Crab: A Recommendation Engine Framework for Python

Marcel Caraciolo‡*, Bruno Melo‡, Ricardo Caspirro‡

Abstract—Crab is a flexible, fast recommender engine for Python that integrates classic information filtering recommendation algorithms in the world of scientific Python packages (NumPy,SciPy, Matplotlib). The engine aims to provide a rich set of components from which you can construct a customized recommender system from a set of algorithms. It is designed for scability, flexibility and performance making use of scientific optimized python packages in order to provide simple and efficient solutions that are acessible to everybody and reusable in various contexts: science and engineering. The engine takes users' preferences for items and returns estimated preferences for other items. For instance, a web site that sells movies could easily use Crab to figure out, from past purchase data, which movies a customer might be interested in watching to. This work presents our inniative in developing this framework in Python following the standards of the well-known machine learning toolkit Scikit-Learn to be an alternative solution for Mahout Taste collaborative framework for Java. Finally, we discuss its main features, real scenarios where this framework is already applied and future extensions.

Index Terms—data mining, machine learning, recommendation systems, information filtering, framework, web

Introduction

With the great advancements of machine learning in the past few years, many new learning algorithms have been proposed and one of the most recognizable techniques in use today are the recommender engines [Adoma2005]. There are several services or sites that attempt to recommend books or movies or articles based on users past actions [Linden2003] , [Abhinandan2007] . By trying to infer tastes and preferences, those systems focus to identify unknown items that are are of interest given an user. Although people's tastes vary, they do follow patterns. People tend to like things that are similar to other items they like. For instance, because a person loves bacon-lettuce-and-tomato sandwiches, the recommender system could guess that he would enjoy a club sandwich, which is mostly the same sandwich, with turkey. Likewise, people tend to like things that similar people like. When a friend entered design school, he saw that just about every other design student owned a Macintosh computer - which was no surprise, as she already a lifetime Mac User. Recommendation is all about predicting these patterns of taste, and using them to discover new and desirable things a person didn't know about.

* *Corresponding author: marcel@muricoca.com*
‡ *Muricoca Labs*

Copyright © 2011 Marcel Caraciolo et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Recommendation engines have been implemented in programming languages such as C/C++, Java, among others and made publicly available. One of the most popular implementations is the open-source recommendation library Taste, which was included in the Mahout framework project in 2008 [Taste] . Mahout is a well-known machine learning toolkit written in Java for building scalable machine libraries [Mahout] . It is specially a great resource for developers who are willing to take a step into recommendation and machine learning technologies. Taste has enabled systematic comparisons between standard developed recommendation methods, leading to an increased visibility, and supporting their broad adoption in the community of machine learning, information filtering and industry. There are also several another publicly available implementations of recommender engines toolkits in the web [EasyRec] , [MyMediaLite]. Each one comes with its own interface, sometimes even not updated anymore by the project owners, a small set of recommendation techniques implemented, and unique benefits and drawbacks.

For Python developers, which has a considerable amount of machine learning developers and researchers, there is no single unified way of interfacing and testing these recommendation algorithms, even though there are some approaches but found incomplete or missing the required set for creating and evaluating new methods [PySuggest], [djangorecommender]. This restraints developers and researchers from fully taking advantage of the recent developments in recommendation engines algorithms as also an obstacle for machine learning researchers that will not want to learn complex programming languages for writing their recommender approaches. Python has been considered for many years a excellent choice for programming beginners since it is easy to learn with simple syntax, portable and extensive. In scientific computing field, high-quality extensive libraries such as Scipy, MatPlotlib and Numpy have given Python an attractive alternative for researchers in academy and industry to write machine learning implementations and toolkits such as Brain, Shogun, Scikit-Learn, Milk and many others.

