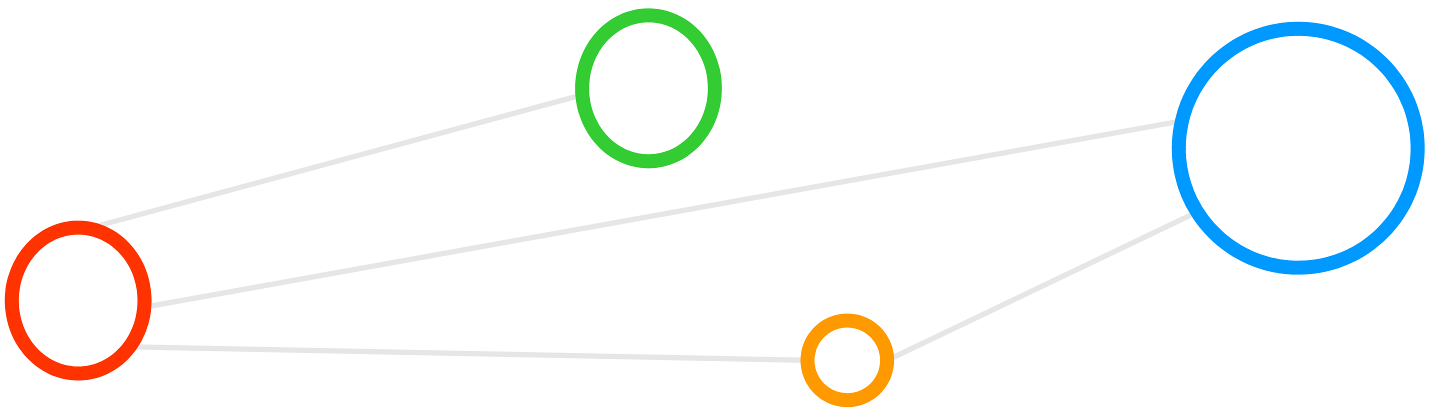
ON4OFF – Overview of Activities

Prof. Dr. – Ing. Morris Riedel et al.
Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany
School of Engineering & Natural Sciences, University of Iceland
19.04.2021, Online

Executive – Summary & Overview



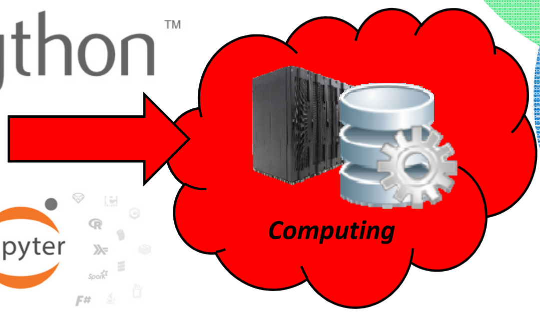
Machine Learning, Data Mining & Statistics overlap to enable Data Science

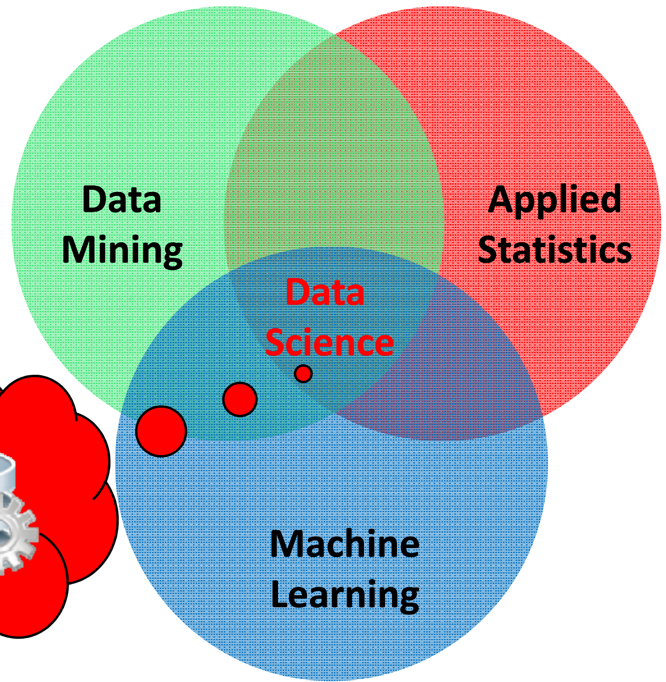
1. Some pattern exists
2. No exact mathematical formula
3. Data exists

- Idea 'Learning from Big Data'
  - Shared with a wide variety of other disciplines
  - E.g. signal processing, big data mining, etc.
- Challenges
  - Data is often complex
  - Requires 'Big Data analytics'
  - Learning from data requires processing time → Clouds or High Performance Compuing

[25] Python

[24] Jupyter





- Machine learning is a very broad subject and goes from very abstract theory to extreme practice ('rules of thumb')
- Training machine learning models needs processing time (clouds or high performance computing)
- While data analysis is more describing the process of analysin the data, the term data analytics also includes and the necessary scalable or parallel infrastructure to perform analysis of 'big data'

ON4OFF Review – Executive Summary – Machine Learning in ON4OFF

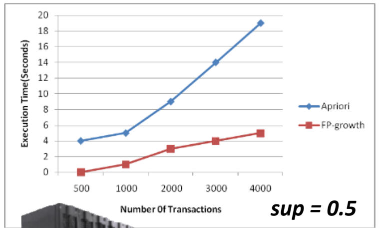
▪ Part I – Association Rule Mining

ID	Items
1	{Bread, Milk}
2	{Bread, Diapers, Beer, Eggs}
3	{Milk, Diapers, Beer, Cola}
4	{Bread, Milk, Diapers, Beer}
5	{Bread, Milk, Diapers, Cola}
…	…

market basket transactions

{Diapers, Beer} Example of a frequent itemset

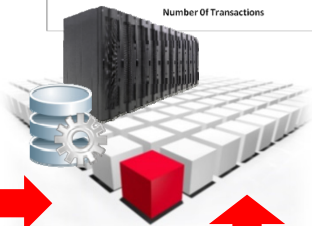
{Diapers} → {Beer} Example of an association rule



sup = 0.5

TID	Items
1	{Bread,Milk}
2	{Bread,Diapers,Beer,Eggs}
3	{Milk,Diapers,Beer,Cola}
4	{Bread,Milk,Diapers,Beer}
5	{Bread,Milk,Diapers,Cola}

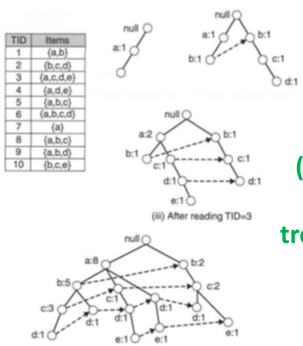
(APRIORI uses frequent itemsets & iterative approach)



Item	Count
Beer	3
Bread	4
Cola	2
Diapers	4
Milk	4
Eggs	1

2-Itemset	Count
{Beer,Bread}	2
{Beer,Diapers}	3
{Beer,Milk}	2
{Bread,Diapers}	3
{Bread,Milk}	3
{Diapers,Milk}	3

3-Itemset	Count
{Bread,Milk,Diapers}	3



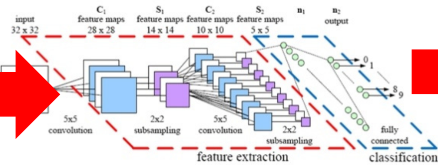
(FP-GROWTH uses requent pattern tree-based approach)

▪ Part II - Deep Learning to 'mine' product tags for DBs

(customer question in store: I want to have a perfume that looks like a 'gold bar')

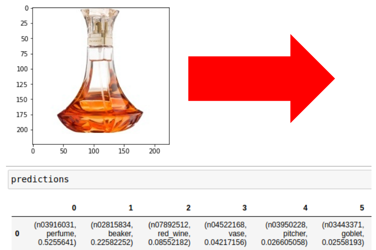
(name unknown, but shape & color known by customer)

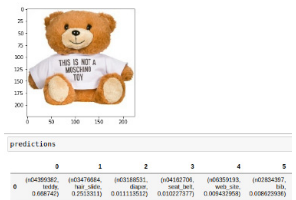
(databases - DBs – have no color or shape info)

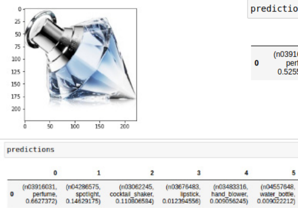


(shapes)

(colors with another script)







[12] Big Data Tips, Association Rules

[13] Performance Evaluation Apriori vs. FP-Growth

KI Part I: Unpersonalisierte Vorschläge

ON4OFF

Anonymisierte Verkaufsdaten per Händler ohne Kunde

Analyse mit Algorithmen der Künstlichen Intelligenz

Verbesserte Datenauswahl & KI Modell Parameter

Regeln welche Produkte zusammen gekauf werden

Händler Gibt

Transaktionen mit:
Produkt-IDs im Warenkorb

KI Algorithmen

Association Rule Mining
Apriori, FP-Growth, MAFIA

Modell-Tuning

KI Algorithmen-Parameter zur Anpassung beim Händler:
confidence, support, lift, Filiale

Händler Bekommt

Liste mit Regeln nach Muster:
Gekauft Produkt A → Kauf Produkt B & C
Gekauft Product D & E → Kauf Produkt F

KI Part III: Personalisierte Vorschläge (Details in Folien)

ON4OFF

Anonymisierte Verkaufsdaten per Kunde

Analyse mit Algorithmen der Künstlichen Intelligenz

Verbesserte Datenauswahl & KI Modell Parameter

Für jeden Kunden individuelle Kaufvorschläge

Händler Gibt

Transaktionen mit:
Kunden-ID
Produkt-IDs
Rating (falls vorhanden)

KI Algorithmen

Collaborative Filtering
Singular Value Decomposition (SVD), Deep Learning

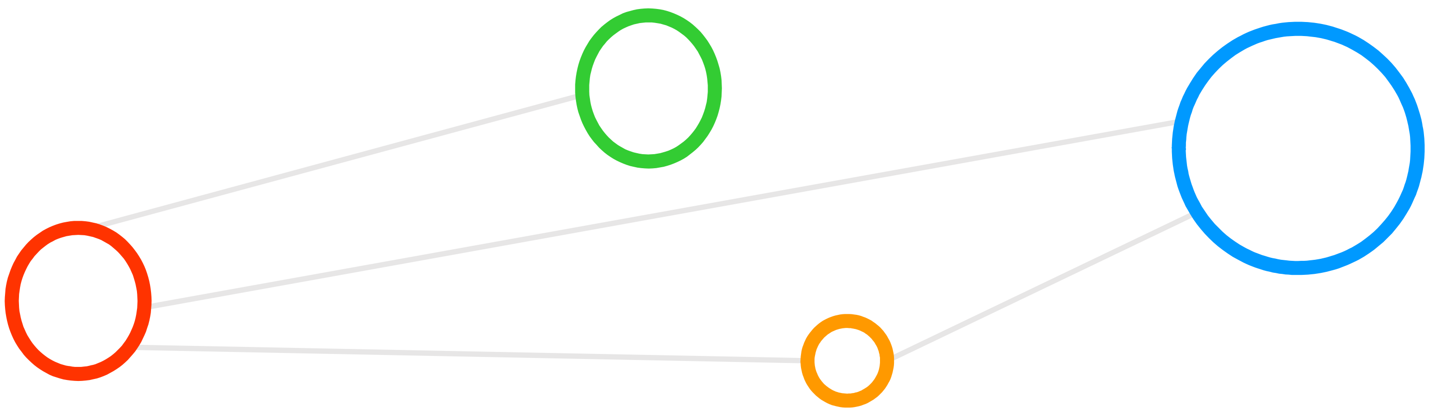
Modell-Tuning

KI Algorithmen-Parameter zur Anpassung beim Händler:
Rating-Modell, Filiale, Saison

Händler Bekommt

Liste mit Vorschlägen nach Muster:
Kunde A → Kauft Produkt B & C
Kunde A → Ähnlich wie Kunde B

Part III – Collaborative Filtering



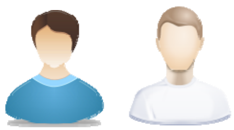
Executive Summary – Collaborative Filtering & Clustering Approaches

- Example: K-Means Clustering

[10] YouTube video,
Visualization of K-Means Clustering

- Example: Recommendation Engine
  - Using Collaborative Filtering via Google 'Colab'

[9] Google Colaboratory

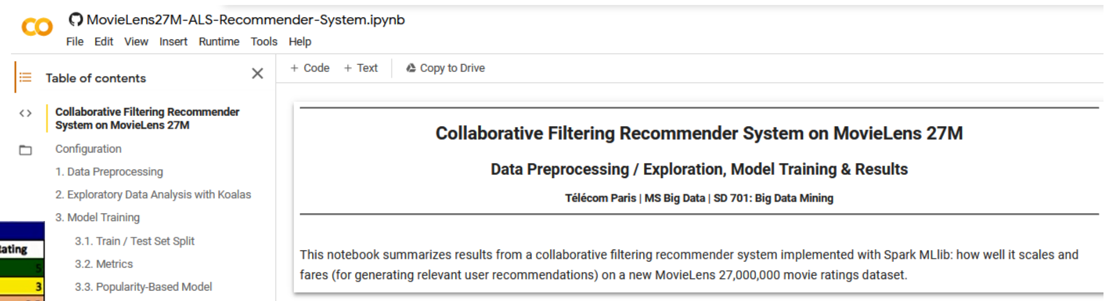


Customers	
ID	Name
1	Mike Jones
2	Steve Thomas
3	Peter Sloan
4	Vish Macnhi
5	Tim Albright

LEGEND
1
2
3
4
5

Movies		
ID	Name	Category
101	The Internship	Comedy
102	The Purge	Thriller
103	Much About Nothing	Comedy
104	Dirty Wars	Drama
105	Wish You Were Here	Drama
106	Syrup	Comedy
107	Fast & Furios	Thriller

Transactions		
Customer ID	Movie ID	Rating
1	101	5
1	102	3
1	103	2.5
2	101	2
2	102	2.5
2	103	5
2	104	2
3	101	2.5
3	104	4
3	105	4.5
3	107	5
4	101	5
4	103	3
4	104	4.5
4	106	4
5	101	4
5	102	3
5	103	2
5	104	4
5	105	3.5
5	106	4



item	User 1	User 2	User 3	User 4	User 5
101	5	2	2.5	5	4
102	3	2.5			3
103	2.5	5		3	2
104		2	4.5	4.5	4
105			4.5		3.5
106				4	4
107			5		

(similar personality?)

[8] Collaborative Filtering Recommender System on Colab

Another Form of ‘Data Mining‘ using Recommender Systems – Overview

- Content-based / Product-based Recommendation Systems
  - E.g. Netflix user has watched many cowboy movies in the past (focus on product feature)
  - Recommendation: movie classified in the database as having the ‘cowboy‘ genre tag
  - (not covered here as relatively straightforward to implement: e.g., DB lookup)
  - Might be still useful in combination with more elaborate systems if space in GUI is available

(e.g., vegan tag)

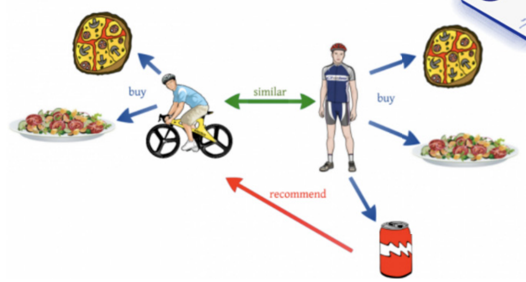
- Collaborative Filtering-based / Customer-based Recommendation Systems
  - E.g. Similiarity of the customer ratings for products (not focus on product feature)
  - Identify: looking at other customers that are most similar to this customer
  - Recommendation: products that are liked or preferred by the other ‘similiar customers‘
  - Focus in this lecture and on one concrete algorithm: matrix factorization

(similar personality?)

