**Recommendation System using Python**

**1. Collaborative filtering (CF)**

* User based CF
* Item based CF

**2. Content based filtering**

**3. Neighbourhood based collaborative filtering**

**4. Model based filtering (SVD) Matrix factorization approach to**

**generate recommendations**

**Singular Value Decomposition (SVD)**

**5. Recommendations with Deep Learning (not covered)**

Ex. YouTube, Netflix

**Many flavours of recommenders:**

* Recommending things (Gas-Gas pipe, TV-Dish Antenna)
* Recommending contents (Newspaper contents to recommend)
* Recommending music (based on music properties to recommend)
* Recommending people (Dating sites)
* Recommending search results (More personalized search result contents based on your past search contents)

**Understanding users through Explicit and Implicit Ratings**

1. Explicit

* Asking users to rate and give reviews

2. Implicit

* We do it anyhow (Things we purchase (purchase products from amazon, flip-kart),

Things we consume (YouTube minutes watched),

Things we click on (clicking links irrespective of we like it or don't like it)

**Item based Collaborative filtering CF**

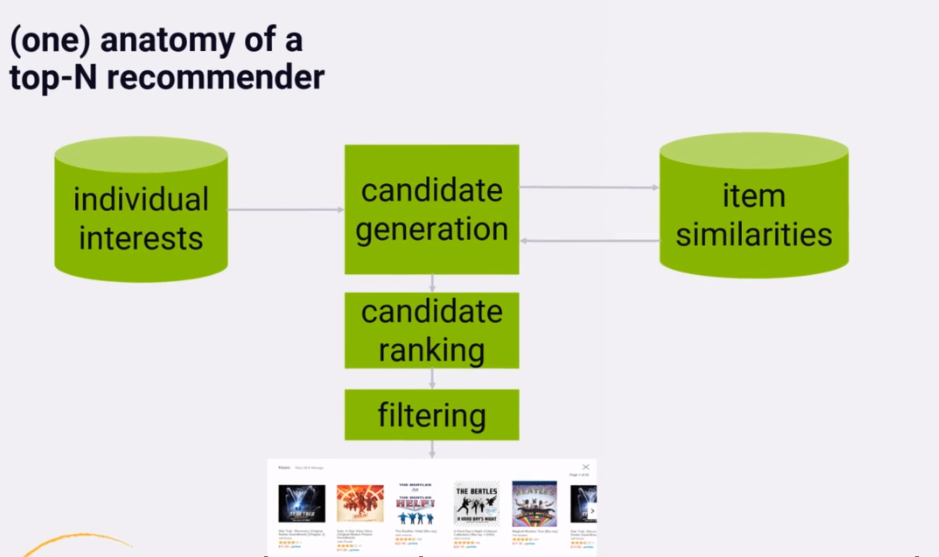
**Top N Recommender Systems:**

* Finite list of the best things to present to a given person.

**Recommendation candidate:**

* Items we think might be interesting to the user based on their past behaviour.
* Recommendation candidate generation take the entire items user indicated
* Interesting before and construct a new dataset of items similar to previous
* Items of aggregate interest behaviour.

Data store of Individual Interests of each user -> Generate Recommendation candidate -> Item Similarities



Ex:

Consider we are making recommendations for Mr. Kapil, then we might consult Mr. Kapil's database of individual interests and check whether he liked "Bourne Identity" movie in the past. Based on everyone else's behaviour we know that the people who like "Bourne Identity" movie also liked the movie "Taken". So based on his interest on "Bourne Identity" he might get recommendations candidates which contains movie "Taken" in it. In the process of building those recommendations we might assign some scores to each candidate based on how he rated the items they came from and how strong the similarities are between the item and the candidates are came from them. We might filter out candidates whose score is not high enough.

**Candidate Ranking:**

Many candidates will appear more than once and need to combine together in some way. May be boosting their score in the process since they keep coming up repeatedly. After that we sort the resulting recommendations candidates by score, then there are top N list of recommendations.

Many more approaches are exists for just learning the rank:

Ex. Machine Learning employed to find the optimal ranking of the candidates.

**Filtering:**

Some filtering will be required before presenting the final sorted list of recommendations 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 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 is also where we apply the N in top N recommenders and cut things of if we have more results than we need.