The reason of not having an alternative for python machine learning developers by providing an unified and easy-to-use recommendation framework motivated us to develop a recommendation engine toolbox that provides a rich set of features from which the developer can build a customized recommender system. The result is a framework, called Crab, with focus on large-scale recommendations making use of scientific python packages such as Scipy, Numpy and Matplotlib to provide simple and efficient solutions for constructing recommender systems that are acessible and reusable in various contexts. Crab provides a generic

interface for recommender systems implementations, among them the collaborative filtering approaches such as User-Based and Item-Based filtering, which are already available for use. The recommender interfaces can be easily combined with more than 10 different pairwise metrics already implemented, like the cosine, tanimoto, pearson, euclidean using Scipy and Numpy basic optimized functions [Breese1998]. Moreover, it offers support for using similarities functions such as user-to-user or item-to-item and allows easy integration with different input domains like databases, text files or python dictionaries.

Currently, the collaborative filtering algorithms are widely supported. In addition to the User-Based and Item-Based filtering techniques, Crab implements several pairwise metrics and provides the basic interfaces for developers to build their own customized recommender algorithms. Finally, several widely used performance measures, such as accuracy, precision, recall are implemented in Crab.

An important aspect in the design of Crab was to enable very large-scale recommendations. Crab is currently being rewritten to support optimized scientific computations by using Scipy and Numpy routines. Another feature concerned by the current maintainers is to make Crab support sparse and large datasets in a way that there is a little as possible overhead for storing the data and intermediate results. Moreover, Crab also aims to support scaling in recommender systems in order to build high-scale, dynamic and fast recommendations over simple calls. It is also planned to support distributed recommendation computation by interfacing with the distributed computation library MrJob written in Python currently developed by Yelp [MrJob]. What sets Crab apart from many other recommender systems toolboxes, is that it provides interactive interfaces to build, deploy and evaluate customized recommender algorithms written in Python running on several platforms such as Linux, BSD, MacOS and Windows.

The outline of this paper is as follows. We first discuss the Crab's main features by explaining the architecture of the framework. Next, we provide our current approach for representing the data in our system and current challenges. Then, we also presents how Crab can be used in production by showing a real scenario where it is already deployed. Finally, we discuss about our plans to handle with distributed recommendation computations. Also, our conclusions and future works are also presented at the end of this paper.

Recommender Engines

Crab contains a recommender engine, in fact, several types beginning with conventional in the literature user-based and item-based recommenders. It provides an assortment of components that may be plugged together and customized to create an ideal recommender for a particular domain. The toolkit is implemented using Python and the scientific environments for numerical applications such as Scipy and NumPy. The decision of choosing those libraries is because they are widely used in scientific computations specially in python programs. Another reason is because the framework uses the Scikit-learn toolkit as dependant, which provides basic components from our recommender interfaces derive [Scikitlearn] . The Figure 1 presents the relationship between these basic components. Not all Crab-based recommenders will look like this -- some will employ different components with different relationships, but this gives a sense of the role of each component.

The Data Model implementation stores and provides access to all the preferences, user and item data needed in the recommendation. The Similarity interface provides the notion of how similar two users or items are; where this could be based on one of many possible pairwise metrics or calculations. Finally, a Recommender interface which inherits the BaseEstimator from Scikit-learn pull all these components together to recommend items to users, and related functionality.

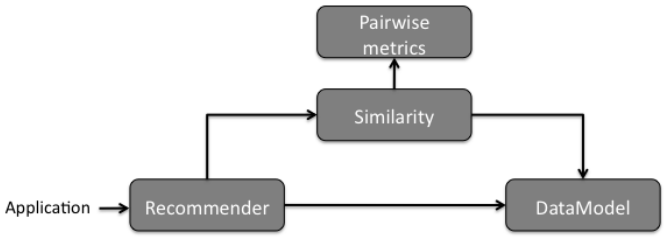

Fig. 1: *Simplified illustration of the component interaction in Crab*

It is easy to explore recommendations with Crab. Let's go through a trivial example. First, we need input to the recommender, data on which to base recommendations. Generally, this data takes the form of preferences which are associations from users to items, where these users and items could be anything. A preference consist of a user ID and an item ID, and usually a number expressing the strength of the user's preference for the item. IDs in Crab can be represented by any type indexable such as string, integers, etc. The preference value could be anything, as long as larger values mean strong positive preferences. For instance, these values can be considered as ratings on a scale of 1 to 5, where the user has assigned "1" to items he can't stand, and "5" to his favorites.