[14] Pieper.de Duefte

Collaborative Filtering – Methodology

- Methodology
  - Recommendation systems that leverage existing shopping/watching/listening behaviour patterns
  - Predicts what customers could like in future based on previous customers behavior patterns
  - Assumes that customers like products similar to other products they like, but also products that are liked by other people with similar taste
- Approach
  - Uses different machine learning methods
  - Collaborative filtering is a general concept and there are many algorithms (e.g., singular vector decomposition, neural networks, etc.)
  - Two main techniques: memory & model-based collaborative filtering

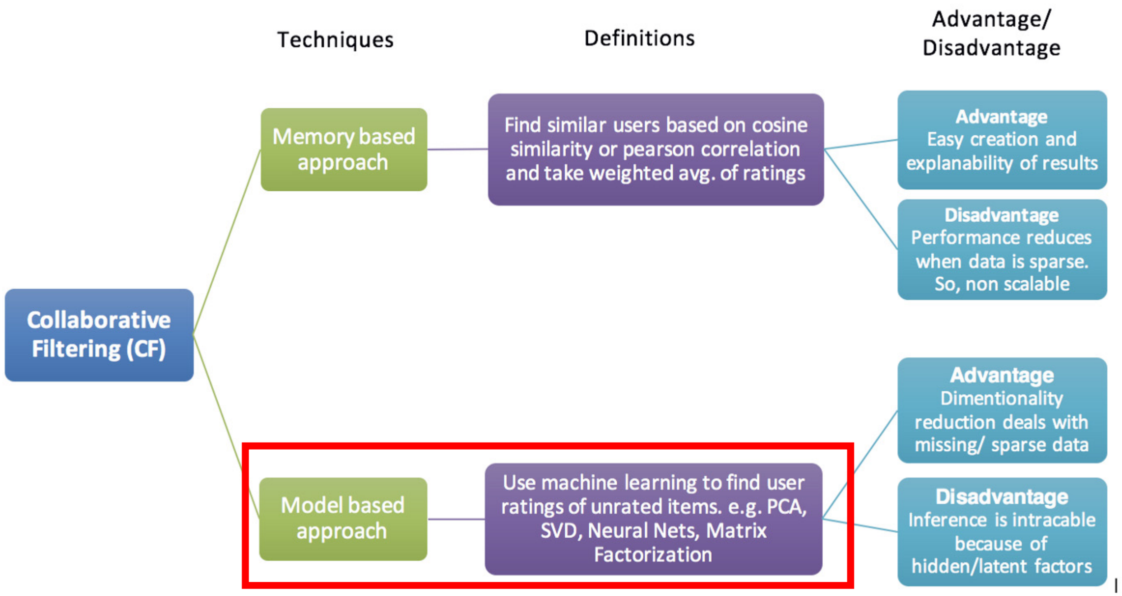


[16] Towards Data Science, Various CFs

[15] Big Data Tips, Recommender Systems

Collaborative Filtering – Memory and Model-based Techniques

- Two quite different approaches for the same problem
  - Popular approaches are based on low-dimensional factor models these days
  - Different approaches have different advantages and disadvantages and could be used both (if needed)



- **Memory-based approaches for Collaborative Filtering can quickly become computationally expensive, but enables better explainability**

- **Model-based approaches work with dimensionality reduction, but results are not easy to explain to store managers**

(this lecture will focus on Matrix Factorization – with a simple demonstration)

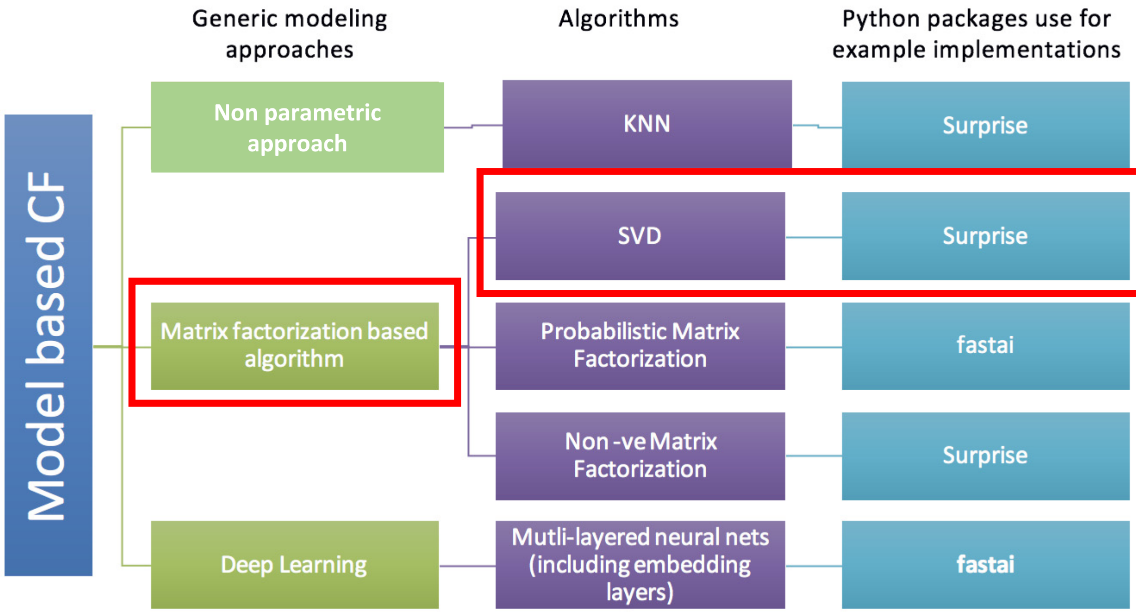
[16] Towards Data Science, Various CFs

Collaborative Filtering – Famous Dataset Example & Challenges

- Famous Example in Retail
  - Illustrating the underlying assumption that if a customer A has the same opinion/rating as a customer B on a certain product…
  - … A is more likely to have B's opinion on a different product as well than that of a randomly chosen customer
- Challenges
  - In real datasets millions or billions of transactions are used, including ratings if possible (otherwise buy & not buy only)
  - Unfortunately in practice not always ratings are existing
- Algorithms Benefit
  - Automation of the process using collaborative filtering algorithms
  - Patterns help to identify new opportunities and ways for cross-selling products to customers

[17] Towards Data Science, CF & Embeddings

Matrix Factorization-based Algorithms & Tool Support Examples



```
[14]: from surprise import accuracy
      accuracy.rmse(predictions)

      RMSE: 0.8770
[14]: 0.8769846736342048

[15]: data.head()
```

[15]:

	customer_id	product_id	rating
0	1	1193	5
1	1	661	3
2	1	914	3
3	1	3408	4
4	1	2355	5

```
[16]: iids = data['customer_id'].unique()

[17]: iids50 = data.loc[data['customer_id'] == 50, 'product_id']

[18]: iids_to_pred = np.setdiff1d(iids, iids50)

[19]: predictionset = [[50, iid, 4] for iid in iids_to_pred]
      realpredictions = algo.test(predictionset)
      realpredictions[0]

[19]: Prediction(uid=50, iid=1, r_ui=4, est=3.7017768774834705, details={'was_impossible': False})

[20]: pred_ratings = np.array([pred.est for pred in realpredictions])

[21]: index_max = pred_ratings.argmax()

[22]: topprediction = iids_to_pred[index_max]

[23]: print('Stefans Gui shows for user 50 the product_id {0} with predicted rating {1}'.format(topprediction, pred_ratings[index_max]))
      Stefans Gui shows for user 50 the product_id 2905 with predicted rating 4.381198237789029
```

surpr!se
A Python scikit for recommender systems.

[18] scikit-surprise library

fast.ai
Making neural nets uncool again

- **Deep Learning techniques change the experiences over the last decades and become more popular with very good accuracies and good packages (e.g., fast.ai) using innovative HPC & Cloud computing**

- **Scikit-surprise is a package specifically designed for recommendation systems and includes a variety of algorithms in Python useful for data mining tasks**

[19] fast.ai library

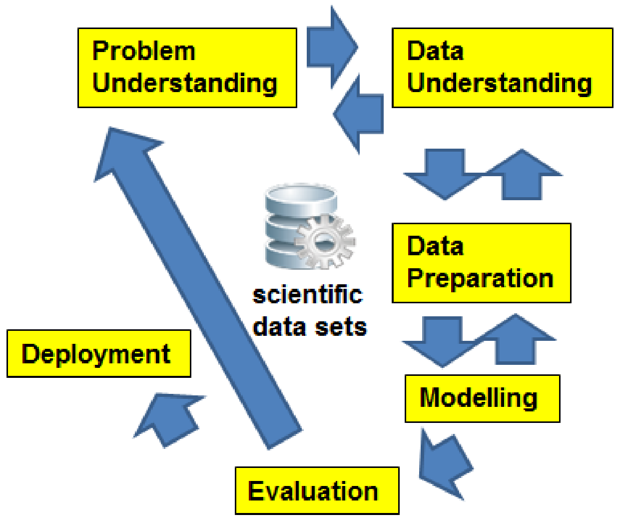
[16] Towards Data Science, Various CFs

Systematic Process to Support Learning From Data – Revisited

- Systematic data analysis guided by a 'standard process'
  - Cross-Industry Standard Process for Data Mining (CRISP-DM)

- A data mining project is guided by these six phases:
  (1) Problem Understanding;
  (2) Data Understanding;
  (3) Data Preparation;
  (4) Modeling;
  (5) Evaluation;
  (6) Deployment

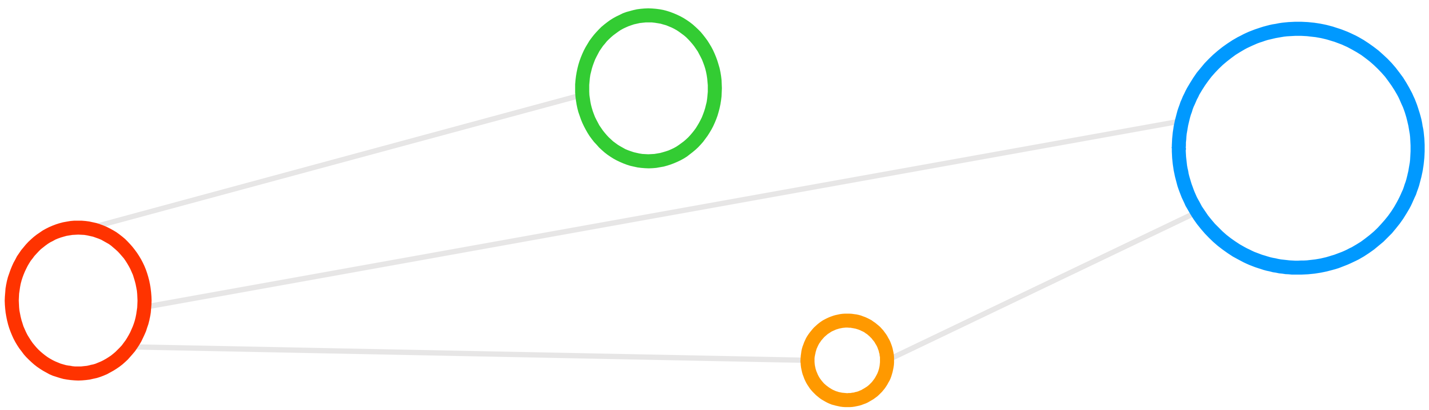
(learning takes place)



[20] CRISP-DM Model

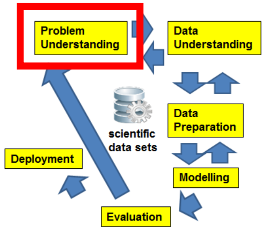
- Significant time goes into Steps 2-3 as well!

Collaborative Filtering in ON4OFF – General Understanding & Approach



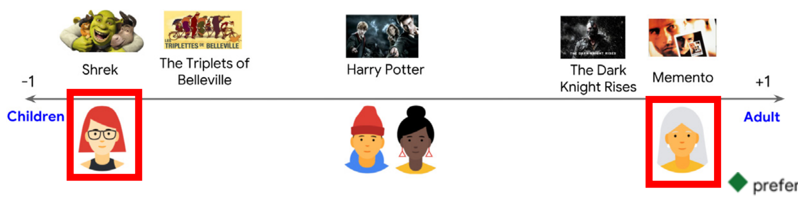
Movie Recommendation Example using Collaborative Filtering Techniques

- Given movie feedback matrix
  - Row represents a user
  - Column represents a movie
- Feedback encoding
  - One of two categories: explicit & implicit feedback
  - Example: feedback matrix is binary with a value of 1 that indicates interest in the movie
- Embeddings Approach
  - Can be learned automatically (no need for hand-engineering of features)
  - 1D Embedding Example



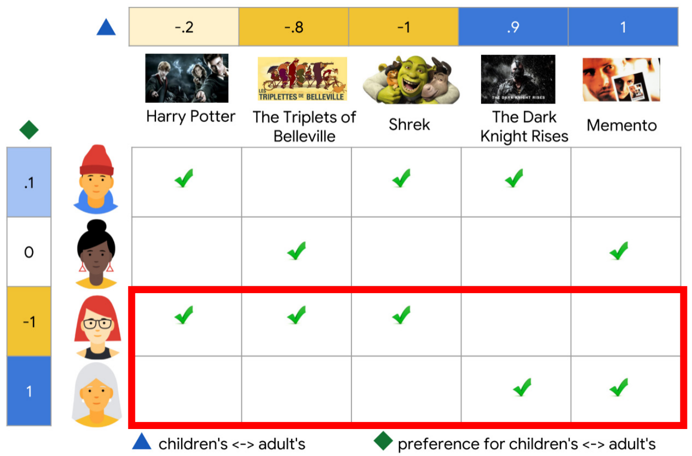
- **Collaborative filtering uses similarities between users and items simultaneously to provide recommendations**
- **Collaborative filtering models can recommend an item to user A based on the interests of a similar user B**
- **Explicit feedback in collaborative filtering means that users specify how much they liked a particular movie by providing a numerical rating**
- **Implicit feedback in collaborative filtering means that if a user watches a movie, the system infers that the user is interested**
- **The goal of collaborative filtering systems in movie ratings are to recommend (1) similarity to movies the user has liked in the past, and (2) movies that similar users liked and are not seen yet**

[1] Google Colab Exercise



(assign scalar for movies [-1,1])

(assign scalar for users [-1,1])

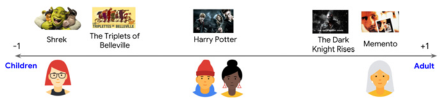


product of movie embedding & user embedding should be higher (closer to 1) for movies that we expect the user to like

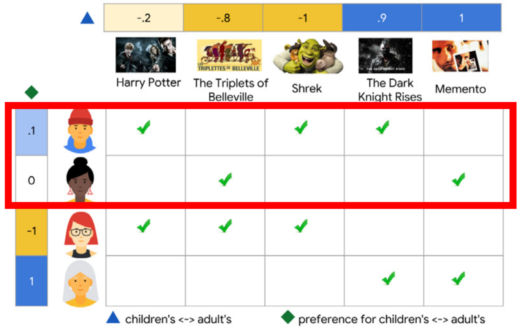
users watched these movies & preferences are well explained by this feature

Collaborative Filtering Techniques & Automatically Learned Embeddings

- Embeddings Approach
  - 1D feature not enough to explain preferences well
  - 2D Embedding Example: add a second feature
  - E.g. the degree to which each movie is a blockbuster or an arthouse movie



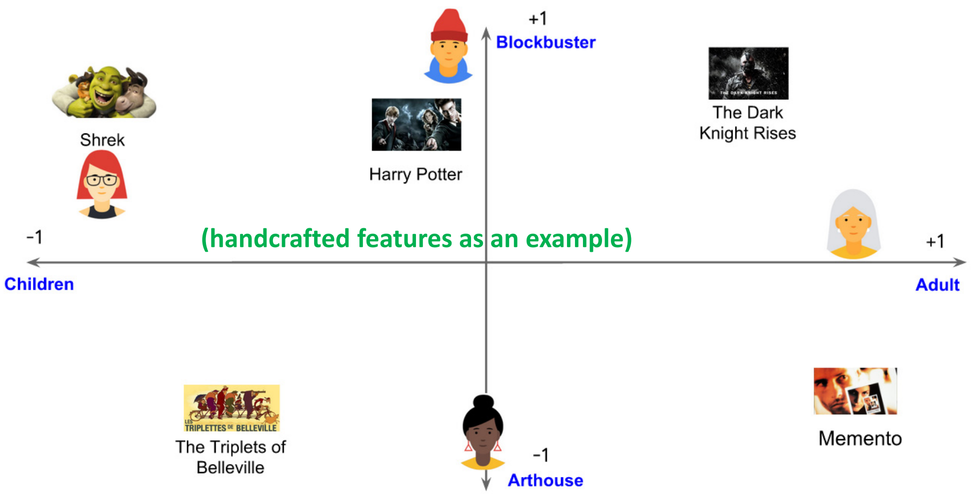
product of movie embedding & user embedding should be higher (closer to 1) for movies that we expect the user to like