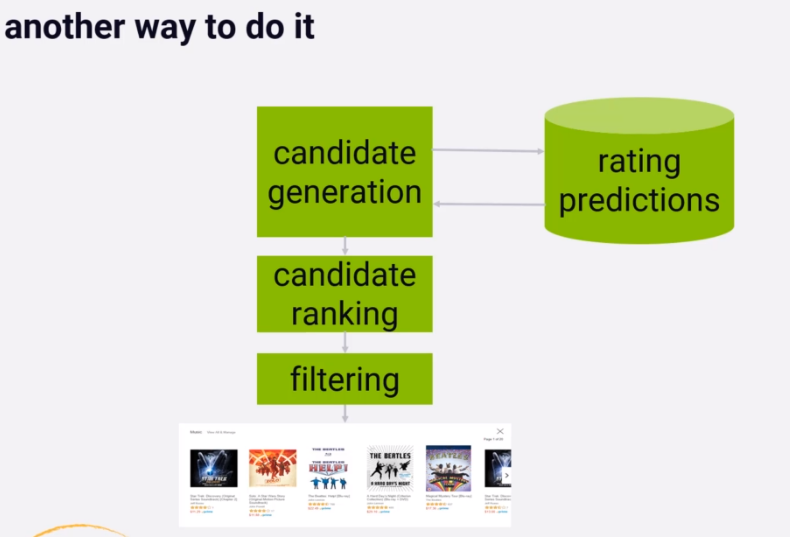
The output of the filtering stage is handed off to the display layer where pretty product recommendations presented to the user.

So all these three, Candidate Generation, candidate ranking and filtering will live inside some distributed recommendation web service that our web frontend talks to for the process of rendering a page for specific user.

This diagram is simplified version of item based collaborative filtering, and it’s the same algorithm published by amazon in 2003. The hard part here is making the similarities data set.

And this is one of the way to do it.

**Another way:**



In this we work on to build up a database ahead of time of predicted ratings of every item by every user. The Candidate Generation phase is that just retrieving all of the rating predictions for given user for every item and ranking is sorting them.

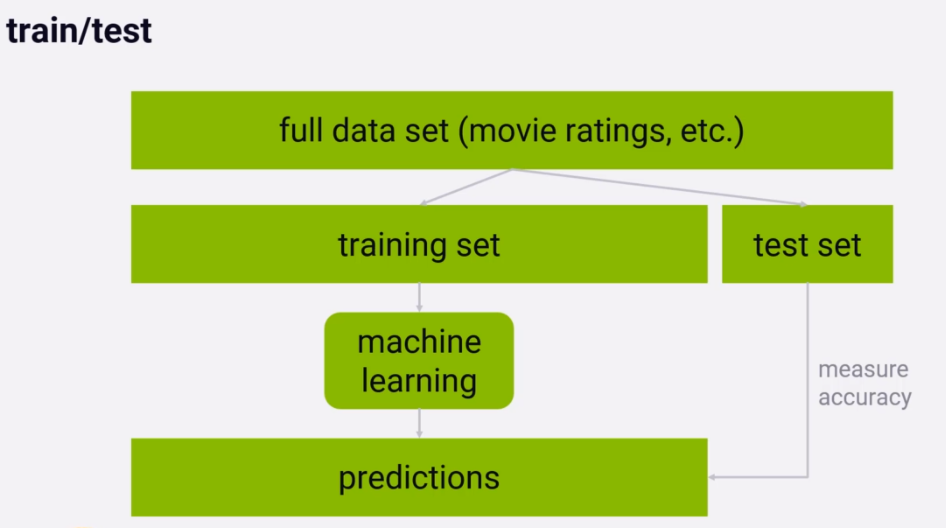
This requires us to look at every single item in our catalog for every single user however which isn't efficient at one time. This approach will work for small dataset and not for huge data set.

