to be "item-centered" since it represents items based on user interactions with them and calculates distances between these. We will work on the MovieLens dataset and build a model to recommend movies to the end users. This data has been collected by the GroupLens Research Project at the University of Minnesota. The dataset can be found in the folder on the desktop. This dataset consists of: ● 100,000 ratings (1–5) from 943 users on 1682 movies ● Demographic information of the users (age, gender, occupation, etc.) Please start a new script by navigating to File > New. Go ahead and save the file in Documents or the desktop as something like collaborative_filtering.py. First, we'll import our standard libraries and read the dataset in Python.		

		import pandas as pd
		import matplotlib
		import matplotlib.pyplot as plt
		import numpy as np
		
		# pass in column names for each CSV as the column name is not given in the file and read them using pandas.
		# You can check the column names from the readme file
		#Reading users file:
		u_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']
		users = pd.read_csv('ml-100k/u.user', sep='|',
		names=u_cols,encoding='latin-1')
		
		#Reading ratings file:
		r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
		ratings = pd.read_csv('ml-100k/u.data', sep='\t',
		names=r_cols,encoding='latin-1')
		
		#Reading items file:
		i_cols = ['movie id', 'movie title' ,'release date','video release date',
		'IMDb URL', 'unknown', 'Action', 'Adventure',
		'Animation', 'Children\'s', 'Comedy', 'Crime', 'Documentary', 'Drama',
		'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi',
		'Thriller', 'War', 'Western']		

		items = pd.read_csv('ml-100k/u.item', sep='|',
		names=i_cols,encoding='latin-1')
		
		After loading the dataset, we should look at the content of each file (users, ratings, items). In the IDLE shell, type:
		print(users.shape)
		users.head()
		
		So, we have 943 users in the dataset and each user has 5 features, i.e. user_ID, age, sex, occupation and zip_code. Now let’s look at the ratings file.
		
		Then again in the shell, type:
		 print(ratings.shape)
		ratings.head()
		
		We have 100k ratings for different user and movie combinations. Now finally examine the items file.
		
		Finally, again in the shell, check the items:
		print(items.shape)
		items.head()		

		This dataset includes 1682 movie attributes. There are 24 columns out of which the last 19 columns define the genre of a given movie. These are binary columns, — in other words a value of 1 means that the movie belongs to the genre, and a value of 0 indicates otherwise. The data set has already been split into train and test by GroupLens, where the test data has 10 ratings for each user, i.e. a total of 9,430 rows. We're going to read both of these files in the Python environment. r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp'] ratings_train = pd.read_csv('ml-100k/ua.base', sep='\t', names=r_cols, encoding='latin-1') ratings_test = pd.read_csv('ml-100k/ua.test', sep='\t', names=r_cols, encoding='latin-1') ratings_train.shape, ratings_test.shape Now that we have prepped our dataset, we can build our collaborative filtering recommendation engine!		

		We will recommend movies based on user-user similarity and item-item similarity. For that, first we need to calculate the number of unique users and movies. n_users = ratings.user_id.unique().shape[0] n_items = ratings.movie_id.unique().shape[0] Now, we will create a user-item matrix which can be used to calculate the similarity between users and items. we will first initialize it with zeros array of shape 943 x 1643 having 943 users and 1643 movies then we will loop in the ratings data frame row by row. data_matrix = np.zeros((n_users, n_items)) for line in ratings.itertuples(): data_matrix[line[1]-1, line[2]-1] = line[3] here, line[1] is the userId and we are subtracting 1 from it since array indexing starts from 0 = row line[2]-1 is the movie id = column now at that specifec row and column i.e, user and movie we will add line[3] which is the movie rating Now, when we have rating os all the movies given by each user in a matrix we will calculate the similarity. We can use the		

		pairwise_distance function from sklearn to calculate the cosine similarity.
		from sklearn.metrics.pairwise import pairwise_distances
		user_similarity = pairwise_distances(data_matrix, metric='cosine')
		item_similarity = pairwise_distances(data_matrix.T, metric='cosine')
		This gives us the item-item and user-user similarity in an array form. The next step is to make predictions based on these similarities. Let’s define a function to do just that.
		