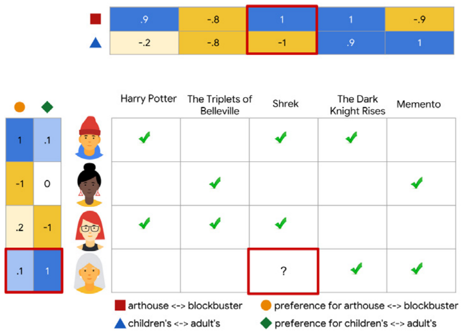
[1] Google Colab Exercise

users watched these movies & preferences are not well explained by this feature

- **The embedding space is an abstract representation common to both items and users, in which we can measure similarity or relevance using a similarity metric**



for each (user, item) pair, the dot product of the user embedding & the item embedding should be close to 1 when the user watched the movie, and to 0 otherwise



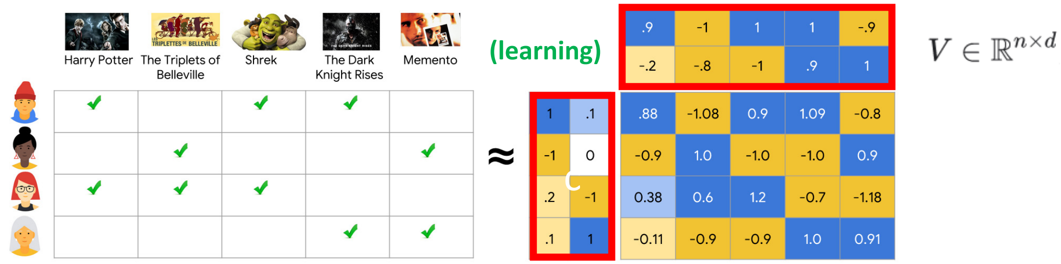
- **Embeddings can be learned automatically, which is the power of collaborative filtering models**
- **Embeddings of users with similar preferences will be close together**
- **Embeddings of movies liked by similar users will be close in the embedding space**

Example: (0.1 x 1) + (1 * -1) = - 0.9

Matrix Factorization Approaches as Simple Embedding Model

- Given movie feedback matrix A
  - Row represents a user: *m users*
  - Column represents a movie: *n movies*
- Model learns automatically:
  - User embedding matrix U (row i is the embedding for user i)
  - Movie embedding matrix V (row j is the embedding for movie j)
  - Learning = minimize 'errors'
- Embeddings
  - Have an embedding dimension d (here we have a 2D example)
  - Learned such that the product UV^T is a good approximation of matrix A

[1] Google Colab Exercise



dot product $\langle U_i, V_j \rangle \approx A_{i,j}$ (learning goal)

(objective function, cf. Lecture 6)

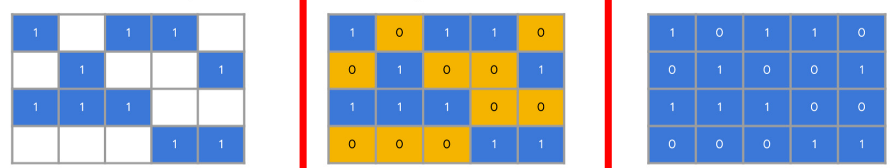
$$\min_{U \in \mathbb{R}^{m \times d},\, V \in \mathbb{R}^{n \times d}} \sum_{(i,j) \in \text{obs}} (A_{ij} - \langle U_i, V_j \rangle)^2$$

(minimize the sum of squared errors over all pairs of observed entries = Observed Only MF)

(optimization problem can be solved with SGD for example)

(treat the unobserved values as zero, and sum over all entries in the matrix)

$$\min_{U \in \mathbb{R}^{m \times d},\, V \in \mathbb{R}^{n \times d}} \|A - UV^T\|_F^2$$



Observed Only MF: $\Sigma_{(i,j) \in obs} (A_{ij} - U_i . V_j)^2$

Weighted MF: $\Sigma_{(i,j) \in obs} (A_{ij} - U_i . V_j)^2 + w_0 \Sigma_{(i,j) \notin obs} (0 - U_i . V_j)^2$

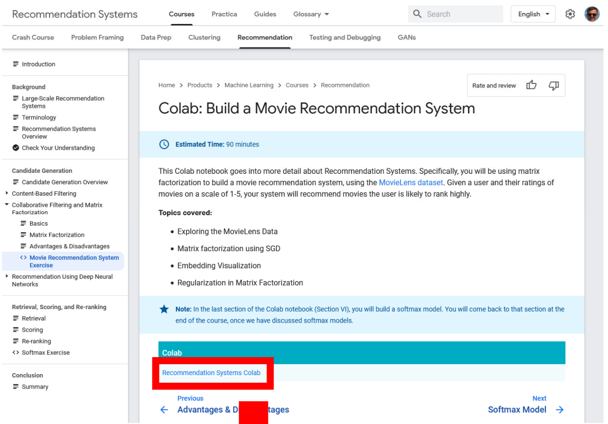
SVD: $|A - U V^T|_F^2 = \Sigma_{(i,j)} (A_{ij} - U_i . V_j)^2$

(SVD = singular value decomposition, poor generalization in sparse movie rating matrix setup)

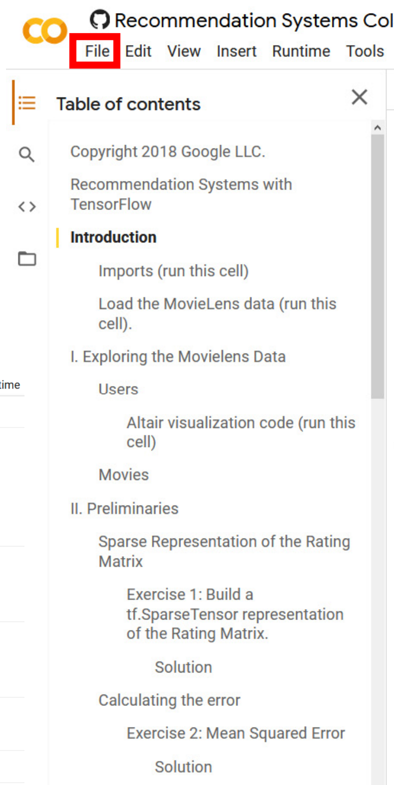
$$\min_{U \in \mathbb{R}^{m \times d},\, V \in \mathbb{R}^{n \times d}} \sum_{(i,j) \in \text{obs}} (A_{ij} - \langle U_i, V_j \rangle)^2 + \boxed{w_0} \sum_{(i,j) \notin \text{obs}} (\langle U_i, V_j \rangle)^2$$

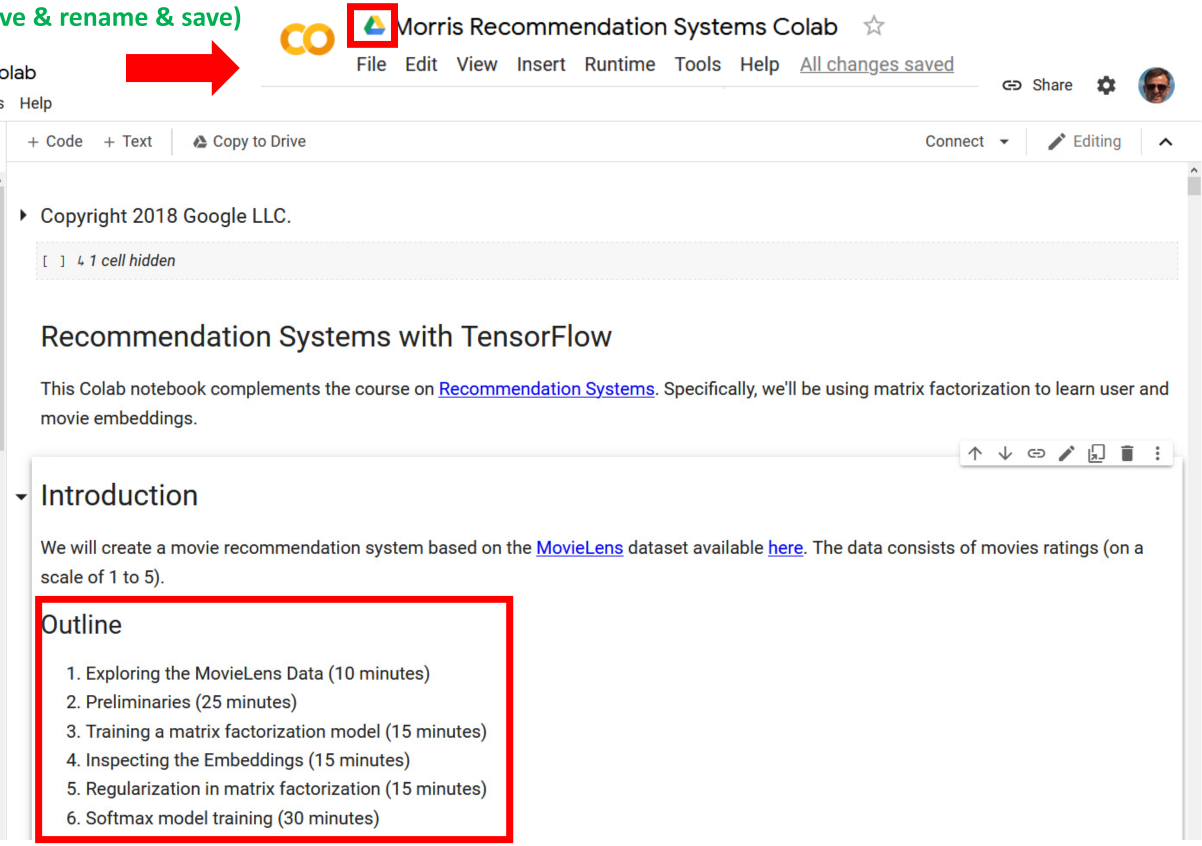
(sum over observed entries + sum over not observed entries using hyperparameter w_0)

Google Colab – Movie Rental Recommendation Notebook & Porting Juelich

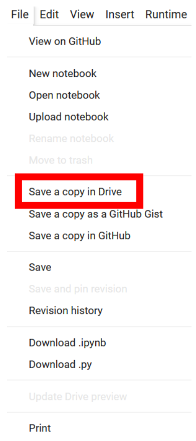


(save copy in Drive & rename & save)





[1] Google Colab Exercise



[9] Google Colaboratory

Google Colab – Movie Rental Recommendation Notebook – Connect & Setup

Morris Recommendation Systems Colab

File Edit View Insert Runtime Tools Help

Comment Share

+ Code + Text

Connect

Editing

Table of contents

- **Copyright 2018 Google LLC.**
- Recommendation Systems with TensorFlow
- Introduction
  - Imports (run this cell)
  - Load the MovieLens data (run this cell).
- I. Exploring the Movielens Data
  - Users
    - Altair visualization code (run this cell)
  - Movies
- II. Preliminaries
  - Sparse Representation of the Rating Matrix
    - Exercise 1: Build a tf.SparseTensor representation of the Rating Matrix.
      - Solution
  - Calculating the error
    - Exercise 2: Mean Squared Error
      - Solution
      - Alternate Solution
    - Exercise 3 (Optional): adding your own ratings to the data set
      - Run to create a spreadsheet, then use it to enter your ratings.
      - Run to load your ratings.
- III. Training a Matrix Factorization model
  - CFModel helper class (run this cell)

Imports (run this cell)

Connect to hosted runtime

Connect to local runtime

Manage sessions

Focus the last run cell

```
[ ] # @title Imports (run this cell)
    from __future__ import print_function

    import numpy as np
    import pandas as pd
    import collections
    from mpl_toolkits.mplot3d import Axes3D
    from IPython import display
    from matplotlib import pyplot as plt
    import sklearn
    import sklearn.manifold
    import tensorflow.compat.v1 as tf
    tf.disable_v2_behavior()
    tf.logging.set_verbosity(tf.logging.ERROR)
```

```
    # Add some convenience functions to Pandas DataFrame.
    pd.options.display.max_rows = 10
    pd.options.display.float_format = '{:.3f}'.format
    def mask(df, key, function):
      """Returns a filtered dataframe, by applying function to key"""
      return df[function(df[key])]

    def flatten_cols(df):
      df.columns = [' '.join(col).strip() for col in df.columns.values]
      return df

    pd.DataFrame.mask = mask
    pd.DataFrame.flatten_cols = flatten_cols
```

```
    # Install Altair and activate its colab renderer.
    print("Installing Altair...")
    !pip install git+git://github.com/altair-viz/altair.git
    import altair as alt
    alt.data_transformers.enable('default', max_rows=None)
    alt.renderers.enable('colab')
    print("Done installing Altair.")
```

```
    # Install spreadsheets and import authentication module.
    USER_RATINGS = False
    !pip install --upgrade -q gspread
    from google.colab import auth
    import gspread
    from oauth2client.client import GoogleCredentials
```

- Recommendation engine is built using TensorFlow deep learning package
- Recommendation engine script requires imports from various useful libraries: e.g., pandas, numpy, sklearn, etc.

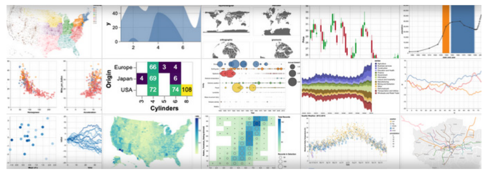
- Manipulating Pandas DataFrame options and adding selected convenient functions

- Installing Altair and importing the library
- Altair is a library for declarative visualization in Python and offers very good interactive visualizations for data analysis

[23] gspread API

- Installing and importing gspread that is a Python API for Google Sheets
- Optional as this enables own ratings to inject into the recommendation system (if needed)

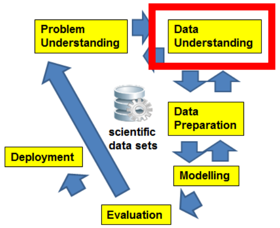
```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/compat/v2_compat.py:96: disable_resource_variables (from tensorflow.python.ops.variable_scope) is deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
Installing Altair...
Collecting git+git://github.com/altair-viz/altair.git
  Cloning git://github.com/altair-viz/altair.git to /tmp/pip-req-build-7bujxoat
  Running command git clone -q git://github.com/altair-viz/altair.git /tmp/pip-req-build-7bujxoat
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
Requirement already satisfied: jsonschema in /usr/local/lib/python3.6/dist-packages (from altair==4.2.0.dev0) (2.6.0)
Requirement already satisfied: toolz in /usr/local/lib/python3.6/dist-packages (from altair==4.2.0.dev0) (0.11.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from altair==4.2.0.dev0) (1.18.5)
Requirement already satisfied: entrypoints in /usr/local/lib/python3.6/dist-packages (from altair==4.2.0.dev0) (0.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.6/dist-packages (from altair==4.2.0.dev0) (2.11.2)
Requirement already satisfied: pandas>=0.18 in /usr/local/lib/python3.6/dist-packages (from altair==4.2.0.dev0) (1.1.4)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.6/dist-packages (from jinja2->altair==4.2.0.dev0) (1.1.1)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.18->altair==4.2.0.dev0) (2.8.1)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.18->altair==4.2.0.dev0) (2018.9)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.6/dist-packages (from python-dateutil>=2.7.3->pandas>=0.18->altair==4.2.0.dev0) (1.15.0)
Building wheels for collected packages: altair
  Building wheel for altair (PEP 517) ... done
  Created wheel for altair: filename=altair-4.2.0.dev0-cp36-none-any.whl size=730019 sha256=105046ec4046345511f2d0b38e9dd73592f333d625dc73c35f8f11136ce9e81f
  Stored in directory: /tmp/pip-ephem-wheel-cache-tfh39cna/wheels/01/fd/91/025b6149b3949af76e93b3b3ceca5bf12cbdebc98fa46f9ec6
Successfully built altair
Installing collected packages: altair
  Found existing installation: altair 4.1.0
    Uninstalling altair-4.1.0:
      Successfully uninstalled altair-4.1.0
Successfully installed altair-4.2.0.dev0
Done installing Altair.
```