Crab is able to work with text files containing information about users and their preferences. The current state of the framework allows developers to connect with databases via Django's ORM or text files containing the user IDs, product IDs and preferences. For instance, we will consider a simple dataset including data about users, cleverly named "1" to "5" and their preferences for four movies, which we call "101" through "104". By loading this dataset and passing as parameter to the dataset loader, all the inputs will be loaded in memory by creating a Data Model object.

Analyzing the data set shown at Figure 2, it is possible to notice that Users 1 and 5 seem to have similar tastes. Users 1 and 3 don't overlap much since the only movie they both express a preference for is 101. On other hand, users 1 and 2 tastes are opposite- 1 likes 101 while 2 doesn't, and 1 likes 103 while 2 is just the opposite. By using one of recommender algorithms available in Crab such as the User-Based-Filtering with the given data set loaded in a Data Model as input, just run this script using your favorite IDE as you can see the snippet code below.

```
from models.basic_models import FileDataModel
from recommenders.basic_recommenders
   import UserBasedRecommender
from similarities.basic_similarities
   import UserSimilarity
from neighborhoods.basic_neighborhoods
   import NearestUserNeighborhood
from metrics.pairwise import pearson_correlation

user_id = 1
# load the dataset
model = FileDataModel('simple_dataset.csv')
similarity = UserSimilarity(model,
                            pearson_correlation)
neighbor = NearestUserNeighborhood(similarity,
                                   model, 4, 0.0)
```

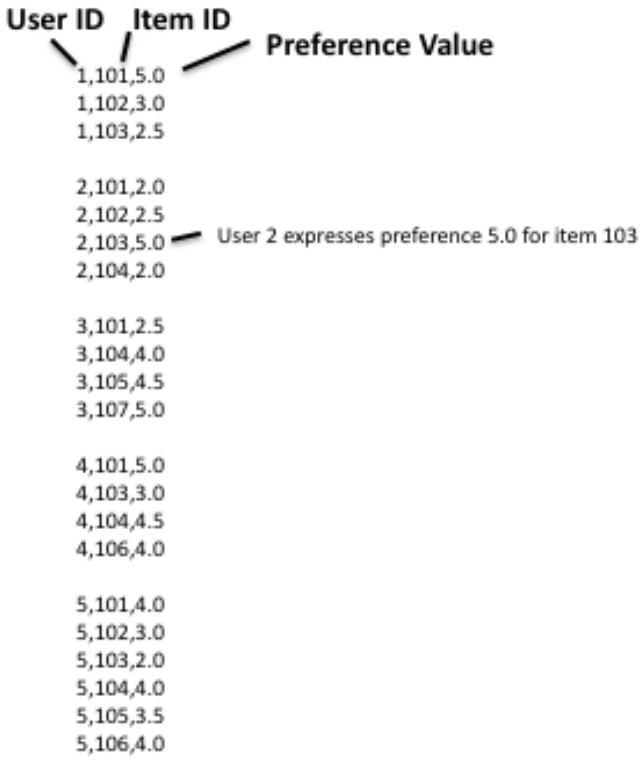

Fig. 2: *Book ratings data set - intro.csv*

```
# create the recommender engine
recommender = UserBasedRecommender(model, similarity,
                                   neighbor, False)
# recommend 1 item to user 1
print recommender.recommend(user_id, 1)
```

The output of running program should be: 104. We asked for one top recommendation, and got one. The recommender engine recommended the book 104 to user 1. This happens because it estimated user 1's preference for book 104 to be about 4.3 and that was the highest among all the items eligible for recommendations. It is important to notice that all recommenders are estimators, so they estimate how much users may like certain items. The recommender worked well considering a small data set. Analyzing the data you can see that the recommender picked the movie 104 over all items, since 104 is a bit more highly rated overall. This can be refforced since user 1 has similar preferences to the users 4 and 5, where both have highly rated.