		def predict(ratings, similarity, type='user'):
		
		    if type == 'user':
		        mean_user_rating = ratings.mean(axis=1).reshape(-1,1)
		        #We use np.newaxis so that mean_user_rating has same format as ratings
		
		        ratings_diff = (ratings - mean_user_rating)
		        pred = mean_user_rating + similarity.dot(ratings_diff) / np.array([np.abs(similarity).sum(axis=1)]).T
		
		    elif type == 'item':
		        pred = ratings.dot(similarity) / np.array([np.abs(similarity).sum(axis=1)])
		
		    return pred		

		here, we are taking mean of axis=1 / rows Finally, we will make predictions based on user similarity and item similarity. user_prediction = predict(data_matrix, user_similarity, type='user') item_prediction = predict(data_matrix, item_similarity, type='item') You can print these if you'd like. From here, your recommender system would pass the result to the part of your system that can use it, as one part of a larger system, such as a website or mobile app. Collaborative filtering works on the interactions that users have with items. These interactions can help to discover correlations that the data about the items or the users themselves can't do. Here are several points that will help you determine whether to use collaborative filtering: Collaborative filtering does not require information about objects or consumers to be known. It is ideal for a set of different types of		

		items, such as a retail inventory where items of different categories can be included. However, in a collection of related products such as those of a bookstore, known features such as authors and genres may be useful and may benefit from content-based or hybrid approaches. Netflix's recommender system filtering architecture is based on collaborative filtering. Now you have the ability to develop a similar system. Remember, some substitute method must be used at the start, because of the "cold start" problem! In this task, we completed a Python script for a recommendation system based on collaborative filtering. In the next script, we will code another recommender system based on content filtering. See you then!		
Task 5		Welcome back! Charles here. In the last task, we completed a Python script for a recommendation system based on collaborative filtering. In this script, we will code another recommender system based on content filtering.	coding in IDLE	

		A content-based recommendation system works with user input, either explicitly (rating) or implicitly (clicking on a link). Based on this data, a user profile is generated, which is then used to make recommendations to the user. As the user gives more inputs or takes action on these recommendations, the engine gets more and more effective. For this task, we will build a recommender system for a retail dataset. We will use what is known as a TF-IDF Vectorizer to calculate scores for items. The TF*IDF algorithm is used to weigh a keyword in any document and to assign value to that keyword depending on the number of times it appears in the document. Generally speaking, the higher the TF*IDF (weight) score, the rarer and more significant the word, and vice versa. Each word or term has its respective TF and IDF total score. The product of the word TF and IDF scores is referred to as the TF*IDF weight of that term. The TF (term frequency) of a word is the number of times that it appears in a text. If you know that, you will see if you use a word too frequently or too seldom.		

		Mathematically: TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).
		
		The IDF (inverse document frequency) of a word is the measure of how significant that term is in the whole corpus. Mathematically: IDF(t) = log_e(Total number of documents / Number of documents with term t in it).
		
		In Python, scikit-learn provides you with a pre-built TF-IDF vectorizer that calculates the TF-IDF score for the description of each text, word-by-word. Basically, our recommender system will use these metrics to compare the similarity of items based on their text descriptions.
		
		After loading the appropriate libraries, lets initialize our vectorizer.
		
		`import pandas as pd`
		`from sklearn.feature_extraction.text import TfidfVectorizer`
		`from sklearn.metrics.pairwise import linear_kernel`
		`ds = pd.read_csv(r"C:\Users\darf3\Documents\FLG Work\Rhyme Recommender System/sample-data.csv")`
		
		`tf = TfidfVectorizer(analyzer='word', ngram_range=(1, 3), min_df=0, stop_words='english')`
		`tfidf_matrix = tf.fit_transform(ds['description'])`		