[21] Altair Visualization library

[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Data Understanding



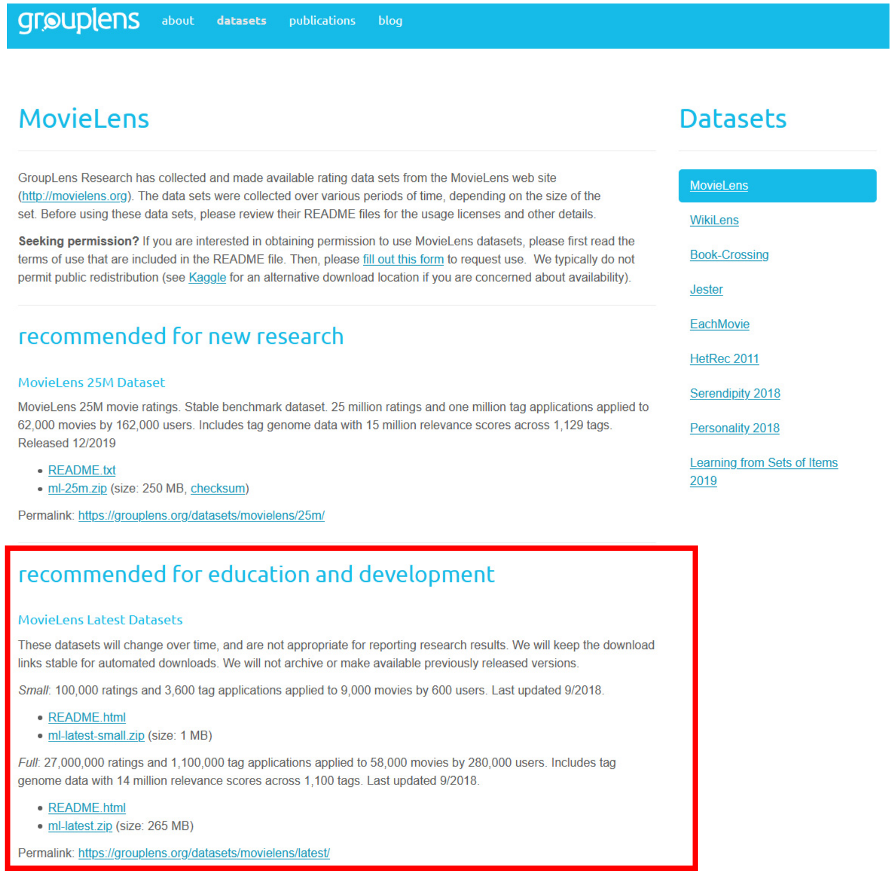
Training Examples

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$$

(historical records, groundtruth data, examples)

- Movielens Dataset
  - http://files.grouplens.org/datasets/movielens/ml-100k.zip

```python
# @title Load the MovieLens data (run this cell).

# Download MovieLens data.
print("Downloading movielens data...")
from urllib.request import urlretrieve
import zipfile

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'))
```



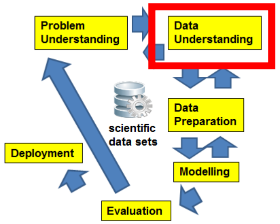
[22] MovieLens Dataset

[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Check Dataset

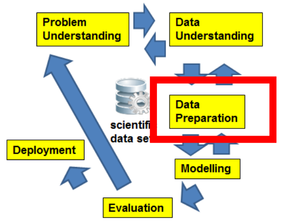
- Check available data
  - Users
  - Ratings
  - Genre
  - Movies

- **The MovieLens dataset consists of data about users, ratings about movies, the genre of movies and information about movies**




```python
# 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')
```

```python
# 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))

# Compute the number of movies to which a genre is assigned.
genre_occurences = movies[genre_cols].sum().to_dict()

# Since some movies can belong to more than one genre, we create different
# 'genre' columns as follows:
# - all_genres: all the active genres of the movie.
# - genre: randomly sampled from the active genres.
def mark_genres(movies, genres):
  def get_random_genre(gs):
    active = [genre for genre, g in zip(genres, gs) if g==1]
    if len(active) == 0:
      return 'Other'
    return np.random.choice(active)
  def get_all_genres(gs):
    active = [genre for genre, g in zip(genres, gs) if g==1]
    if len(active) == 0:
      return 'Other'
    return '-'.join(active)
  movies['genre'] = [
      get_random_genre(gs) for gs in zip(*[movies[genre] for genre in genres])]
  movies['all_genres'] = [
      get_all_genres(gs) for gs in zip(*[movies[genre] for genre in genres])]

mark_genres(movies, genre_cols)

# Create one merged DataFrame containing all the movielens data.
movielens = ratings.merge(movies, on='movie_id').merge(users, on='user_id')
```



[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Training & Testing

```python
# Utility to split the data into training and test sets.
def split_dataframe(df, holdout_fraction=0.1):
  """Splits a DataFrame into training and test sets.
  Args:
    df: a dataframe.
    holdout_fraction: fraction of dataframe rows to use in the test set.
  Returns:
    train: dataframe for training
    test: dataframe for testing
  """
  test = df.sample(frac=holdout_fraction, replace=False)
  train = df[~df.index.isin(test.index)]
  return train, test
```

```
Downloading movielens data...
Done. Dataset contains:
b'943 users\n1682 items\n100000 ratings\n'
```

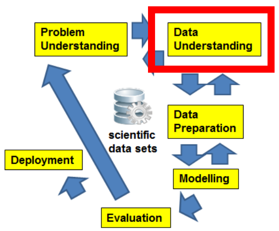
- **Like in previous examples of machine learning and data mining, also for learning the recommendation engine we split the available dataset into two disjunct datasets: train and test datasets**

'training set' *'test set'*

Training Examples

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$$

(historical records, groundtruth data, examples)



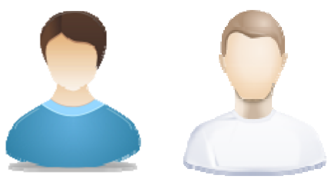
(understanding some basic statistics describing numeric user features)

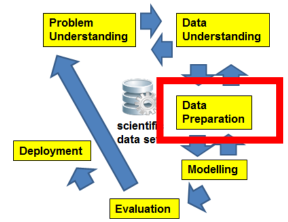
```
[3] users.describe()
```

	age
count	943.000
mean	34.052
std	12.193
min	7.000
25%	25.000
50%	31.000
75%	43.000
max	73.000

```
[4] users.describe(include=[np.object])
```

	user_id	sex	occupation	zip_code
count	943	943	943	943
unique	943	2	21	795
top	805	M	student	55414
freq	1	670	196	9





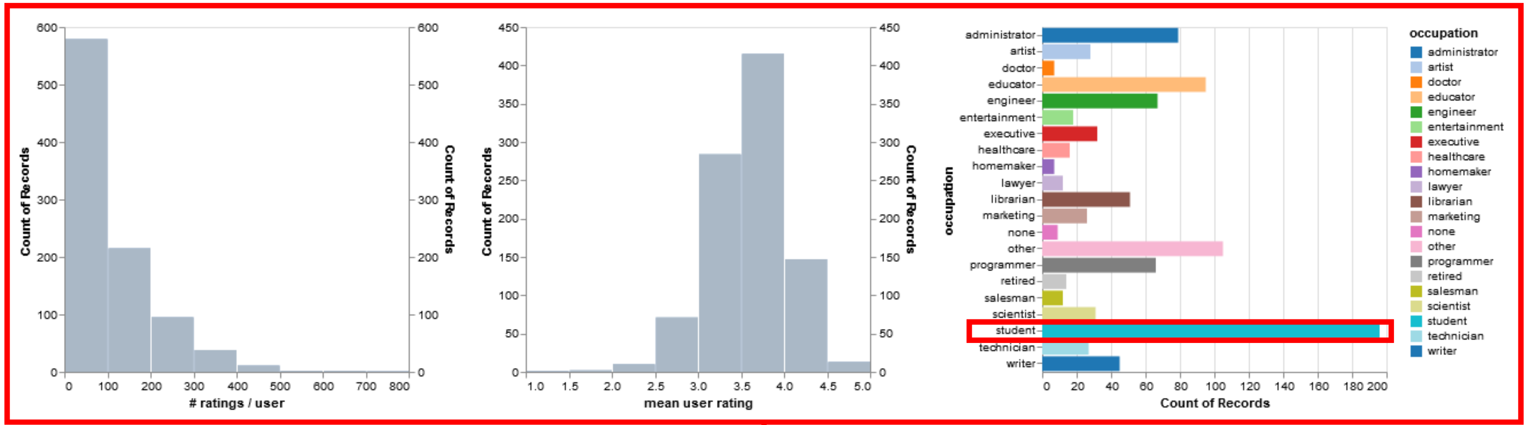
(warning appears if you not use the notebook)

Warning: you are connected to a GPU runtime, but not utilizing the GPU. Change to a standard runtime ✕

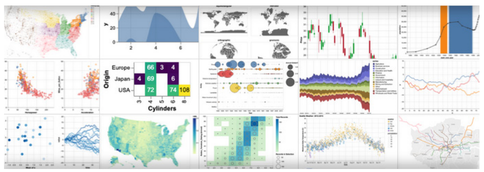
[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Data Visualization

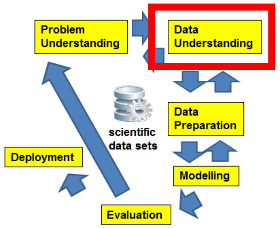
```
[5] # @title Altair visualization code (run this cell)
    # The following functions are used to generate interactive Altair charts.
    # We will display histograms of the data, sliced by a given attribute.

    # Create filters to be used to slice the data.
    occupation_filter = alt.selection_multi(fields=["occupation"])
    occupation_chart = alt.Chart().mark_bar().encode(
        x="count()",
        y=alt.Y("occupation:N"),
        color=alt.condition(
            occupation_filter,
            alt.Color("occupation:N", scale=alt.Scale(scheme='category20')),
            alt.value("lightgray")),
    ).properties(width=300, height=300, selection=occupation_filter)

    # A function that generates a histogram of filtered data.
    def filtered_hist(field, label, filter):
      """Creates a layered chart of histograms.
      The first layer (light gray) contains the histogram of the full data, and the
      second contains the histogram of the filtered data.
      Args:
        field: the field for which to generate the histogram.
        label: String label of the histogram.
        filter: an alt.Selection object to be used to filter the data.
      """
      base = alt.Chart().mark_bar().encode(
          x=alt.X(field, bin=alt.Bin(maxbins=10), title=label),
          y="count()",
      ).properties(
          width=300,
      )
      return alt.layer(
          base.transform_filter(filter),
          base.encode(color=alt.value('lightgray'), opacity=alt.value(.7)),
      ).resolve_scale(y='independent')
```



- Interactive data visualization techniques are an important approach to have a better data understanding
- E.g. histograms to further understand the distribution of the users, distribution of ratings per user, occupation in the right chart filters the data by that occupation
- E.g. data understanding observed: very high number of ratings from students → recommendation bias



[21] Altair Visualization library



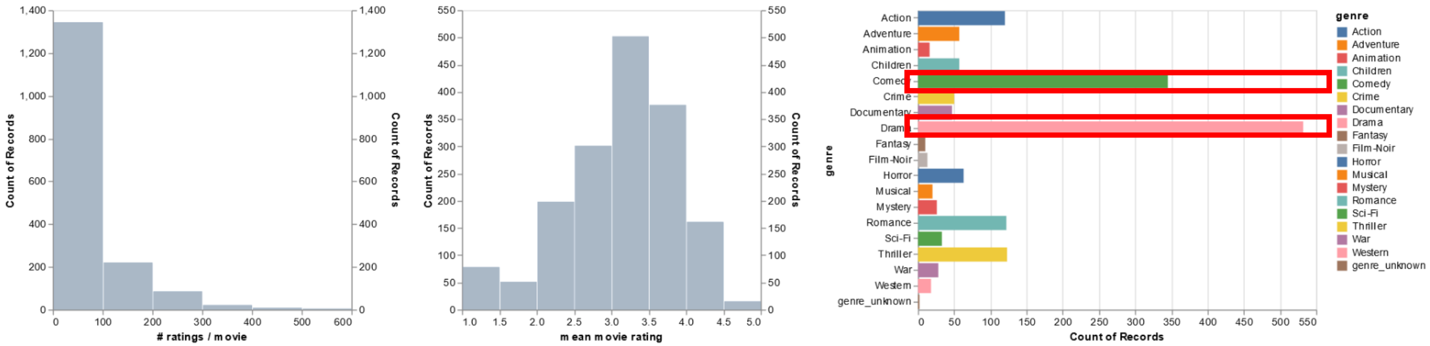

```
[6] users_ratings = (
        ratings
        .groupby('user_id', as_index=False)
        .agg({'rating': ['count', 'mean']})
        .flatten_cols()
        .merge(users, on='user_id')
    )

    # Create a chart for the count, and one for the mean.
    alt.hconcat(
        filtered_hist('rating count', '# ratings / user', occupation_filter),
        filtered_hist('rating mean', 'mean user rating', occupation_filter),
        occupation_chart,
        data=users_ratings)
```

(students have a lot of records compared to other users who rated movies)

[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Movie Data

```
[7] movies_ratings = movies.merge(
        ratings
        .groupby('movie_id', as_index=False)
        .agg({'rating': ['count', 'mean']})
        .flatten_cols(),
        on='movie_id')

    genre_filter = alt.selection_multi(fields=['genre'])
    genre_chart = alt.Chart().mark_bar().encode(
        x="count()",
        y=alt.Y('genre'),
        color=alt.condition(
            genre_filter,
            alt.Color("genre:N"),
            alt.value('lightgray'))
    ).properties(height=300, selection=genre_filter)
```




```
[8] (movies_ratings[['title', 'rating count', 'rating mean']]
     .sort_values('rating count', ascending=False)
     .head(10))
```

	title	rating count	rating mean
49	Star Wars (1977)	583	4.358
257	Contact (1997)	509	3.804
99	Fargo (1996)	508	4.156
180	Return of the Jedi (1983)	507	4.008
293	Liar Liar (1997)	485	3.157
285	English Patient, The (1996)	481	3.657
287	Scream (1996)	478	3.441
0	Toy Story (1995)	452	3.878
299	Air Force One (1997)	431	3.631
120	Independence Day (ID4) (1996)	429	3.438

```
[10] # Display the number of ratings and average rating per movie.
     alt.hconcat(
         filtered_hist('rating count', '# ratings / movie', genre_filter),
         filtered_hist('rating mean', 'mean movie rating', genre_filter),
         genre_chart,
         data=movies_ratings)
```

```
[9] (movies_ratings[['title', 'rating count', 'rating mean']]
     .mask('rating count', lambda x: x > 20)
     .sort_values('rating mean', ascending=False)
     .head(10))
```

	title	rating count	rating mean
407	Close Shave, A (1995)	112	4.491
317	Schindler's List (1993)	298	4.466
168	Wrong Trousers, The (1993)	118	4.466
482	Casablanca (1942)	243	4.457
113	Wallace & Gromit: The Best of Aardman Animatio...	67	4.448
63	Shawshank Redemption, The (1994)	283	4.445
602	Rear Window (1954)	209	4.388
11	Usual Suspects, The (1995)	267	4.386
49	Star Wars (1977)	583	4.358
177	12 Angry Men (1957)	125	4.344

[1] Google Colab Exercise

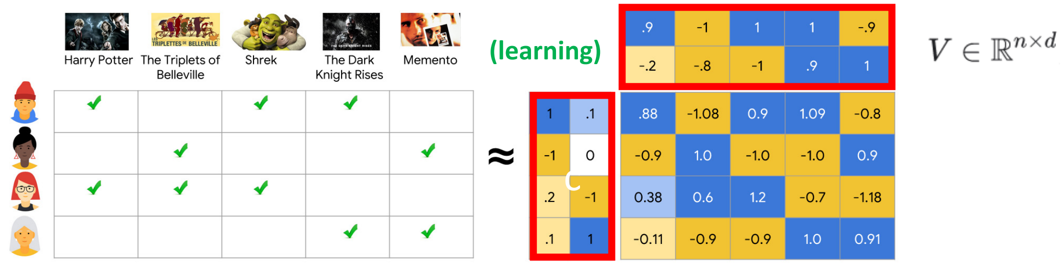
Starting the Modeling Approach – Matrix Factorization – Revisited

- Factorize the ratings matrix A
  - Into the product of a user embedding matrix U and movie embedding matrix V

$$A \approx UV^\top \text{ with } U = \begin{bmatrix} u_1 \\ \vdots \\ u_N \end{bmatrix} \text{ and } V = \begin{bmatrix} v_1 \\ \vdots \\ v_M \end{bmatrix}.$$

Here

- N is the number of users,
- M is the number of movies,
- A_{ij} is the rating of the jth movies by the ith user,
- each row U_i is a d-dimensional vector (embedding) representing user i,
- each row V_j is a d-dimensional vector (embedding) representing movie j,
- the prediction of the model for the (i, j) pair is the dot product $\langle U_i, V_j \rangle$.