**Evaluating Recommender Systems**

**1. Train/Test and Cross Validation**

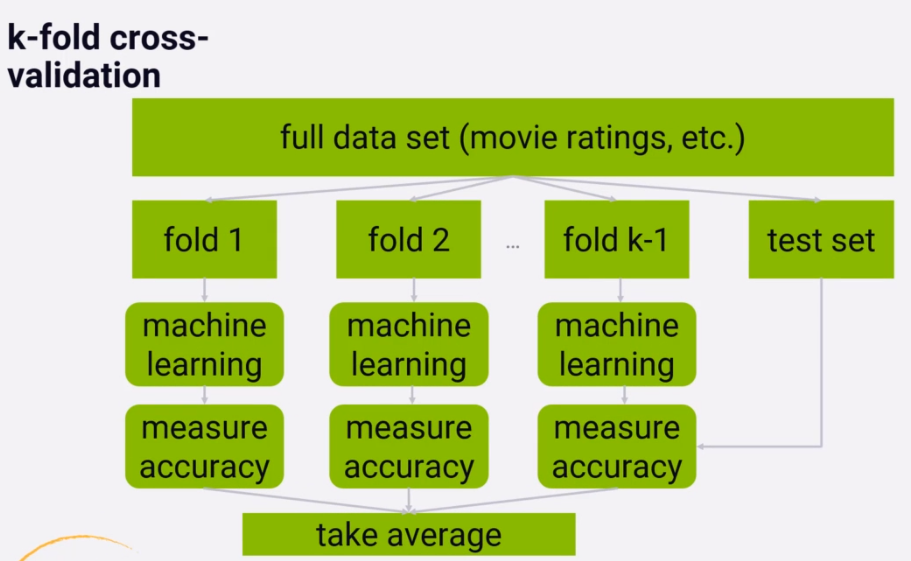
Testing recommender system offline

**Train/Test splits**



It is possible to improve on a single train test split by using a technique called:

**K-Fold Cross Validation**.



It is the same idea as train-test split but instead of single training set we create many randomly assigned training sets. Each individual **training set or fold** is used to train our recommender system independently and then we measure the accuracy of the resulting systems against the test set.

So we end up with the score how accurately each fold ends up predicting user ratings and we can average them together. This obviously will take a lot more computing power to do but the advantage is that we don't end up over-fitting to a single training set.

If our training data is small we are under the risk of optimizing for the ratings that are specifically in our training set instead of test set.

So K-Fold Cross Validation provides some insurance against that and insures that our created recommender system that works for any set of ratings and not the just the ones in the training set that we happened to choose.

So both methods are ways to measure accuracy of our recommendation system.

That is how accurately we can predict rated movies they have already seen and provided the rating for.

By using train-test all we can do is to test our ability to predict how people rated movies they already saw.

**This is not the point of recommendation system; we want to recommend new things**

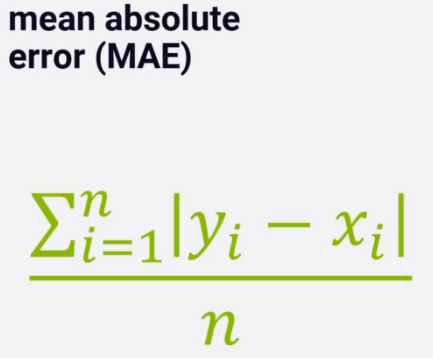
**to the people they haven't seen.**

**So this is fundamentally impossible to test recommendation system offline.**

**Accuracy metrics of Recommendation system (RMSE, MAE)**

**1. MAE (Mean Absolute Error)**

This is the most straight forward metric.



**n - n ratings in the test set we want to evaluate.**

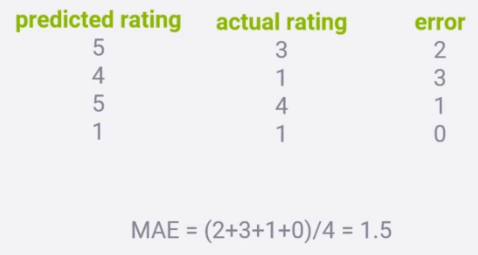
**Y - for each rating our system predicts the rating y.**