		Here, the tfidf_matrix is the matrix containing each word and its TF-IDF score with regard to each document, or item in this case. Also, stop words are simply words that add no significant value to our system, like ‘an’, ‘is’, ‘the’, and hence are ignored by the system. Now, we have a representation of each item in terms of its description. Next, we need to measure the similarity or relevance of a text to another text. To do this, we will calculate their cosine similarity. In this model, each item is stored as a vector of its attributes (which are also vectors) in an n-dimensional space, and the angles between the vectors are calculated to determine the similarity between the vectors. The method of calculating the user’s likes / dislikes / measures is calculated by taking the cosine of the angle between the user profile vector (Ui) and the document vector; or in our case, the angle between two document vectors. The ultimate reason behind using cosine is that the value of cosine will increase as the angle between vectors with decreases, which signifies more similarity.		

		The vectors are length-normalized, after which they become vectors of length 1.
		
		cosine_similarities = linear_kernel(tfidf_matrix, tfidf_matrix) results = {}
		for idx, row in ds.iterrows():
		  similar_indices = cosine_similarities[idx].argsort()[:-100:-1]
		  similar_items = [(cosine_similarities[idx][i], ds['id'][i]) for i in similar_indices]
		  results[row['id']] = similar_items[1:]
		
		Here, we computed the cosine similarity of each item with each other in the dataset, then ordered them according to their similarity to item i and stored the values in results.
		
		Now we only have to compute and output the recommendation.
		
		Here, you will just input an item_id and the number of recommendations that we want, and voilà! Our function collects the results[] corresponding to that item_id, and we get our recommendations on screen. Of course, the output can be sent elsewhere and used by another system.
		
		def item(id):		

```
  return ds.loc[ds['id'] == id]['description'].tolist()[0].split(' - ')[0]

# Just reads the results out of the dictionary.def
recommend(item_id, num):
    print("Recommending " + str(num) + " products similar to " +
item(item_id) + "...")
    print("-------")    recs = results[item_id][:num]
    for rec in recs:
       print("Recommended: " + item(rec[1]) + " (score:" +      str(rec[0])
+ ")")
```

Now you can call the function:

```
recommend(item_id=11, num=5)
```

Let's review some pros and cons to content-based filtering, so we might better understand why a mixed hybrid system is ideal.

Advantages in content-based filtering

- User independence: Collaborative filtering requires ratings from other users to identify correlations between users and then make recommendations. Instead, the content-based approach simply has to evaluate the products and the suggestion profile of a particular individual, making the

		procedure less tedious. Content-based filtering can then deliver more accurate results for fewer users in the system. • Transparency: Collaborative filtering offers suggestions based on other anonymous users who have the same taste as the user, but content-based filtering is recommended on a feature-level basis. • No cold start: as opposed to collaborative filtering, new items can be recommended before a large number of users have rated. Disadvantages of content based filtering • Limited content analysis: if the content does not contain enough information to differentiate precisely between items, the recommendation itself risks being imprecise. • Over-specialization: content-based filtering has a limited degree of novelty, as it must fit the characteristics of the user profile with the available items. In the case of item-based filtering, only item profiles are created and consumers are recommended items that are close to what they rate or search for regardless of their past history. A great content-based filtering scheme could not recommend anything unpredictable or surprising. In this task, you coded a fully functional prototype for a content-based recommendation system. In the next task, you will		

		code a modern deep-learning based recommender using neural networks trained with Tensorflow. See you then!		
Task 6		Welcome back! Charles here. In this task, you coded a fully functional prototype for a content-based recommendation system. In the next task, you will code a modern deep-learning based recommender using neural networks trained with Tensorflow. TensorFlow is a free, open-source machine learning software library. It can be used across a variety of functions, but has a special emphasis on the training and inference of deep neural networks. Tensorflow is a symbolic math library focused on data flow and differentiable programming. It's an open source artificial intelligence library that uses data flow graphs to construct models. It enables developers to write large-scale neural networks with multiple layers. The recommender system we are going to build with Tensorflow is based on the recommendation algorithm outlined in the famous YouTube videos recommendation paper — Deep Neural Networks for YouTube Recommendations.	Coding in a Jupyter Notebook	

		We will do this one in the form of a Jupyter IPYNB. This way, some of the boilerplate will be ready for you. On your desktop, you will find a file called "youtube_recommender.ipynb". Please double click it. This should bring up the Jupyter interface in the Chrome browser.
		