$U \in \mathbb{R}^{m \times d}$ A $\in R^{m \times n}$

dot product $\langle U_i, V_j \rangle$ $\approx$ $A_{i,j}$ (learning goal)

- **A rating matrix used in collaborative filtering with matrix factorization in the movie recommendation example shows that most of the entries are unobserved, since users will only rate a small subset of all movies**

- Using TensorFlow (cf. Lecture 6 & 7) SparseTensor
  - Sparse representation of the rating matrix A required
  - Rating matrix could be very large (many users and many movies)
  - Example:

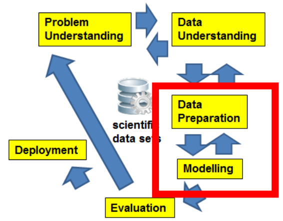
user_id	movie_id	rating
0	0	5.0
0	1	3.0
1	3	1.0

(full rating matrix A example)

$$A = \begin{bmatrix} 5.0 & 3.0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix}$$

(efficient SparseTensor representation example)

```
SparseTensor(
  indices=[[0, 0], [0, 1], [1,3]],
  values=[5.0, 3.0, 1.0],
  dense_shape=[2, 4])
```



[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Rating Matrix & Error

```
[11] def build_rating_sparse_tensor(ratings_df):
      """
      Args:
        ratings_df: a pd.DataFrame with `user_id`, `movie_id` and `rating` columns.
      Returns:
        A tf.SparseTensor representing the ratings matrix.
      """
      # ========================= Complete this section ============================
      # indices =
      # values =
      # ============================================================================

      return tf.SparseTensor(
          indices=indices,
          values=values,
          dense_shape=[users.shape[0], movies.shape[0]])
```

(a function that maps from ratings DataFrame to a TensorFlow SparseTensor)

(solution)

```
[12] #@title Solution
    def build_rating_sparse_tensor(ratings_df):
      """
      Args:
        ratings_df: a pd.DataFrame with `user_id`, `movie_id` and `rating` columns.
      Returns:
        a tf.SparseTensor representing the ratings matrix.
      """
      indices = ratings_df[['user_id', 'movie_id']].values
      values = ratings_df['rating'].values
      return tf.SparseTensor(
          indices=indices,
          values=values,
          dense_shape=[users.shape[0], movies.shape[0]])
```

```
[13] def sparse_mean_square_error(sparse_ratings, user_embeddings, movie_embeddings):
      """
      Args:
        sparse_ratings: A SparseTensor rating matrix, of dense_shape [N, M]
        user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
          dimension, such that U_i is the embedding of user i.
        movie_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
          dimension, such that V_j is the embedding of movie j.
      Returns:
        A scalar Tensor representing the MSE between the true ratings and the
          model's predictions.
      """
      # ========================= Complete this section ============================
      # loss =
      # ============================================================================
      return loss
```

(solution, but infeasible for 'big data')

```
[14] #@title Solution
    def sparse_mean_square_error(sparse_ratings, user_embeddings, movie_embeddings):
      """
      Args:
        sparse_ratings: A SparseTensor rating matrix, of dense_shape [N, M]
        user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
          dimension, such that U_i is the embedding of user i.
        movie_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
          dimension, such that V_j is the embedding of movie j.
      Returns:
        A scalar Tensor representing the MSE between the true ratings and the
          model's predictions.
      """
      predictions = tf.gather_nd(
          tf.matmul(user_embeddings, movie_embeddings, transpose_b=True),
          sparse_ratings.indices)
      loss = tf.losses.mean_squared_error(sparse_ratings.values, predictions)
      return loss
```

[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Mean Squarer Error

- Error to guide the learning process
  - Calculating the error to improve learning and to measure the approximation error
  - Model approximates the ratings matrix A by a low-rank product UV^T
- First Approach (last slide)
  - Initially Mean Squared Error (MSE) of observed entries only

$$\text{MSE}(A, UV^\top) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (A_{ij} - (UV^\top)_{ij})^2$$
$$= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (A_{ij} - \langle U_i, V_j \rangle)^2$$

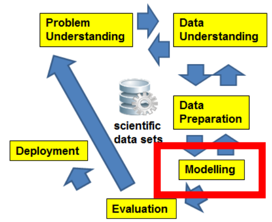
where Ω is the set of observed ratings, and $|\Omega|$ is the cardinality of Ω.

  - Compute the full prediction matrix UV^T costly, then gather entries corresponding to the observed pairs not feasible for 'big data'

$O(NM)$ (memory footprint) $N = 943, M = 1682$ (simple example here ok since it fits into memory)

dot product $\langle U_i, V_j \rangle \approx A_{i,j}$ (learning goal)

```
[15] #@title Alternate Solution
     def sparse_mean_square_error(sparse_ratings, user_embeddings, movie_embeddings):
       """
       Args:
         sparse_ratings: A SparseTensor rating matrix, of dense_shape [N, M]
         user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
           dimension, such that U_i is the embedding of user i.
         movie_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
           dimension, such that V_j is the embedding of movie j.
       Returns:
         A scalar Tensor representing the MSE between the true ratings and the
           model's predictions.
       """
       predictions = tf.reduce_sum(
           tf.gather(user_embeddings, sparse_ratings.indices[:, 0]) *
           tf.gather(movie_embeddings, sparse_ratings.indices[:, 1]),
           axis=1)
       loss = tf.losses.mean_squared_error(sparse_ratings.values, predictions)
       return loss
```



- Second Approach (above)
  - Only gather the embeddings of the observed pairs, then compute their dot products
  - More efficient to fit into memory for 'big data'

$O(|\Omega| d)$ where d is the embedding dimension.

$|\Omega| = 10^5$ (embedding dimension on order of 10)

(Note: next notebook steps in adding new ratings via spread sheets is not used here)

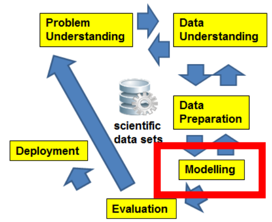
[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Model Building

- Approach
  - Class to train a matrix factorization model using stochastic gradient descent (cf. Lecture 6)

```
[16] # @title CFModel helper class (run this cell)
     class CFModel(object):
       """Simple class that represents a collaborative filtering model"""
       def __init__(self, embedding_vars, loss, metrics=None):
         """Initializes a CFModel.
         Args:
           embedding_vars: A dictionary of tf.Variables.
           loss: A float Tensor. The loss to optimize.
           metrics: optional list of dictionaries of Tensors. The metrics in each
             dictionary will be plotted in a separate figure during training.
         """
         self._embedding_vars = embedding_vars
         self._loss = loss
         self._metrics = metrics
         self._embeddings = {k: None for k in embedding_vars}
         self._session = None

       @property
       def embeddings(self):
         """The embeddings dictionary."""
         return self._embeddings

       def train(self, num_iterations=100, learning_rate=1.0, plot_results=True,
                 optimizer=tf.train.GradientDescentOptimizer):
         """Trains the model.
         Args:
           iterations: number of iterations to run.
           learning_rate: optimizer learning rate.
           plot_results: whether to plot the results at the end of training.
           optimizer: the optimizer to use. Default to GradientDescentOptimizer.
         Returns:
           The metrics dictionary evaluated at the last iteration.
         """
```

```
with self._loss.graph.as_default():
  opt = optimizer(learning_rate)
  train_op = opt.minimize(self._loss)
  local_init_op = tf.group(
      tf.variables_initializer(opt.variables()),
      tf.local_variables_initializer())
  if self._session is None:
    self._session = tf.Session()
    with self._session.as_default():
      self._session.run(tf.global_variables_initializer())
      self._session.run(tf.tables_initializer())
      tf.train.start_queue_runners()

with self._session.as_default():
  local_init_op.run()
  iterations = []
  metrics = self._metrics or ({},)
  metrics_vals = [collections.defaultdict(list) for _ in self._metrics]

  # Train and append results.
  for i in range(num_iterations + 1):
    _, results = self._session.run((train_op, metrics))
    if (i % 10 == 0) or i == num_iterations:
      print("\r iteration %d: " % i + ", ".join(
            ["%s=%f" % (k, v) for r in results for k, v in r.items()]),
            end='')
      iterations.append(i)
      for metric_val, result in zip(metrics_vals, results):
        for k, v in result.items():
          metric_val[k].append(v)

  for k, v in self._embedding_vars.items():
    self._embeddings[k] = v.eval()

  if plot_results:
    # Plot the metrics.
    num_subplots = len(metrics)+1
    fig = plt.figure()
    fig.set_size_inches(num_subplots*10, 8)
    for i, metric_vals in enumerate(metrics_vals):
      ax = fig.add_subplot(1, num_subplots, i+1)
      for k, v in metric_vals.items():
        ax.plot(iterations, v, label=k)
      ax.set_xlim([1, num_iterations])
      ax.legend()
  return results
```



- **After training a matrix factorization model using stochastic gradient descent (SGD) we obtain the trained embeddings via a model.embeddings dictionary that in turn is used to perform recommendations**

`model.embeddings`

(learning)

$$\begin{bmatrix} .9 & -1 & 1 & 1 & -.9 \\ -.2 & -.8 & -1 & .9 & 1 \end{bmatrix}$$

$$\approx \begin{bmatrix} 1 & .1 \\ -1 & 0 \\ .2 & -1 \\ .1 & 1 \end{bmatrix} \begin{bmatrix} .88 & -1.08 & 0.9 & 1.09 & -0.8 \\ -0.9 & 1.0 & -1.0 & -1.0 & 0.9 \\ 0.38 & 0.6 & 1.2 & -0.7 & -1.18 \\ -0.11 & -0.9 & -0.9 & 1.0 & 0.91 \end{bmatrix}$$

Google Colab – Movie Rental Recommendation Notebook – Model Learning

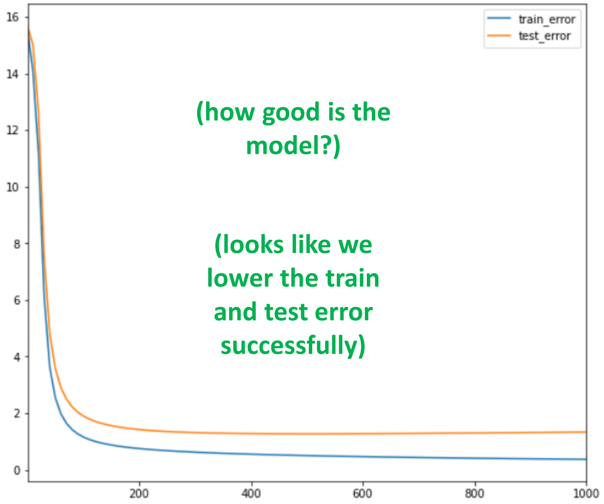
```
[17] def build_model(ratings, embedding_dim=3, init_stddev=1.):
       """
       Args:
         ratings: a DataFrame of the ratings
         embedding_dim: the dimension of the embedding vectors.
         init_stddev: float, the standard deviation of the random initial embeddings.
       Returns:
         model: a CFModel.
       """
       # Split the ratings DataFrame into train and test.
       train_ratings, test_ratings = split_dataframe(ratings)
       # SparseTensor representation of the train and test datasets.
       # ======================== Complete this section ============================
       # A_train =
       # A_test =
       # ===========================================================================
       # Initialize the embeddings using a normal distribution.
       U = tf.Variable(tf.random_normal(
           [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
       V = tf.Variable(tf.random_normal(
           [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))
       # ======================== Complete this section ============================
       # train_loss =
       # test_loss =
       # ===========================================================================
       metrics = {
           'train_error': train_loss,
           'test_error': test_loss
       }
       embeddings = {
           "user_id": U,
           "movie_id": V
       }
       return CFModel(embeddings, train_loss, [metrics])
```

[18] Solution **(hidden solution)**

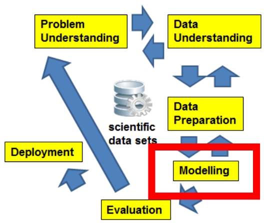
(embedding dimension 30 parameter)

```
[19] # Build the CF model and train it.
     model = build_model(ratings, embedding_dim=30, init_stddev=0.5)
     model.train(num_iterations=1000, learning_rate=10.)
```

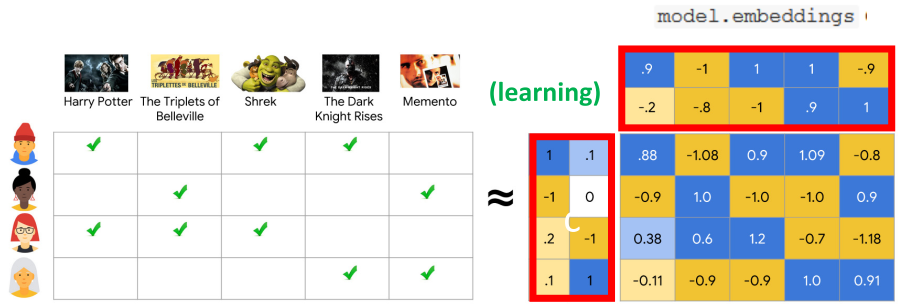
```
iteration 1000: train_error=0.372467, test_error=1.334507[{'test_error': 1.334507, 'train_error': 0.37246665}]
```



- **Latent features are learned, but are hard to explain**
- **Latent features are known as 'hidden features' to distinguish them from observed features**
- **Latent features are computed from observed features using matrix factorization techniques**

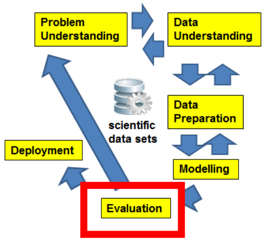


[1] Google Colab Exercise



Google Colab – Movie Rental Recommendation Notebook – Model Evaluation

- Evaluation Viewpoints
  - Movie recommendation
  - Nearest neighbors of some movies
  - Norms of the movie embeddings
  - (Visualizing the embedding in a projected embedding space)



(how good is the model really?)

- Computes the scores of the candidates
  - Different similarity measures will yield different results

```
[20] DOT = 'dot'
     COSINE = 'cosine'
     def compute_scores(query_embedding, item_embeddings, measure=DOT):
       """Computes the scores of the candidates given a query.
       Args:
         query_embedding: a vector of shape [k], representing the query embedding.
         item_embeddings: a matrix of shape [N, k], such that row i is the embedding
           of item i.
         measure: a string specifying the similarity measure to be used. Can be
           either DOT or COSINE.
       Returns:
         scores: a vector of shape [N], such that scores[i] is the score of item i.
       """
       # ======================== Complete this section ============================
       # scores =
       # ===========================================================================
       return scores
```

- dot product: the score of item j is $\langle u, V_j \rangle$.
- cosine: the score of item j is $\frac{\langle u, V_j \rangle}{\|u\|\|V_j\|}$.