For small data sets, producing recommendations appears trivial as showed above. However, for data sets that are huge and noisy, it is a different situation. For instance, consider a popular news site recommending new articles to readers. Preferences are inferred from article clicks. But, many of these "preferences" may be noisy - maybe a reader clicked an article but did not like it, or, had clicked the wrong story. Imagine also the size of the data set - perhaps billions of clicks in a month. It is necessary for recommender engines to handle with real-life data sets, and Crab as Mahout is focusing on how to deal with large and sparse data as we will discuss in a future section.

Therefore, before deploying recommender engines in Crab into production, it is necessary to present another main concept in our framework at the next section: representation of data.

Representing Data

Recommender systems are data-intensive and runtime performance is greatly affected by quantity of data and its representation. In Crab the recommender-related data is encapsulated in the implementations of DataModel. DataModel provides efficient access to data required by several recommender algorithms. For instance, a DataModel can provide a count or an array of all user IDs in the input data, or provide access to all preferences associated to an item.

One of the implementations available in Crab is the in-memory implementation DictDataModels. This model is appropriate if the developer wants to construct his data representation in memory by passing a dictionary of user IDs and their preferences for item IDs. One of benefits of this model is that it can easily work with JSON files, which is commonly used as output at web services and REST APIs, since Python converts the json input into a bult-in dictionary.

```
from models.basic_models
   import DictPreferenceDataModel

dataset = {'1':{'101': 3.0, '102': 3.5},
           '2':{'102': 4.0, '103':2.5, '104': 3.5}}

#load the dataset
model = DictPreferenceDataModel(dataset)
print model.user_ids()
#numpy.array(['1','2'])

print model.preference_value('1', '102')
#3.5

print model.preferences_for_item('102')
#numpy.array([('1',3.5),('2',4.0)])
```

Typically the model that developers will use is the FileDataModel - which reads data from a file and stores the resulting preference data in memory, in a DictDataModel. Comma-separated-value or tab-separated files which each line contains one datum: user ID, item ID and preference value are acceptable as input to the model. Zipped and gzipped files will be supported, since they are commonly used for store huge data in a compressed format.

For data sets which ignore the preference values, that is, ignore the strength of preference, Crab also has an appropriate DataModel twin of DictDataModel called BooleanDictDataModel. This is likewise as in-memory DictDataModel implementation, but one which internally does not store the preference values. These preferences also called "boolean preferences" have two states: exists, or does not exist and happens when preferences values aren't available to begin with. For instance, imagine a news site recommending articles to user based on previously viewed article. It is not typical for users to rate articles. So the recommender recommends articles based on previously viewed articles, which establishes some association between user and item, an interesting scenario for using the BooleanDictModel.

```
from models.basic_models
   import DictBooleanDataModel

dataset = {'1':['101','102'],
           '2':['102','103','104']}

#load the dataset
model = DictBooleanDataModel(dataset)

print model.user_ids()
#numpy.array(['1','2'])

print model.preference_value('1', '102')
#1.0 - all preferences are valued with 1.0
```

```
print model.preferences_for_item('102')
#numpy.array([('1',1.0),('2',1.0)])
```

Crab also supports store and access preference data from a relational database. The developer can easily implement their recommender by using customized DataModels integrated with several databases. One example is the MongoDB, a NoSQL database commonly used for non-structured data [MongoDB]. By using MongoEngine, a ORM adapter for integrating MongoDB with Django, we could easily set up a customized Data Model to access and retrieve data from MongoDB databases easily [Django], [MongoEngine]. In fact, it is already in production at a recommender engine using Crab for a brazilian social network called AtéPassar. We will explore more about it in the next sections.