**x - the rating user actually gave is x.**

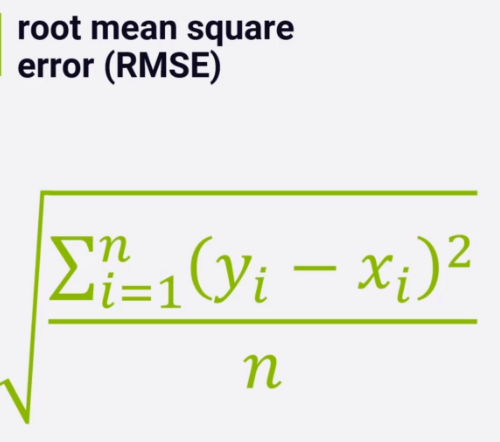
Just taking absolute value of the difference between the two y and x to measure the error for that rating prediction. It is just the difference between predicted rating and actual rating.

We sum that error result across all n ratings in our test set divided by n to get the average or mean. So MAE is exactly this.

Error is bad so, **we want the** **lowest MAE score** we can get and not the highest.



**2. RMSE (Root Mean Square Error)**

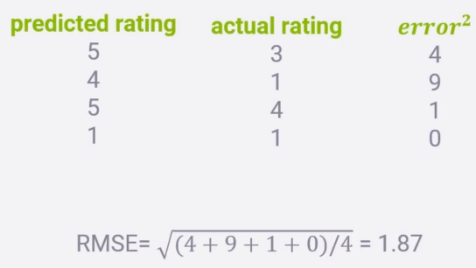


This is more popular metric due to some reasons.

**But it will penalizes us more when prediction is way off and penalizes us less when we are**

**reasonably close.**

Here we take the square errors of the differences between predicted rating y and actual rating x, sum them and divide the sum by n to get the mean and finally take the square root of the mean which is RMSE.



**Accuracy isn’t we want recommendation systems to do instead it should give the best movies recommendation listing.**

**2. TOP-N Hit Rate (Evaluating top-N recommenders)**

User actually rated movies will be termed as Hit.

Add up all hits in our top-n recommendations for every user in our test set divide by the numbers of users and that’s out **Hit rate.**



**3. Leave-one-out cross validation**

Here we cannot use our regular train-test-split or k-fold cross validation approach which we have used to measure accuracy because we are not measuring the accuracy on individual rating instead we are measuring the accuracy on top-n list of ratings for individual users.

Here we compute the top-n recommendations for each user on training data and intentionally remove one of those items from the user’s training data.

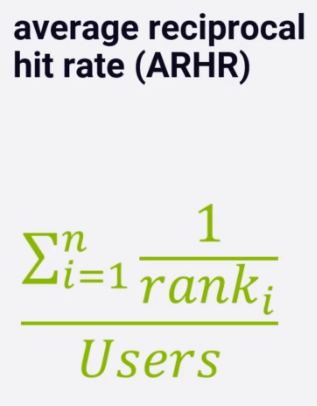
Then we test our recommender systems ability to recommend that item that was left out in the top-n results we creates for that user in the testing phase. So we measure our ability to recommend an item in the top-n list for each user that was left out from training data.

That’s why it is called leave-one-out.

The trouble here is its get lot harder to get one specific movie right while testing than the just get one of the n recommendations.

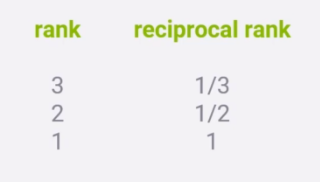
So hit rate with leave-one-out is small and difficult to measure unless we have very large data set to work with. This is much more user focus metric when we know our recommendation system will be producing top-n list in the real world.

**3. Average Reciprocal Hit Rate (ARHR)**



It counts where in the top-n list the hit appears so we end up getting more credit for successfully recommending an item in a top slot than in the bottom slot. Again it is also more user focus metric, since user tends to focus in the beginning of the list.

Here we sum up reciprocal rank of each hit.



**4. Cumulative hit rate (cHR)**



Here we throw away hits of our predicted rating is below some threshold. The idea is that we should not get credit for recommending items to a user that we think they won’t actually enjoy. So in above example if we have a cut of for 3 stars then we throw away the hits for second and fourth items in the test result and our hit rate metric wouldn’t count them at all.