(hidden solution)

[21] Solution

```
[22] # @title User recommendations and nearest neighbors (run this cell)
def user_recommendations(model, measure=DOT, exclude_rated=False, k=6):
  if USER_RATINGS:
    scores = compute_scores(
        model.embeddings["user_id"][943], model.embeddings["movie_id"], measure)
    score_key = measure + ' score'
    df = pd.DataFrame({
        score_key: list(scores),
        'movie_id': movies['movie_id'],
        'titles': movies['title'],
        'genres': movies['all_genres'],
    })
    if exclude_rated:
      # remove movies that are already rated
      rated_movies = ratings[ratings.user_id == "943"]["movie_id"].values
      df = df[df.movie_id.apply(lambda movie_id: movie_id not in rated_movies)]
    display.display(df.sort_values([score_key], ascending=False).head(k))

def movie_neighbors(model, title_substring, measure=DOT, k=6):
  # Search for movie ids that match the given substring.
  ids =  movies[movies['title'].str.contains(title_substring)].index.values
  titles = movies.iloc[ids]['title'].values
  if len(titles) == 0:
    raise ValueError("Found no movies with title %s" % title_substring)
  print("Nearest neighbors of : %s." % titles[0])
  if len(titles) > 1:
    print("[Found more than one matching movie. Other candidates: {}]".format(
        ", ".join(titles[1:])))
  movie_id = ids[0]
  scores = compute_scores(
      model.embeddings["movie_id"][movie_id], model.embeddings["movie_id"],
      measure)
  score_key = measure + ' score'
  df = pd.DataFrame({
      score_key: list(scores),
      'titles': movies['title'],
      'genres': movies['all_genres']
  })
  display.display(df.sort_values([score_key], ascending=False).head(k))
```

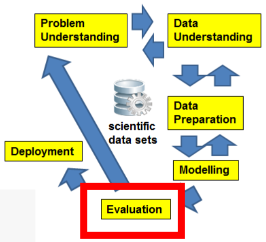
[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Model Evaluation

- Validating recommendations
  - Using different score measures

```
[23] user_recommendations(model, measure=COSINE, k=5)
```

```
[24] movie_neighbors(model, "Aladdin", DOT)
     movie_neighbors(model, "Aladdin", COSINE)
```



(how good is the model really?)

```
Nearest neighbors of : Aladdin (1992)
[Found more than one matching movie. Other candidates: Aladdin and the King of Thieves (1996)]
```

	dot score	titles	genres
94	6.331	Aladdin (1992)	Animation-Children-Comedy-Musical
21	5.593	Braveheart (1995)	Action-Drama-War
142	5.513	Sound of Music, The (1965)	Musical
1159	5.454	Love! Valour! Compassion! (1997)	Drama-Romance
312	5.378	Titanic (1997)	Action-Drama-Romance
95	5.360	Terminator 2: Judgment Day (1991)	Action-Sci-Fi-Thriller

(manual validation, would you assume this to be right?)

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of Thieves (1996)]
```

	cosine score	titles	genres
94	1.000	Aladdin (1992)	Animation-Children-Comedy-Musical
0	0.836	Toy Story (1995)	Animation-Children-Comedy
221	0.832	Star Trek: First Contact (1996)	Action-Adventure-Sci-Fi
81	0.831	Jurassic Park (1993)	Action-Adventure-Sci-Fi
587	0.822	Beauty and the Beast (1991)	Animation-Children-Musical
70	0.819	Lion King, The (1994)	Animation-Children-Musical

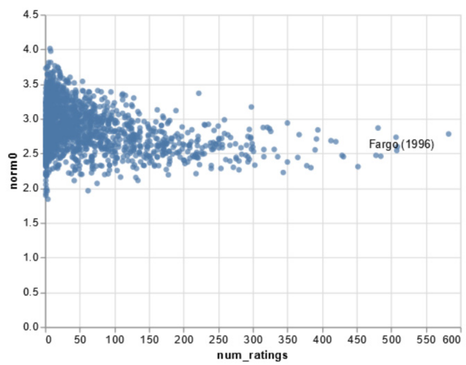
- **Using the dot-product score for model evaluation in training a matrix factorization model the model tends to recommend popular movies**
- **Popular movies are explained by the fact that in matrix factorization models, the norm of the embedding is often correlated with popularity**
- **Popular movies have a larger norm that makes the model more likely to recommend more popular movies**

[25] Embedding Visualization code (run this cell)

```
[26] movie_embedding_norm(model)
```

(confirm this by sorting the movies by their embedding norm)

expected norm of a d-dimensional vector with entries $\sim \mathcal{N}(0, \sigma^2)$ is approximatley $\sigma\sqrt{d}$.



[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Model Tuning

- Working on Hyperparameters
  - Change initial standard deviation hyperparameter init_stddev
  - How does this affect the embedding norm distribution, and the ranking of the top-norm movies?

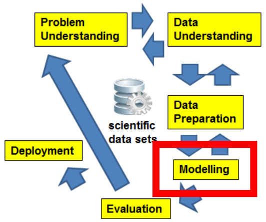
```
[27] #@title Solution
     model_lowinit = build_model(ratings, embedding_dim=30, init_stddev=0.05)
     model_lowinit.train(num_iterations=1000, learning_rate=10.)
     movie_neighbors(model_lowinit, "Aladdin", DOT)
     movie_neighbors(model_lowinit, "Aladdin", COSINE)
     movie_embedding_norm([model, model_lowinit])
```

```
iteration 1000: train_error=0.356582, test_error=0.980989Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of Thieves (1996)]
```

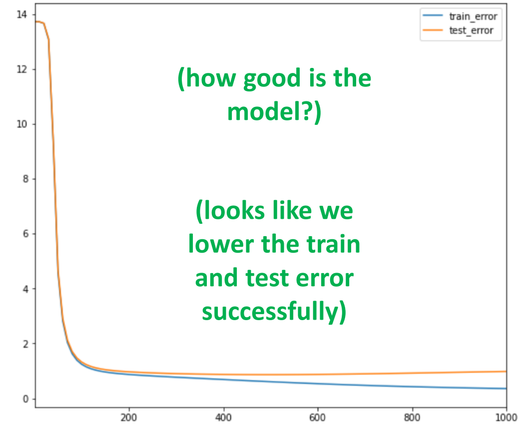
	dot score	titles	genres
94	5.792	Aladdin (1992)	Animation-Children-Comedy-Musical
180	5.119	Return of the Jedi (1983)	Action-Adventure-Romance-Sci-Fi-War
49	4.964	Star Wars (1977)	Action-Adventure-Romance-Sci-Fi-War
171	4.921	Empire Strikes Back, The (1980)	Action-Adventure-Drama-Romance-Sci-Fi-War
63	4.858	Shawshank Redemption, The (1994)	Drama
173	4.805	Raiders of the Lost Ark (1981)	Action-Adventure

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of Thieves (1996)]
```

	cosine score	titles	genres
94	1.000	Aladdin (1992)	Animation-Children-Comedy-Musical
1189	0.839	That Old Feeling (1997)	Comedy-Romance
1145	0.830	Calendar Girl (1993)	Drama
1114	0.830	Twelfth Night (1996)	Comedy-Drama-Romance
266	0.822	unknown	genre_unknown
365	0.820	Dangerous Minds (1995)	Drama



(new model training after tuning hyperparameter)

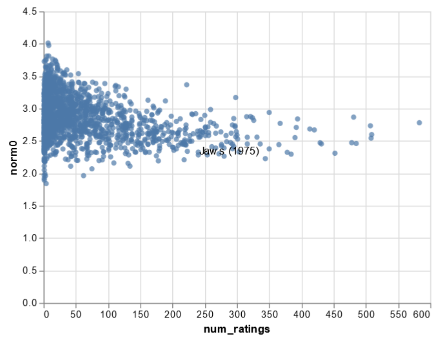


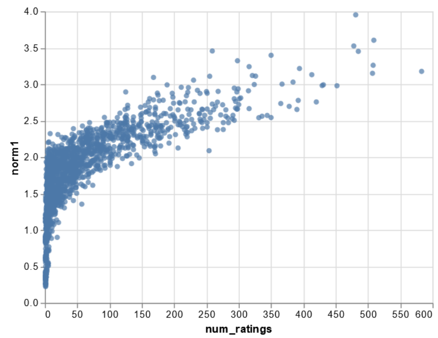
(how good is the model?)

(looks like we lower the train and test error successfully)

(manual validation, changes in recommendation observed)

(remember much more record counts for drama in movies than others → bias)

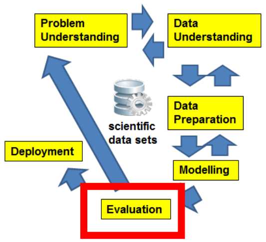




[1] Google Colab Exercise

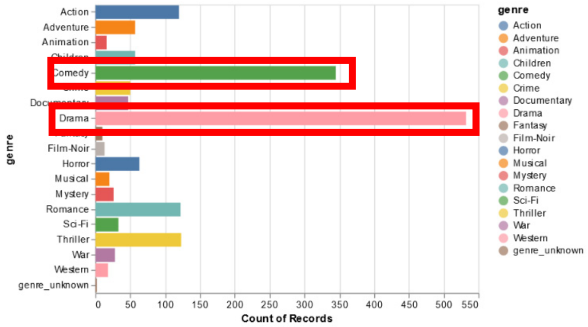
Google Colab – Movie Rental Recommendation Notebook – Model ‘Visualization‘

- Evaluation Viewpoints
  - Movie recommendation
  - Nearest neighbors of some movies
  - Norms of the movie embeddings
  - Visualizing the embedding in a projected embedding space
  - Example t-SNE approach



(how good is the model really?)

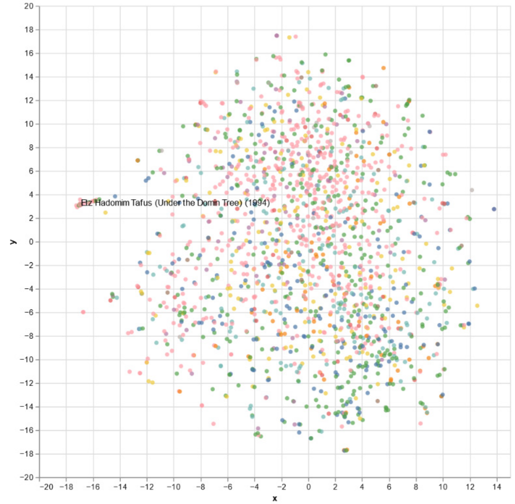
(observed pairs with bias)



(not good model because we trained only on the observed pairs and using no regularization method, cf. Practical Lecture 3.1 & 7.1)

(idea: genre still somewhat close in the embedding space)

```
[28] tsne_movie_embeddings(model_lowinit)

Running t-SNE...
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 1682 samples in 0.000s...
[t-SNE] Computed neighbors for 1682 samples in 0.105s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1682
[t-SNE] Computed conditional probabilities for sample 1682 / 1682
[t-SNE] Mean sigma: 0.117465
[t-SNE] KL divergence after 250 iterations with early exaggeration: 57.780632
[t-SNE] KL divergence after 400 iterations: 2.281902
```



- **It is hard to visualize model embeddings in a higher-dimensional space (>3, here in this example the embedding dimension is 30) so one idea is to project the embeddings to a lower dimensional space**
- **t-distributed Stochastic Neighbor Embedding (t-SNE) is an algorithm that projects the embeddings while attempting to preserve their pairwise distances**
- **t-SNE is used for visualization of models but should be handled with care (embeddings hard to visualize correctly)**

(poor quality model identified by interactive visualization: embeddings have not any notable structure: embeddings of genre are located all over the embedding space)

[26] t-SNE information

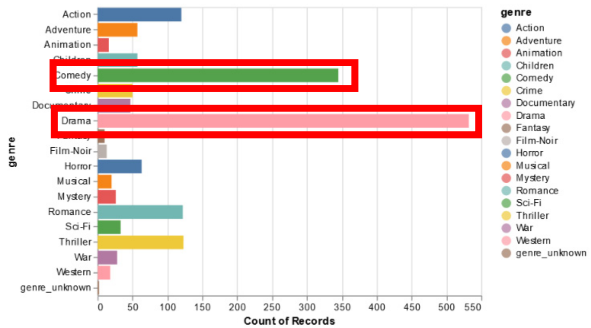
[1] Google Colab Exercise

Google Colab – Movie Rental Recommendation Notebook – Model Regularization

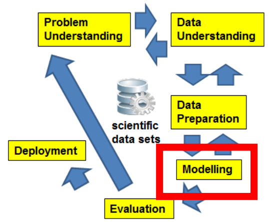
- Learned insights from Evaluation
  - Poor model quality

> - **In learning a recommendation engine with matrix multiplication a potential poor model quality can occur when learning only on the observed part of the rating matrix and not using regularization**
> - **The reason for poor model quality in this case is known as ‘folding’: the model does not learn how to place the embeddings of irrelevant movies**

(observed pairs with bias)




```
[29] def gravity(U, V):
       """Creates a gravity loss given two embedding matrices."""
       return 1. / (U.shape[0].value*V.shape[0].value) * tf.reduce_sum(
           tf.matmul(U, U, transpose_a=True) * tf.matmul(V, V, transpose_a=True))

     def build_regularized_model(
         ratings, embedding_dim=3, regularization_coeff=.1, gravity_coeff=1.,
         init_stddev=0.1):
       """
       Args:
         ratings: the DataFrame of movie ratings.
         embedding_dim: The dimension of the embedding space.
         regularization_coeff: The regularization coefficient lambda.
         gravity_coeff: The gravity regularization coefficient lambda_g.
       Returns:
         A CFModel object that uses a regularized loss.
       """
       # Split the ratings DataFrame into train and test.
       train_ratings, test_ratings = split_dataframe(ratings)
       # SparseTensor representation of the train and test datasets.
       A_train = build_rating_sparse_tensor(train_ratings)
       A_test = build_rating_sparse_tensor(test_ratings)
       U = tf.Variable(tf.random_normal(
           [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
       V = tf.Variable(tf.random_normal(
           [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))

       # ======================== Complete this section ========================
       # error_train =
       # error_test =
       # gravity_loss =
       # regularization_loss =
       # =====================================================================
       total_loss = error_train + regulariza n_loss + gravity_loss
       losses = {
           'train_error': error_train,
           'test_error': error_test,
       }
       loss_components = {
           'observed_loss': error_train,
           'regularization_loss': regulariza n_loss,
           'gravity_loss': gravity_loss,
       }
       embeddings = {"user_id": U, "movie_i V}

       return CFModel(embeddings, total_loss losses, loss_components])
```



- Regularization for Matrix Factorization
  - Regularization often inherent in model building (e.g., logistic regression, or support vector machines)
  - Add two types of regularization terms that will address this issue:

- Regularization of the model parameters. This is a common ℓ_2 regularization term on the embedding matrices, given by $r(U,V) = \frac{1}{N}\sum_i \|U_i\|^2 + \frac{1}{M}\sum_j \|V_j\|^2$.
- A global prior that pushes the prediction of any pair towards zero, called the *gravity* term. This is given by $g(U,V) = \frac{1}{MN}\sum_{i=1}^N \sum_{j=1}^M \langle U_i, V_j\rangle^2$.

(total loss with two new hyper-parameters for tuning: the amount of regularization)

$$\frac{1}{|\Omega|}\sum_{(i,j)\in\Omega}(A_{ij} - \langle U_i, V_j\rangle)^2 + \lambda_r r(U,V) + \lambda_g g(U,V)$$

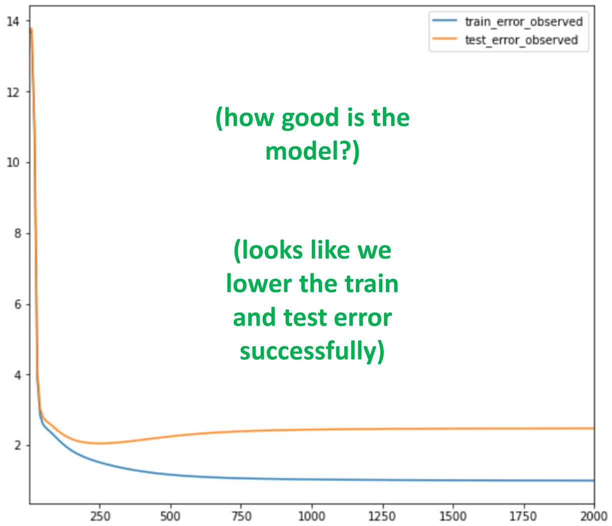
where λ_r and λ_g are two regularization coefficients (hyper-parameters).