One of the current challenges that we are facing is how to handle with all this data in-memory. Specially for recommender algorithms, which are data intensive. We are currently investigating how to store data in memory and work with databases directly without using in-memory data representations. We are concerned that it is necessary for Crab to handle with huge data sets and keep all this data in memory can affects the performance of the recommender engines implemented using our framework. Crab uses Numpy arrays for storing the matrices and in the organization of this paper at the time we were discussing about using scipy.sparse packages, a Scipy 2-D sparse matrix package implemented for handling with sparse a matrices in a efficient way.

Now that we have discussed about how Crab represents the data input to recommender, the next section will examine the recommenders implemented in detail as also how to evaluate recommenders using Crab tools.

Making Recommendations

Crab already supports the collaborative recommender user-based and item-based approaches. They are considered in some of the earliest research in the field. The user-based recommender algorithm can be described as a process of recommending items to some user, denoted by u, as follows:

```
for every item i that u has no preference for yet

  for every other user v that has preference for i

     compute a similarity s between u and v

     incorporate v's preference for i, weighted by s,
        into a running average

return the top items, ranked by weighted average
```

The outer loop suggests we should consider every known item that the user hasn't already expressed a preference for as a candidate for recommendation. The inner loop suggests that we should look to any other user who has expressed a preference for this candidate item and see what his or her preference value for it was. In the end, those values are averaged to come up with an estimate -- a weighted average. Each preference value is weighted in the average by how similar that user is to the target user. The more similar a user, the more heavily that we weight his or her preference value. In the standard user-based recommendation algorithm, in the step of searching for every known item in the data set, instead, a "neighborhood" of most similar users is computed first, and only items known to those users are considered.

In the first section we have already presented a user-based recommender in action. Let's go back to it in order to explore the components the approach uses.

```
# do the basic imports
user_id = 1

# load the dataset
model = FileDataModel('simple_dataset.csv')

# define the similarity used and the pairwise metric
similarity = UserSimilarity(model,
                            pearson_correlation)

# for neighborhood we will use the k-NN approach
neighbor = NearestUserNeighborhood(similarity,
                                   model, 4, 0.0)

# now add all to the UserBasedRecommender
recommender = UserBasedRecommender(model, similarity,
                                   neighbor, False)

#recommend 2 items to user 1
print recommender.recommend(user_id,2)
```

UserSimilarity encapsulates the concept of similarity amongst users. The UserNeighborhood encapsulates the notion of a group of most-similar users. The UserNeighborhood uses a UserSimilarity, which extends the basic interface BaseSimilarity. However, the developers are encouraged to plug in new variations of similarity - just creating new BaseSimilarity implementations - and get quite different results. As you will see, Crab is not one recommender engine at all, but a set of components that may be plugged together in order to create customized recommender systems for a particular domain. Here we sum up the components used in the user-based approach:

- Data model implemented via DataModel
- User-to-User similarity metric implemented via UserSimilarity
- User neighborhood definition implementd via UserNeighborhood
- Recommender engine implemented via Recommender, in this case, UserBasedRecommender

The same approach can be used at UserNeighborhood where developers also can create their customized neighborhood approaches for defining the set of most similar users. Another important part of recommenders to examine is the pairwise metrics implementation. In the case of the User-based recommender, it relies most of all in this component. Crab implements several pairwise metrics using the Numpy and Scipy scientific libraries such as Pearson Correlation, Euclidean distance, Cosine measure and distance implementations that ignore preferences entirely like as Tanimoto coefficient and Log-likehood.

Another approach to recommendation implemented in Crab is the item-based recommender. Item-based recommendation is derived from how similar items are to items, instead of users to users. The algorithm implemented is familiar to the user-based recommender:

```
for every item i that u has no preference for yet

   for every item j that u has a preference for

       compute a similarity s between i and j

       add u's preference for j, weighted by s,
        to a running average
```

```
return the top items, ranked by weighted average
```