**4. Rating hit rate (rHR)**



Here we look at hit rate is to break it down by predicted rating score. It will be good way to get an idea of the distribution of how good our algorithm thinks recommended movies are that actually get a hit. Ideally we want to recommend movies that they are actually liked and breaking down the distribution to some sense of how well we are doing in more detail.

**RMSE and hit rate are always related.**

**===========================================================================**

**Other important measures:**

**1. Coverage**

It is the percentage of possible recommendations that our recommender system is able to provide.

Consider our movie-lens dataset in which many of the movies don’t have ratings provided.

Then the coverage of this recommender system would be low.

So coverage can be odds with accuracy, if we enforce a higher quality threshold on the recommendations we make then we might improve our accuracy at the expense of coverage. Finding the balance of where exactly we better off recommending nothing at all would be challenging.

So we also look at coverage for what percentage of recommendations from large data set with some minimum threshold.

**2. Diversity**

(1 – S)

S = Average similarity between recommendation pairs.

It is a measure of how broad variety of items our recommender system is putting in front of people. We can use similarity score to measure diversity.

It is opposite of average similarity (1-S). So if we have high diversity score then it means bad recommendation. The diversity score should be low to see good recommendations.

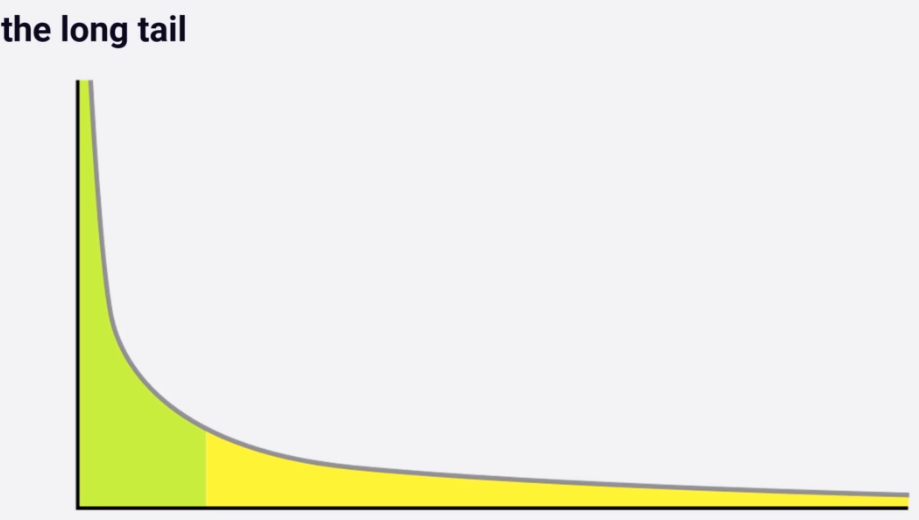
**3. Novelty**

It is the measure of how popular the items are we recommending. Just recommending random stuff will result into high novelty score since majority of items are not top sellers.

There is a concept of user trust in the recommendation system. People want to see few familiar items in the recommendation so that they can say yes it is a good recommendation.

If we only recommend things which people never heard of then they may conclude that our recommender system is not good and they may ignore our recommendations.

(It is the mean popularity rank of recommended items) means how popular the items are you recommending.



**Churn**

How often do recommendations change?

**Responsiveness**

How quickly does new user behaviour influence our recommendations?

If we rate a movie does it affect our recommendation immediately or after some time?

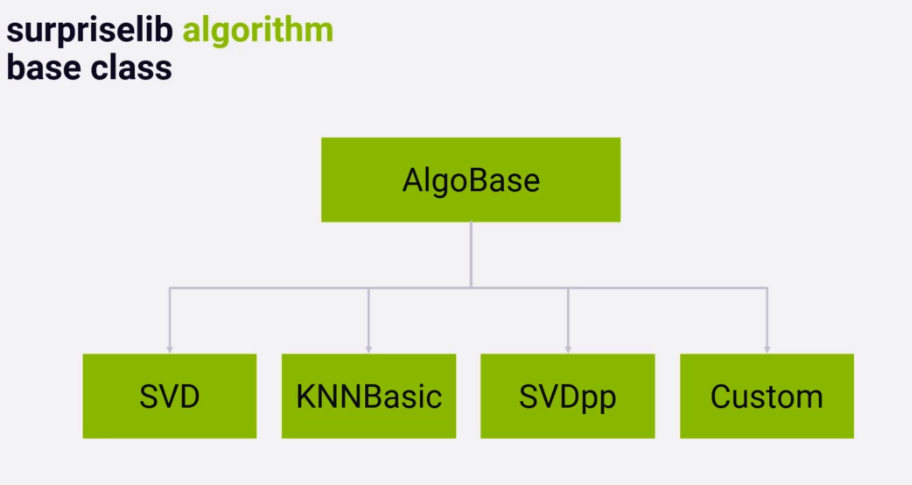
**==========================================================================**