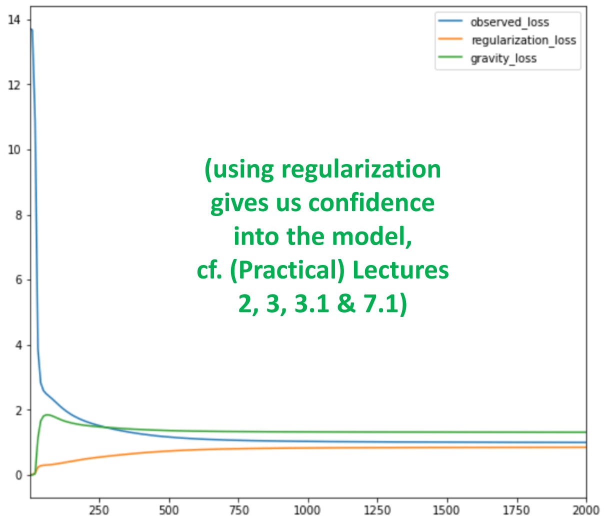
[30] Solution

(hidden solution)

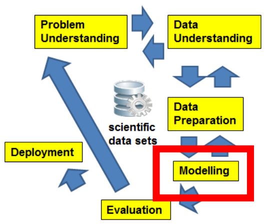
Google Colab – Movie Rental Recommendation Notebook – Model Tuning Again

- Hyper-Parameter Tuning
  - Most complex aspect in machine learning & data mining
  - Takes massive human time with a lot of possibilities to choose from
  - E.g. regularization_coeff
  - E.g. gravity_coeff
  - E.g. embedding_dim
  - E.g. init_stddev
  - E.g. num_iterations (fitting over time)
  - E.g. learning_rate (relevant for SGD)






```
[31] reg_model = build_regularized_model(
         ratings, regularization_coeff=0.1, gravity_coeff=1.0, embedding_dim=35,
         init_stddev=.05)
     reg_model.train(num_iterations=2000, learning_rate=20.)
```

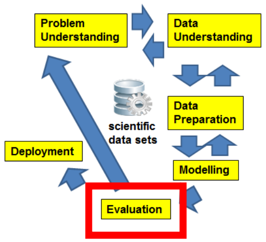


- **Several techniques have been established to help with a more systematic hyper-parameter tuning, like AutoML techniques or genetic algorithms for example**
- **Still many modeling activities require human intervention to really tune a machine learning or data mining model really right so that it generalizes well**

```
iteration 2000: train_error_observed=1.000804, test_error_observed=2.475982, observed_loss=1.000804, regularization_loss=0.852243, gravity_loss=1.313947[{'test_error_observed': 2.4759824, 'train_error_observed': 1.0008036},
{'gravity_loss': 1.3139468,
 'observed_loss': 1.0008036,
 'regularization_loss': 0.8522429}]
```

Google Colab – Movie Rental Recommendation Notebook – Final Model?

- Back to Model Evaluation
  - After Regularization and using different learning scheme
  - E.g. dot-product, cosine, norms




```
[32] user_recommendations(reg_model, DOT, exclude_rated=True, k=10)
```

```
[33] movie_neighbors(reg_model, "Aladdin", DOT)
     movie_neighbors(reg_model, "Aladdin", COSINE)
```

(exclude_rated: own rated ones via spread sheet)

```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of Thieves (1996)]
```

	dot score	titles	genres
94	8.856	Aladdin (1992)	Animation-Children-Comedy-Musical
70	7.973	Lion King, The (1994)	Animation-Children-Musical
587	7.563	Beauty and the Beast (1991)	Animation-Children-Musical
317	7.519	Schindler's List (1993)	Drama-War
171	7.499	Empire Strikes Back, The (1980)	Action-Adventure-Drama-Romance-Sci-Fi-War
173	7.443	Raiders of the Lost Ark (1981)	Action-Adventure

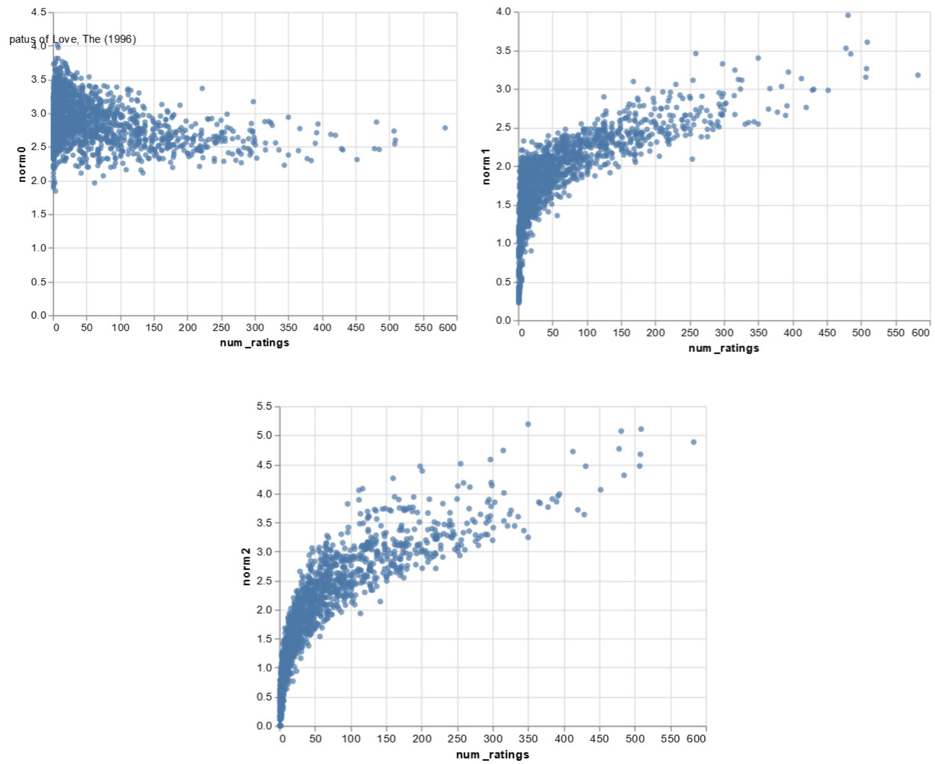
```
Nearest neighbors of : Aladdin (1992).
[Found more than one matching movie. Other candidates: Aladdin and the King of Thieves (1996)]
```

	cosine score	titles	genres
94	1.000	Aladdin (1992)	Animation-Children-Comedy-Musical
70	0.908	Lion King, The (1994)	Animation-Children-Musical
587	0.845	Beauty and the Beast (1991)	Animation-Children-Musical
98	0.774	Snow White and the Seven Dwarfs (1937)	Animation-Children-Musical
201	0.772	Groundhog Day (1993)	Comedy-Romance
81	0.762	Jurassic Park (1993)	Action-Adventure-Sci-Fi

(we seem to improve the modeling)

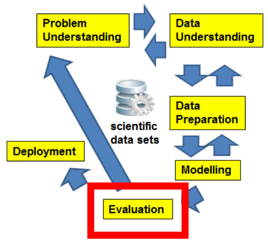
```
[34] movie_embedding_norm([model, model_lowinit, reg_model])
```

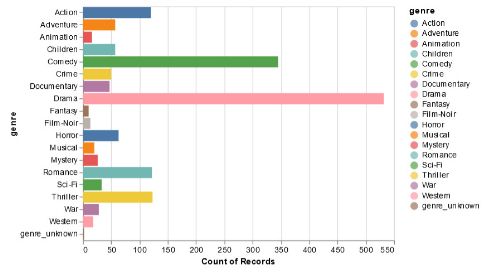
(comparing norms between model & new regularized model)

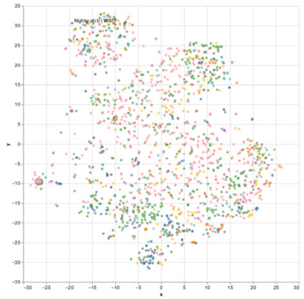


Google Colab – Movie Rental Recommendation Notebook – Final Model!

- Continue Model Evaluation
  - Visualizing the embedding in a projected embedding space
  - Example t-SNE approach

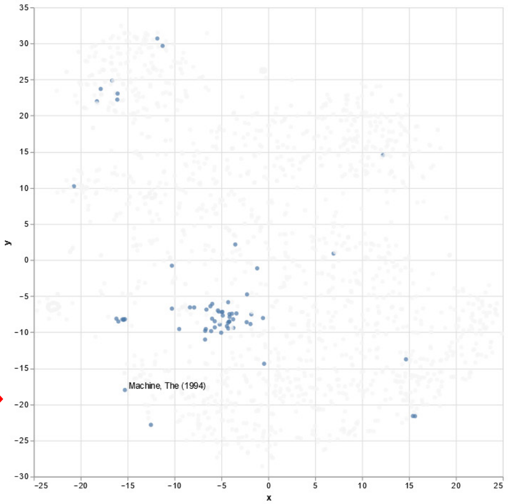


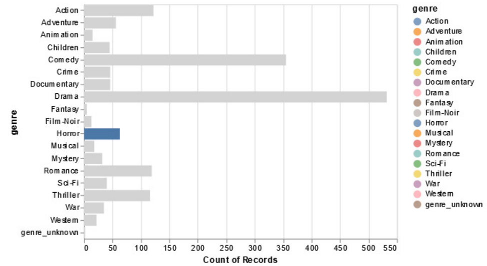




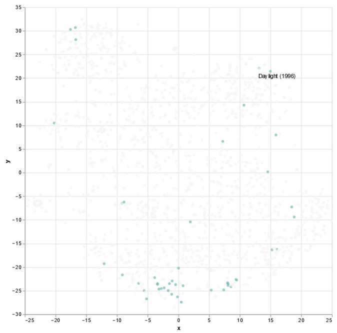
```
[35] # Visualize the embeddings
     tsne_movie_embeddings(reg_model)
```

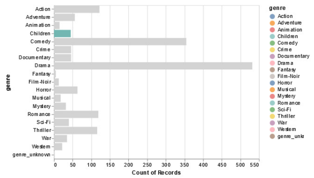
```
Running t-SNE...
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 1682 samples in 0.003s...
[t-SNE] Computed neighbors for 1682 samples in 0.100s...
[t-SNE] Computed conditional probabilities for sample 1000 / 1682
[t-SNE] Computed conditional probabilities for sample 1682 / 1682
[t-SNE] Mean sigma: 0.251028
[t-SNE] KL divergence after 250 iterations with early exaggeration: 58.387978
[t-SNE] KL divergence after 400 iterations: 1.483514
```



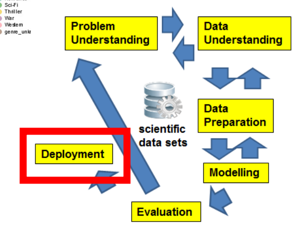


- **The final model evaluation after regularization reveals that the embeddings have a lot more structure than the unregularized model case**
- **Examples include different genres where one can observe how they tend to form clusters (e.g., Horror, Animation and Children), but 'never perfect'**
- **More specialized algorithms such as Alternating Least Squares (ALS) might improve the modeling**

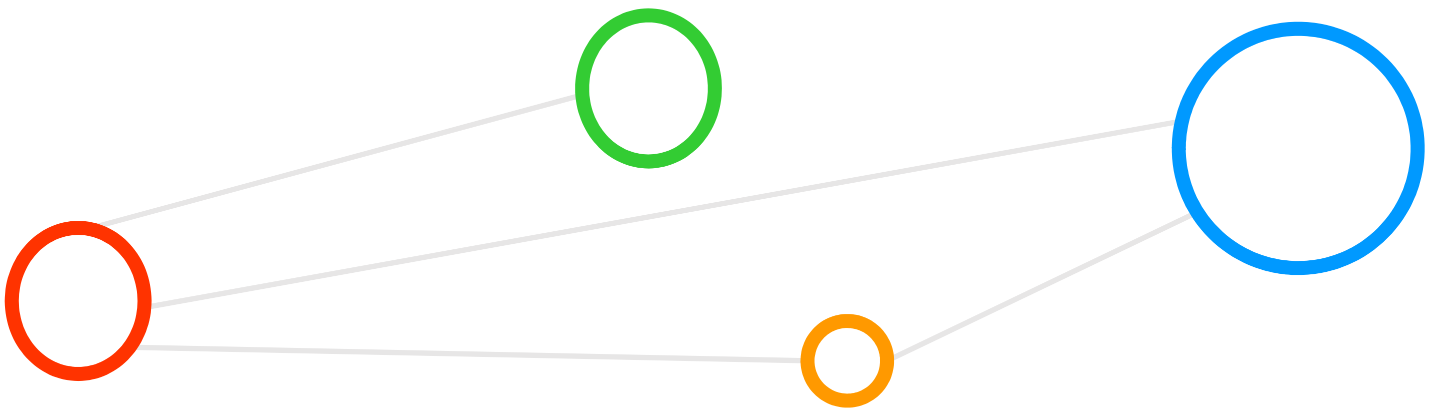




(final step: deploy it in a real movie recommendation environment with unseen data & new movies/users)



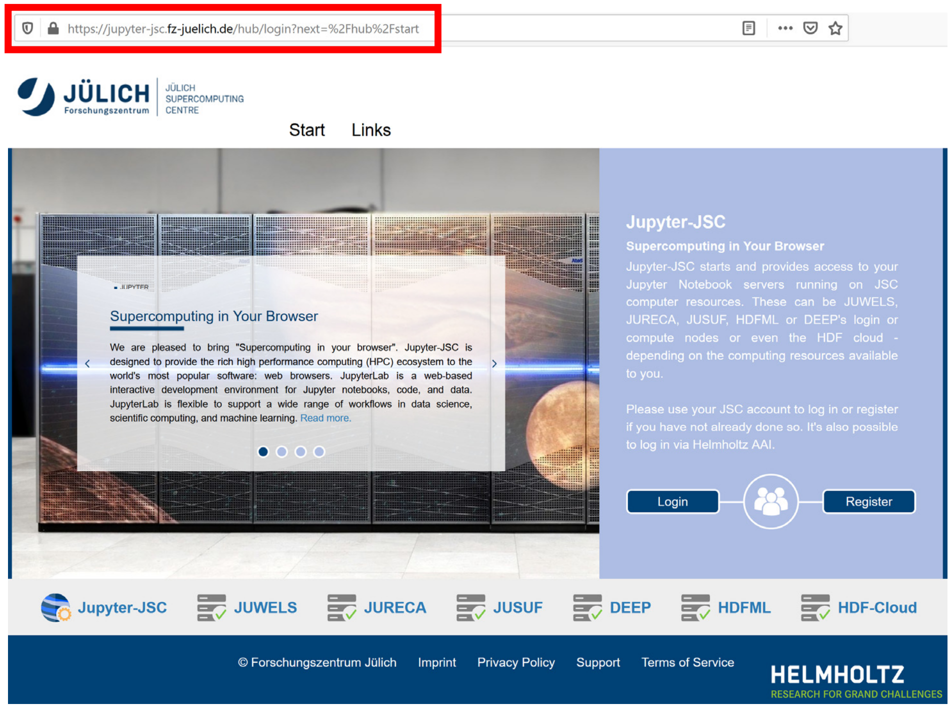
Appendix – Data Mining – Association Rule Mining

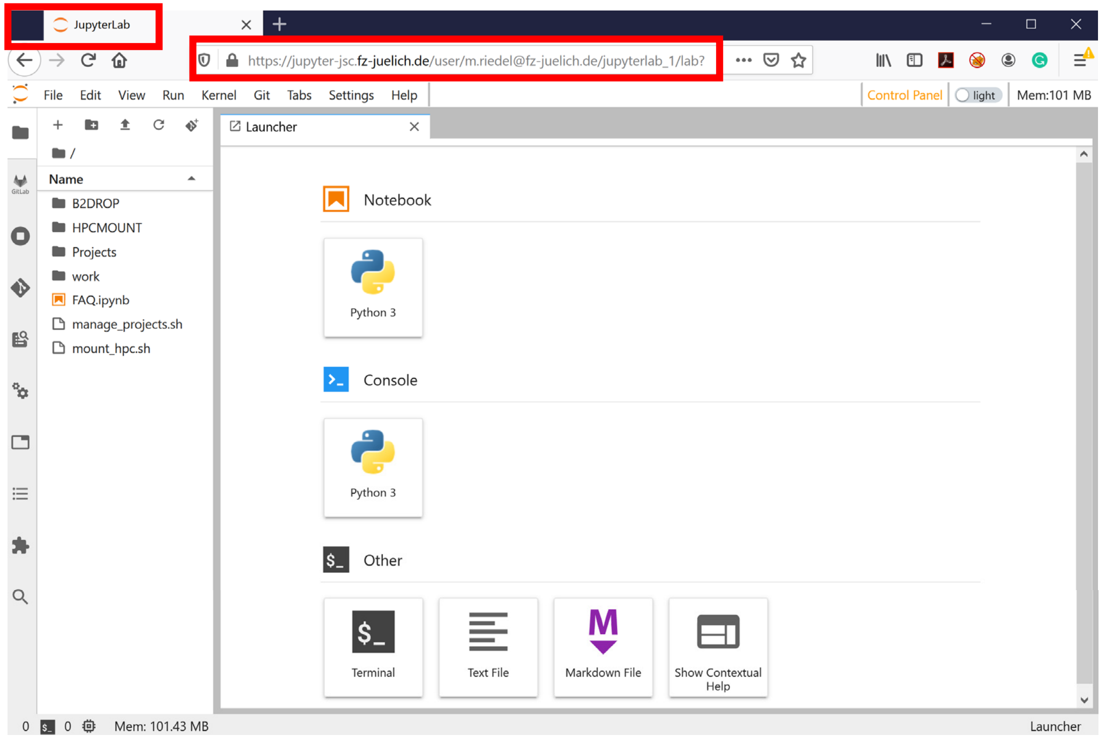


Remote Access to HPC Systems: Jupyter @ Juelich Supercomputing Centre (JSC)