		The first cell is empty. Please follow along with me to fill it out.
		
		# Get the data from Movielens website
		from urllib.request import urlretrieve
		import zipfile
		import pandas as pd
		
		urlretrieve("http://files.grouplens.org/datasets/movielens/ml-100k.zip", "movielens.zip")
		zip_ref = zipfile.ZipFile('movielens.zip', "r")
		zip_ref.extractall()
		print("Done. Dataset contains:")
		print(zip_ref.read('ml-100k/u.info'))
		
		#Process the dataset for movies, users,ratings and genre
		# Load each data set (users, movies, and ratings).
		users_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']
		users = pd.read_csv(		

```
    'ml-100k/u.user', sep='|', names=users_cols, encoding='latin-1')

ratings_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings = pd.read_csv(
    'ml-100k/u.data', sep='\t', names=ratings_cols, encoding='latin-1')

# The movies file contains a binary feature for each genre.
genre_cols = [
    "genre_unknown", "Action", "Adventure", "Animation", "Children",
"Comedy",
    "Crime", "Documentary", "Drama", "Fantasy", "Film-Noir",
"Horror",
    "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller", "War",
"Western",
]
movies_cols = [
    'movie_id', 'title', 'release_date', "video_release_date", "imdb_url"
] + genre_cols
movies = pd.read_csv(
    'ml-100k/u.item', sep='|', names=movies_cols, encoding='latin-1')

# Since the ids start at 1, we shift them to start at 0.
users["user_id"] = users["user_id"].apply(lambda x: str(x-1))
movies["movie_id"] = movies["movie_id"].apply(lambda x: str(x-1))
```

```
movies["year"] = movies['release_date'].apply(lambda x: str(x).split('-')[-1])
ratings["movie_id"] = ratings["movie_id"].apply(lambda x: str(x-1))
ratings["user_id"] = ratings["user_id"].apply(lambda x: str(x-1))
ratings["rating"] = ratings["rating"].apply(lambda x: float(x))
```

In the next cell:

```
#Get all the genres for a movie
import numpy as np
genre_occurences = movies[genre_cols].sum().to_dict()

genres_encoded = {x: i for i, x in enumerate(genre_cols)}

def get_genres(movies, genres):
  def get_all_genres(gs):
    active = [str(genres_encoded[genre]) for genre, g in zip(genres, gs) if g==1]
    if len(active) == 0:
      return '0'
    return ','.join((active))
  movies['all_genres'] = [
      get_all_genres(gs) for gs in zip(*[movies[genre] for genre in genres])]
```

		get_genres(movies, genre_cols)
		
		Now you can run step this through until you come to the next empty cell. In that cell, we will piece together the layers of our network.
		
		#---inputs
		import tensorflow as tf
		import datetime
		import os
		input_title = tf.keras.Input(shape=(None,), name='movie_name')
		inp_video_liked = tf.keras.layers.Input(shape=(None,), name='like')
		inp_video_disliked = tf.keras.layers.Input(shape=(None,), name='dislike')
		input_genre = tf.keras.Input(shape=(None,), name='genre')
		
		#--- layers
		features_embedding_layer = tf.keras.layers.Embedding(input_dim=NUM_CLASSES, output_dim=EMBEDDING_DIMS,
		                                        mask_zero=True, trainable=True, name='features_embeddings')		

		labels_embedding_layer =
		tf.keras.layers.Embedding(input_dim=NUM_CLASSES,
		output_dim=EMBEDDING_DIMS,
		                                        mask_zero=True, trainable=True,
		name='labels_embeddings')
		
		avg_embeddings =
		MaskedEmbeddingsAggregatorLayer(agg_mode='mean',
		name='aggregate_embeddings')
		
		dense_1 = tf.keras.layers.Dense(units=DENSE_UNITS,
		name='dense_1')
		dense_2 = tf.keras.layers.Dense(units=DENSE_UNITS,
		name='dense_2')
		dense_3 = tf.keras.layers.Dense(units=DENSE_UNITS,
		name='dense_3')
		l2_norm_1 = L2NormLayer(name='l2_norm_1')
		
		dense_output = tf.keras.layers.Dense(NUM_CLASSES,
		activation=tf.nn.softmax, name='dense_output')
		