In this algorithm it is evaluated the item-item similarity, not user-user similarity as shown at the user-based approach. Although they look similar, there are different properties. For instance, the running time of an item-based recommender scales up as the number of items increases, whereas a user-based recommender's running time goes up as the number of users increases. The performance advantage in item-based approach is significant compared to the user-based one. Let's see how to use item-based recommender in Crab with the following code.

```
# do the basic imports
user_id = 1

# load the dataset
model = FileDataModel('simple_dataset.csv')

# define the Similarity used and the pairwise metric
similarity = ItemSimilarity(model, euclidean_distance)

# there is no neighborhood in this approach
# now add all to the ItemBasedRecommender
recommender = ItemBasedRecommender(model,
                                   similarity, False)

# recommend 2 items to user 1
print recommender.recommend(user_id,2)
```

Here it employs ItemBasedRecommender rather than UserBasedRecommender, and it requires a simpler set of dependencies. It also implements the ItemSimilarity interface, which is similar to the UserSimilarity that we've already seen. The ItemSimilarity also works with the pairwise metrics used in the UserSimilarity. There is no item neighborhood, since it compares series of preferences expressed by many users for one item instead of by one user for many items.

Now that we have seen some techniques implemented at Crab, which produces recommendations for a user, it is now time to answer another question, "what are the best recommendations for a user ?". A recommender engine is a tool and predicts user preferences for items that he haven't expressed any preference for. The best possible recommender is a tool that could somehow know, before you do, exactly estimate how much you would like every possible item available. The remainder of this section will explore evaluation of a recommender, an important step in the construction of a recommender system, which focus on the evaluating the quality of the its estimated preference values - that is, evaluating how closely the estimated preferences match the actual preferences.

Crab supports several metrics widely used in the recommendation literature such as the RMSE (root-mean-square-error), precision, recall and F1-Score. Let's see the previous example code and instead evaluate the simple recommender we created, on our data set:

```
from evaluators.statistics
    import RMSRecommenderEvaluator

# initialize the recommender
# initialize the RMSE Evaluator
evaluator = RMRecommenderEvaluator()

# using training set with 70% of data and 30% for test
print evaluator.evaluate(recommender,
                         model, 0.7, 1.0)
#0.75
```

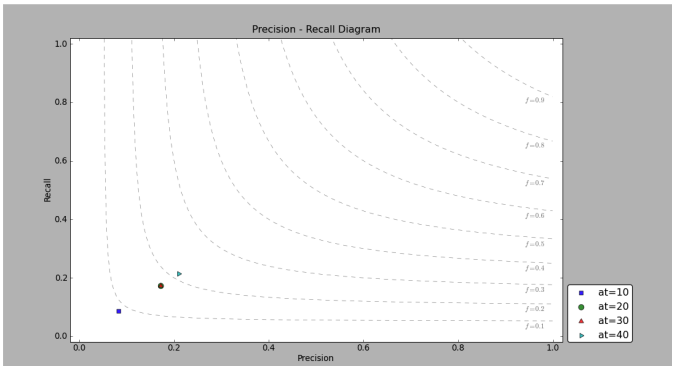

Fig. 3: *PrecisionxRecall Graph with F1-Score.*

Most of the action happens in evaluate(). The RecommenderEvaluator handles splitting the data into a training and test set, builds a new training DataModel and Recommender to test, and compares its estimated preferences to the actual test data. See that we pass the Recommender to this method. This is because the evaluator will need to build a Recommender around a newly created training DataModel. This simple code prints the result of the evaluation: a score indicating how well the Recommender performed. The evaluator is an abstract class, so the developers may build their custom evaluators, just extending the base evaluator.

For precision, recall and F1-Score Crab provides also a simple way to compute these values for a Recommender:

```
from evaluators.statistics
    import IRStatsRecommenderEvaluator

# initialize the recommender
# initialize the IR Evaluator
evaluator = IRStatsRecommenderEvaluator()

# call evaluate considering the top 4 items recommended.
print evaluator.evaluate(recommender, model, 2, 1.0)
# {'precision': 0.75, 'recall': 1.0,
    'f1Score': 0.6777}
```

The result you see would vary significantly due to random selection of training data and test data. Remember that precision is the proportion of top recommendations that are good recommendations, recall is the proportion of good recommendations that appear in top recommendations and F1-Score is a score that analyzes the proportion against precision and recall. So Precision at 2 with 0.75 means on average about a three quarters of recommendations were good. Recall at 2 with 1.0; all good recommendations are among those recommendations. In the following graph at Figure 3, it presents the PrecisionxRecall with F1-Scores evaluated. A brief analysis shows that more training set size grows, more the accuracy score grows. It is important to notice that the evaluator doe not measure if the algorithm is better or faster. It is necessary to make a comparison between the algorithms to check the accuracy specially on other data sets available.