**Building a Recommender Engine Framework**

We use below open source library for building and analysing recommender system.

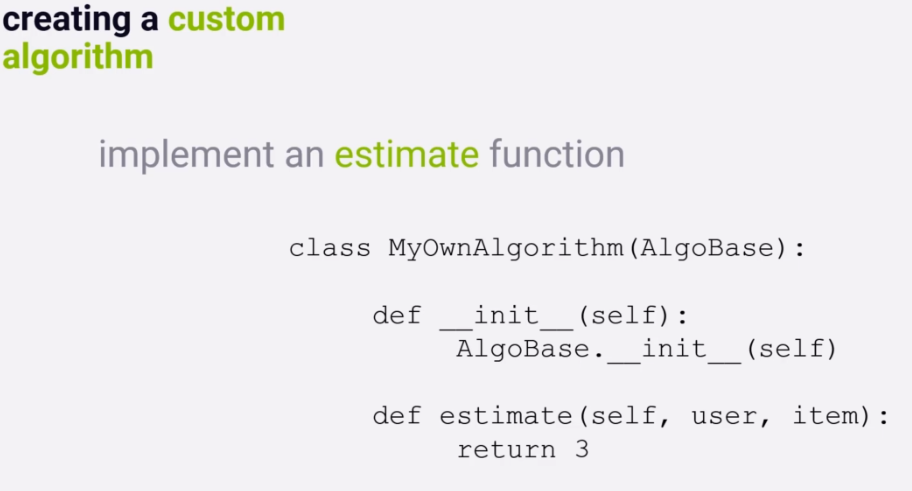
**Surprise**

**A python scikit for building and analysing recommender systems.**

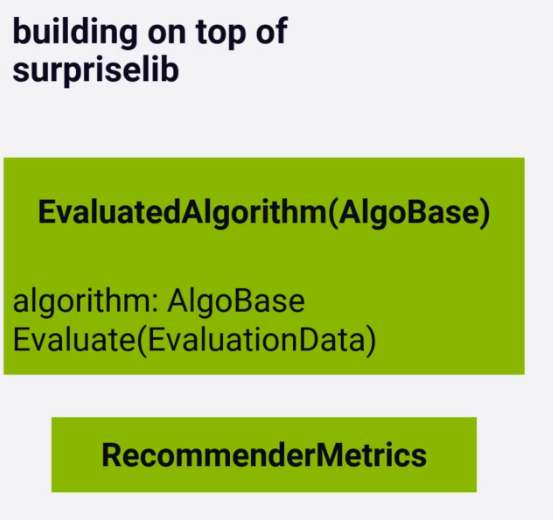


If we want to create our own algorithm like surprise then we can follow below approach.