		#--- features
		features_embeddings = features_embedding_layer(input_title)
		l2_norm_features = l2_norm_1(features_embeddings)
		avg_features = avg_embeddings(l2_norm_features)		

```
labels_liked_embeddings = labels_embedding_layer(inp_video_liked)
l2_norm_liked = l2_norm_1(labels_liked_embeddings)
avg_liked = avg_embeddings(l2_norm_liked)

labels_disliked_embeddings =
labels_embedding_layer(inp_video_disliked)
l2_norm_disliked = l2_norm_1(labels_disliked_embeddings)
avg_disliked = avg_embeddings(l2_norm_disliked)

labels_genre_embeddings = labels_embedding_layer(input_genre)
l2_norm_genre = l2_norm_1(labels_genre_embeddings)
avg_genre = avg_embeddings(l2_norm_genre)



concat_inputs = tf.keras.layers.Concatenate(axis=1)([avg_features,
                                        avg_liked,
                                        avg_disliked,
                                        avg_genre
                                        ])
# Dense Layers

dense_1_features = dense_1(concat_inputs)
```

		dense_1_relu = tf.keras.layers.ReLU(name='dense_1_relu')(dense_1_features) dense_1_batch_norm = tf.keras.layers.BatchNormalization(name='dense_1_batch_norm')(de nse_1_relu) dense_2_features = dense_2(dense_1_relu) dense_2_relu = tf.keras.layers.ReLU(name='dense_2_relu')(dense_2_features) dense_3_features = dense_3(dense_2_relu) dense_3_relu = tf.keras.layers.ReLU(name='dense_3_relu')(dense_3_features) dense_3_batch_norm = tf.keras.layers.BatchNormalization(name='dense_3_batch_norm')(de nse_3_relu) outputs = dense_output(dense_3_batch_norm) #Optimizer optimiser = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE) #--- prep model model = tf.keras.models.Model(inputs=[input_title, inp_video_liked,		

		inp_video_disliked ,input_genre], outputs=[outputs]) logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S")) tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1) model.compile(optimizer=optimiser, loss='sparse_categorical_crossentropy') Then, in the next empty cell, we can call fit() on our model. model.fit([tf.keras.preprocessing.sequence.pad_sequences(user_title_list_e['title_d']), tf.keras.preprocessing.sequence.pad_sequences(user_title_list_e['like']), tf.keras.preprocessing.sequence.pad_sequences(user_title_list_e['dislike']) ,		

		tf.keras.preprocessing.sequence.pad_sequences(user_title_list_e['all_genres'])],user_title_list_e['predict_labels'].values,callbacks=[tensorboard_callback], steps_per_epoch=1, epochs=1000,verbose=1) Note that we have padded our input. We've also added a Tensorboard callback. The next empty cell will allow us to use our model to make predictions. predict =model.predict([tf.keras.preprocessing.sequence.pad_sequences(user_list_1['title_d']), tf.keras.preprocessing.sequence.pad_sequences(user_list_1['like']), tf.keras.preprocessing.sequence.pad_sequences(user_list_1['dislike']),		

		tf.keras.preprocessing.sequence.pad_sequences(user_list_1['all_genres'])]) Note that we use tf.nn.top_k to gather the top 'k' predictions. The final portion creates visualizations of the embeddings we've trained. Go ahead and run through the entire script, every step. What do you think? You've officially coded several types of recommender systems by hand. By now, you should have several working recommender prototypes. It is up to you to adapt them to your needs and combine them in a way that will most benefit your purposes. Congratulations on finishing "Building A Recommender System in Python"! Keep up the good work and keep coding! I wish you the best in your ML adventures!		