Crab supports several tools for testing and evaluating recommenders in a painless way. One of the future releases will support the plot of charts to help the developers to better analyze and visualized their recommender behavior.

Taking Recommenders to Production

So far we have presented the recommender algorithms and variants that Crab provides. We also presented how Crab handles with

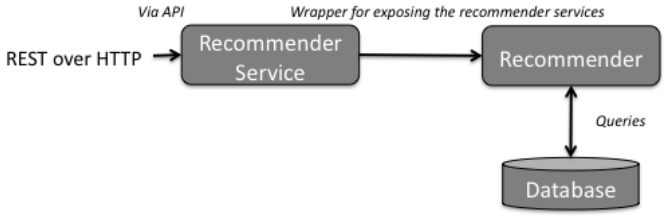

Fig. 4: *Crab Web Services server-side interaction over HTTP*

accuracy evaluation of a recommender. But another important step for a recommender life-cycle is to turn it into a deployable production-ready web service.

We are extending Crab in order to allow developers to deploy a recommender as a stand-alone component of your application architecture, rather than embed it inside your application. One common approach is to expose the recommendations over the web via simple HTTP or web services protocols such as SOAP or REST. One advantage using this service is that the recommender is deployed as a web- accessible service as independent component in a web container or a standalone process. In the other hand, this adds complexity, but it allows other applications written in other languages or running at remote machines to access the service. We are considering use framework web Django with the the Django-Piston RESTful builder to expose the recommendations via a simple API using REST over HTTP [DjangoPiston]. Our current structure is illustrated in Figure 4, which wraps the recommender implementation using the django models and piston handlers to provide the external access.

There is a recommender engine powered by Crab in production using REST APIs to access the the recommendations. The recommender engine uses collaborative filtering algorithms to recommend users, study groups and videos in a brazilian educational social network called AtéPassar [AtePassar] . Besides the suggestions, the recommender was also extended to provide the explanations for each recommendation, in a way that the user not only receives the recommendation but also why the given recommendation was proposed to him. The recommender is in production since January 2011 and suggested almost 60.000 items for more than 50.000 users registered at the network. The following Figure 5 shows the web interface with the recommender engine in action at AtéPassar. One contribution of this work was a new Data Model for integrating with MongoDB database for retrieving and storing the recommendations and it is being rewritten for the new release of Crab supporting Numpy and Scipy libraries.

Crab can comfortably digest medium and small data sets on one machine and produce recommendations in real time. But it still lacks a mechanism that handles a much larger data set. One common approach is distribute the recommendation computations, which will be detailed in the next section.

Distributing Recommendation Computations

For large data sets with millions of preferences, the current approaches for single machines would have trouble processing recommendations in the way we have seen in the last sections. It is necessary to deploy a new type of recommender algorithms using a distributed and parallelized computing approach. One of the most popular paradigms is the MapReduce and Hadoop [Hadoop].

Fig. 5: *AtéPassar recommendation engine powered by Crab Framework*

Crab didn't support at the time of writing this paper distributed computing, but we are planning to develop variations on the item-based recommender approach in order to run it in the distributed world. One of our plans is to use the Yelp framework mrJob which supports Hadoop and it is written in Python, so we may easily integrate it with our framework. One of the main concerns in this topic is to give Crab a scalable and efficient recommender implementation without having high memory and resources consumption as the number of items grows.

Another concern is to investigate and develop other distributed implementations such as Slope One, Matrix Factorization, giving the developer alternatives for choosing the best solution for its need specially when handling with large data sets using the power of Hadoop's MapReduce computations. Another important optimization is to use the JIT compiler PyPy for Python which is being development and will bring faster computations on NumPy [NumpyFollowUp].