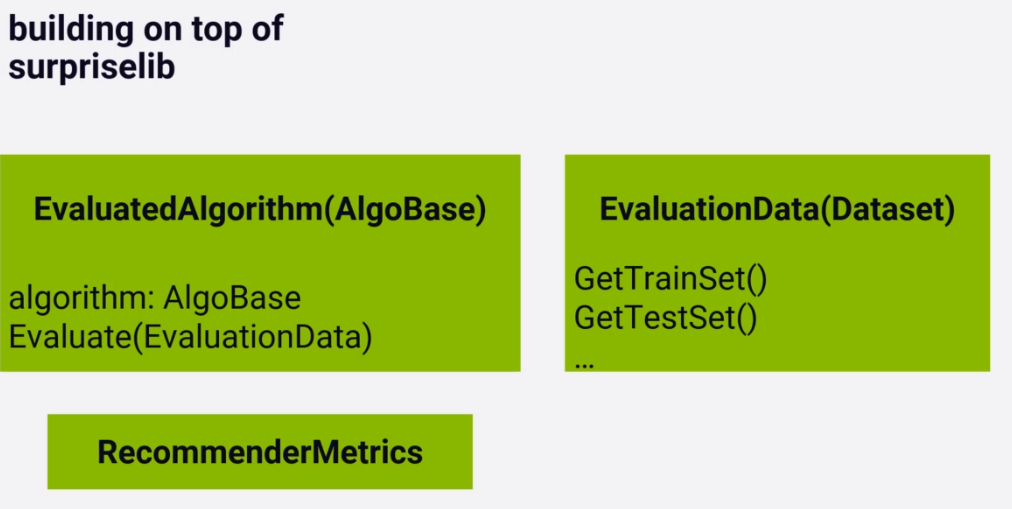
This custom algorithms predicting rating with 3 stars.



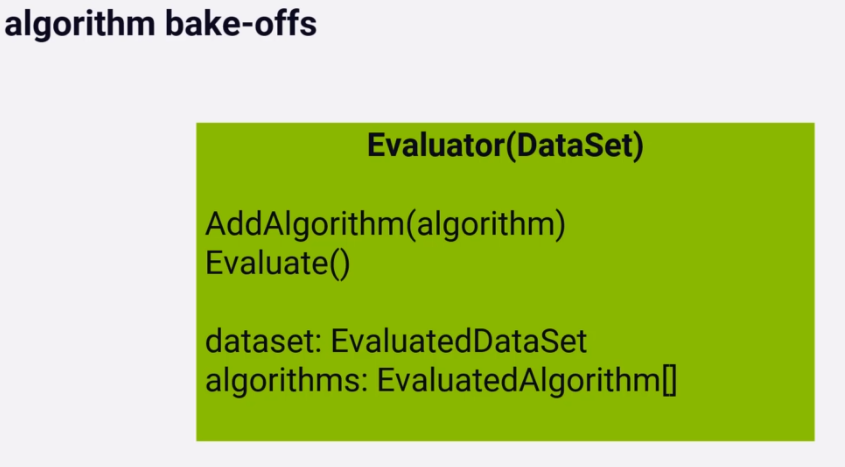
We would like to do more than just predicting ratings. So we will develop a new class **EvaluatedAlgorithm** with a function **evaluate** on top of surprise library that runs all metrics in the **RecommenderMetrics.py** on that algorithm.



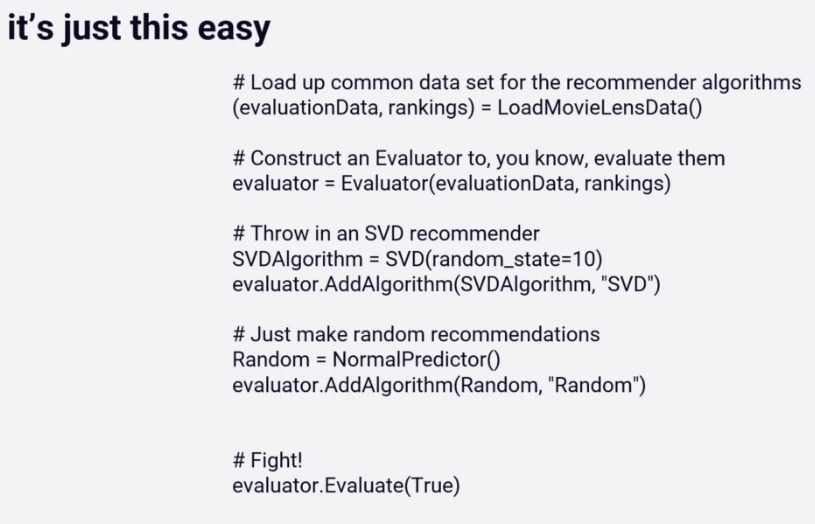
Split data set into train and test set with new class **EvaluationData**.



Now to compare algorithms with each other for their predictions we have one more class **Evaluator**.



It is easy to compare algorithms in below way:



===========================================================================

Now we see some actual recommendation algorithms.