Conclusion and Future Works

In this paper we have presented our efforts in building a recommender engine toolkit in Python, which we believe that may be useful and make an increasing impact beyond the recommendation systems community by benefiting diverse applications. We are confident that Crab will be an interesting alternative for machine learning researchers and developers to create, test and deploy their recommendation algorithms writing a few lines of code with the simplicity and flexibility that Python with the scientific libraries Numpy and Scipy offers. The project uses as dependency the Scikit-learn toolkit, which forces the Crab framework to cope with high standards of coding and testing, turning it into a madure

and efficient machine learning toolkit. Discussing the technical aspects, we are also always improving the framework by planning to develop new recommender algorithms such as Matrix Factorization, SVD and Boltzmann machines. Another concern is to bring to the framework not only collaborative filtering algorithms but also content based filtering (content analysis), social relevance proximity graphs (social/trust networks) and hybrid approaches. Finally it is also a requirement to a recommender engine to be scalable, that is, to handle with large and sparse data sets. We are planning to develop a scalable recommendation implementation by using Yelp framework mrJob which supports Hadoop and MapReduce as explained in the previous section.

Our project is hosted at Github repository and it is open for machine learning community to use, test and help this project to grow up. Future releases are planned which will include more projects building on it and a evaluation tool with several plots and graphs to help the machine learning developer better understand the behavior of his recommender algorithm. It is an alternative for Python developers to the Mahout machine learning project written in Java. The source code is freely available under the BSD license at http://github.com/muricoca/crab.

REFERENCES

[Adoma2005] Adomavicius, G.; Tuzhilin, A. *Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions*, IEEE Transactions on Knowledge and Data Engineering; 17(6):734–749, June 2005.

[Linden2003] Greg Linden, Brent Smith, and Jeremy York. *Amazon.com Recommendations: Item-to-Item Collaborative Filtering.*, IEEE Internet Computing 7, 1, 76-80, January 2003.

[Abhinandan2007] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram, *Google news personalization: scalable online collaborative filtering.*, In Proceedings of the 16th international conference on World Wide Web (WWW '07). ACM, New York, NY, USA, 271-280, 2007.

[Taste] *Taste - Collaborative Filtering For Java* , accessible at: http://taste.sourceforge.net/.

[Mahout] *Mahout - Apache Machine Learning Toolkit* ,accessible at: http://mahout.apache.org/

[EasyRec] *EasyRec* ,accessible at: http://www.easyrec.org/

[MyMediaLite] *MyMediaLite Recommender System Library*, accessible at: http://www.ismll.uni-hildesheim.de/mymedialite/

[PySuggest] *PySuggest*, accessible at: http://code.google.com/p/pysuggest/

[djangorecommender] *Django-recommender* accessible at: http://code.google.com/p/django-recommender/

[Breese1998] J. S. Breese, D. Heckerman, and C. Kadie. *Empirical analysis of predictive algorithms for collaborative filtering.*, UAI, Madison, WI, USA, pp. 43-52, 1998.

[MrJob] *mrjob*, accessible at: https://github.com/Yelp/mrjob

[Scikitlearn] *Scikit-learn*, accessible at: http://scikit-learn.sourceforge.net/

[MongoDB] *MongoDB*, accessible at: https://www.mongodb.org/

[Django] *Django*, accessible at: https://www.djangoproject.com/

[MongoEngine] *MongoEngine*, accessible at: https://www.mongoengine.org/

[DjangoPiston] *Django-Piston*, accessible at: https://bitbucket.org/jespern/django-piston/wiki/Home

[AtePassar] *AtéPassar*, accessible at: http://atepassar.com

[Hadoop] *Hadoop*, accessible at: http://hadoop.apache.org/

[NumpyFollowUp] *Numpy Follow up*, accessible at: http://morepypy.blogspot.com/2011/05/numpy-follow-up.html