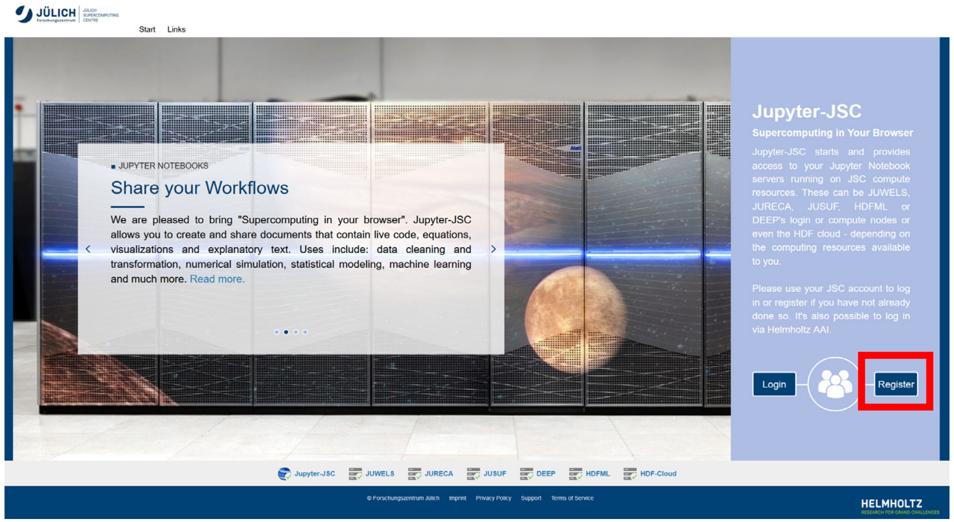
- Startup Remote Jupyter (Jupyter @ JSC)
  - Understanding differences between local laptop vs. remote cloud or HPC system
  - Understanding differences Jupyter vs. JupyterLab

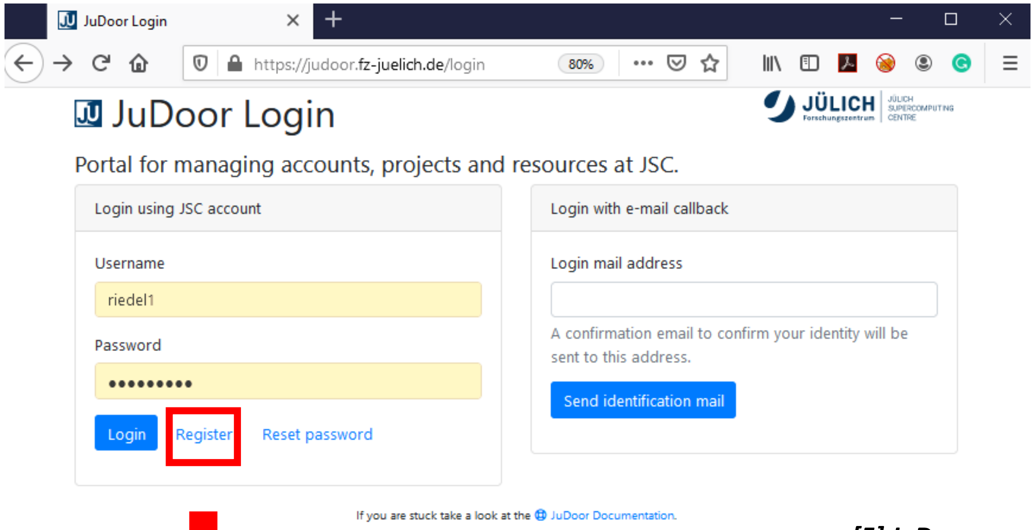
[2] Jupyter *[3] Jupyter @ JSC*





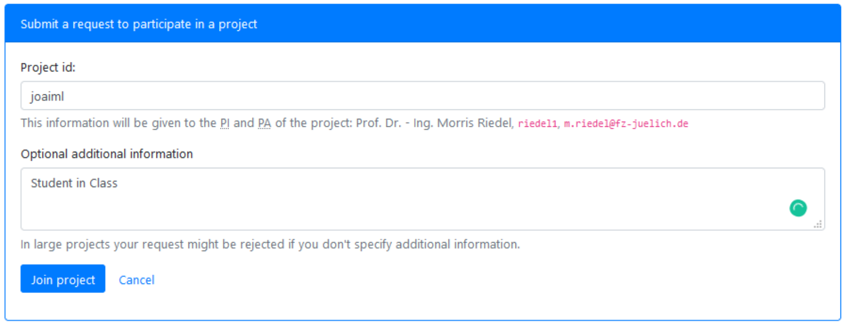
Jupyter @ JSC – Register & Access

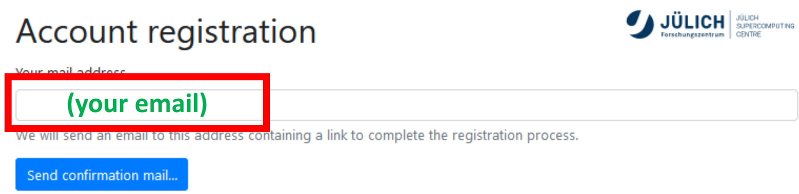




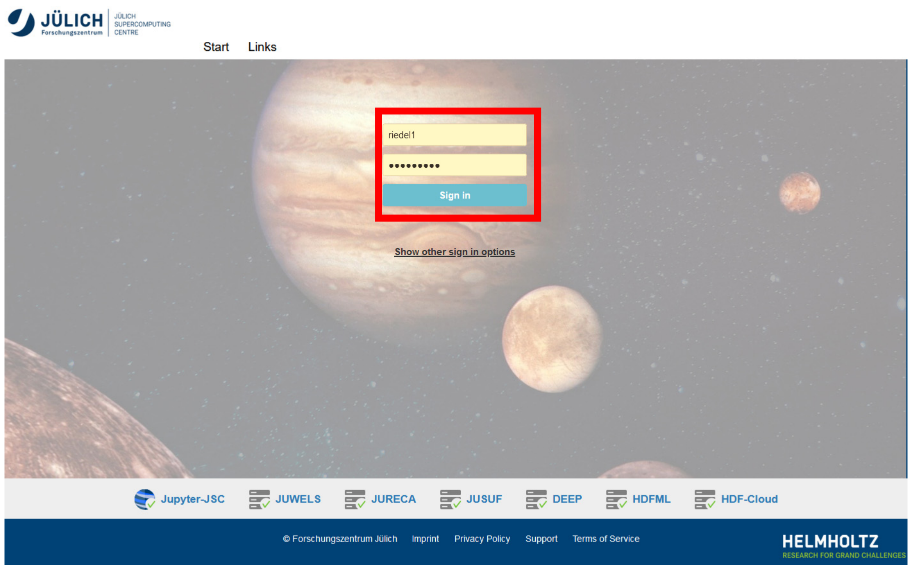
[5] JuDoor

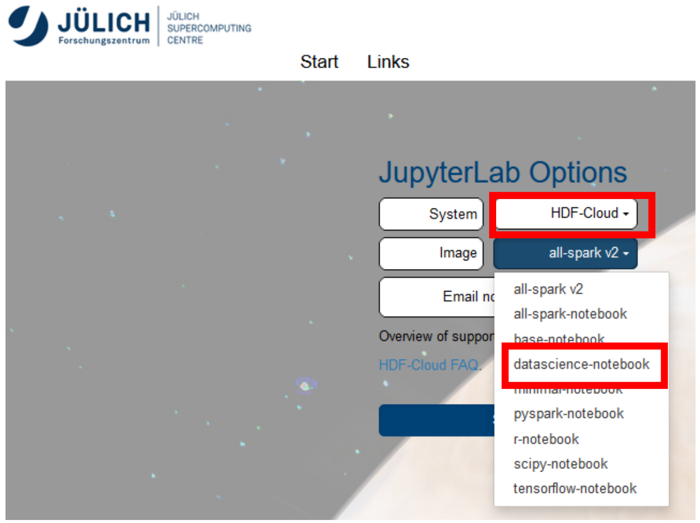
(after login press 'join project' and fill out the information as below)





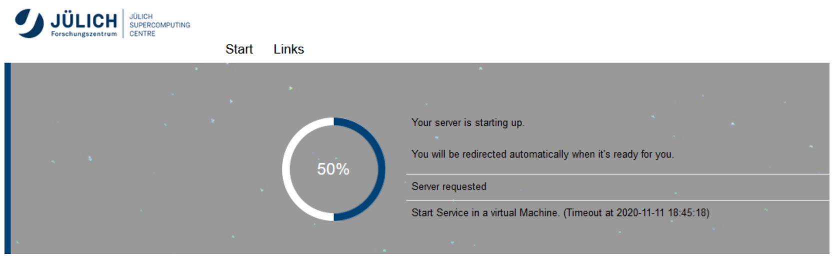
Jupyter @ JSC – Access via JuDoor Account & Use HDF – Cloud

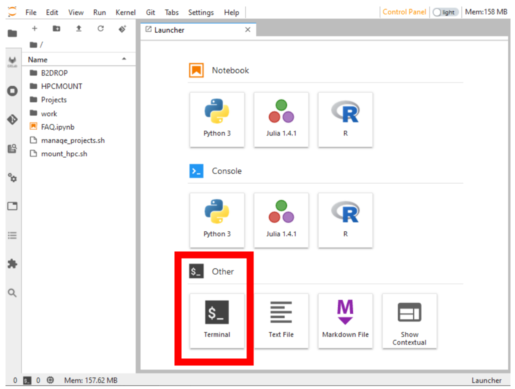




- Helmholtz Data Federation (HDF) Cloud Computing Platform @ JSC *[6] HDF-Cloud*
  - Comprises OpenStack compute, network, and volume services as well as an integration with the DATA file system also available on the HPC systems
  - Includes links to other services relevant for the EOSC Cloud (e.g., B2DROP academic dropbox)

Jupyter @ JSC using the HDF-Cloud – Startup & Install MLxtend Library

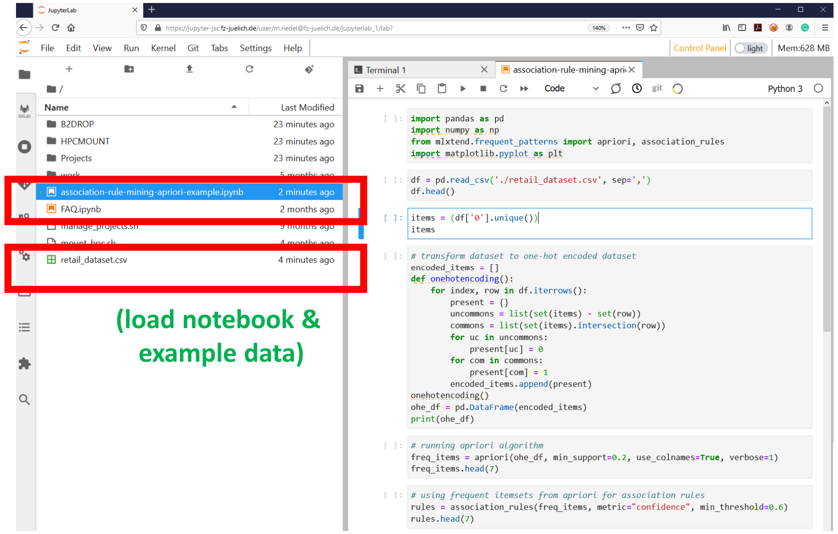




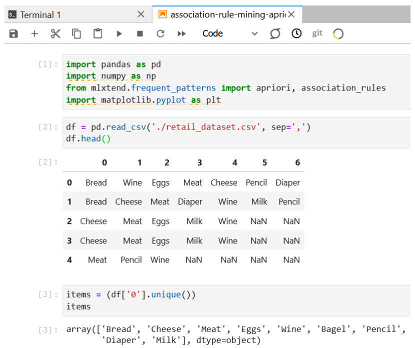
[7] mlxtend lib, Apriori

```
(base) jovyan@005f97a19787:~$ pip install mlxtend
WARNING: Retrying (Retry(tot                     =None, redirect=None, status=None)) after connection broken by 'ReadTimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed out. (read timeout=15)
")': /simple/mlxtend/
WARNING: Retrying (Retry(total=3, connect=None, read=None, redirect=None, status=None)) after connection broken by 'ReadTimeoutError("HTTPSConnectionPool(host='pypi.org', port=443): Read timed out. (read timeout=15)
")': /simple/mlxtend/
Collecting mlxtend
  WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None, status=None)) after connection broken by 'ReadTimeoutError("HTTPSConnectionPool(host='files.pythonhosted.org', port=443): Read timed out. (
read timeout=15)")': /packages/4c/0d/4a73b8bc49e2cfee178fe50dd8e84d5ba817d0b2454b09308397416e0e48/mlxtend-0.17.3-py2.py3-none-any.whl
  Downloading mlxtend-0.17.3-py2.py3-none-any.whl (1.3 MB)
     |████████████████████████████████| 1.3 MB 6.1 MB/s
Requirement already satisfied: joblib>=0.13.2 in /opt/conda/lib/python3.7/site-packages (from mlxtend) (0.14.1)
Requirement already satisfied: scipy>=1.2.1 in /opt/conda/lib/python3.7/site-packages (from mlxtend) (1.4.1)
Requirement already satisfied: scikit-learn>=0.20.3 in /opt/conda/lib/python3.7/site-packages (from mlxtend) (0.22)
Requirement already satisfied: matplotlib>=3.0.0 in /opt/conda/lib/python3.7/site-packages (from mlxtend) (3.0.3)
Requirement already satisfied: setuptools in /opt/conda/lib/python3.7/site-packages (from mlxtend) (47.3.1)
Requirement already satisfied: numpy>=1.16.2 in /opt/conda/lib/python3.7/site-packages (from mlxtend) (1.19.0)
Requirement already satisfied: pandas>=0.24.2 in /opt/conda/lib/python3.7/site-packages (from mlxtend) (1.0.5)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib>=3.0.0->mlxtend) (2.4.7)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.7/site-packages (from matplotlib>=3.0.0->mlxtend) (0.10.0)
Requirement already satisfied: python-dateutil>=2.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib>=3.0.0->mlxtend) (2.8.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/lib/python3.7/site-packages (from matplotlib>=3.0.0->mlxtend) (1.0.1)
Requirement already satisfied: pytz>=2017.2 in /opt/conda/lib/python3.7/site-packages (from pandas>=0.24.2->mlxtend) (2020.1)
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-packages (from cycler>=0.10->matplotlib>=3.0.0->mlxtend) (1.15.0)
Installing collected packages: mlxtend
Successfully installed mlxtend-0.17.3
(base) jovyan@005f97a19787:~$
```

Using Apriori Algorithm with Retail Shopping Data Example



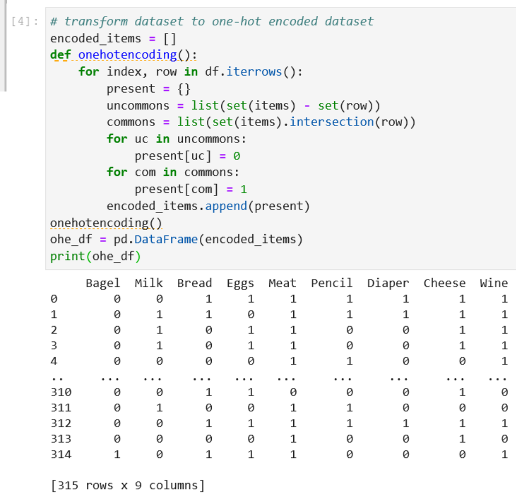
(load notebook & example data)



```
[1]: import pandas as pd
     import numpy as np
     from mlxtend.frequent_patterns import apriori, association_rules
     import matplotlib.pyplot as plt

[2]: df = pd.read_csv('./retail_dataset.csv', sep=',')
     df.head()
```

	0	1	2	3	4	5	6
0	Bread	Wine	Eggs	Meat	Cheese	Pencil	Diaper
1	Bread	Cheese	Meat	Diaper	Wine	Milk	Pencil
2	Cheese	Meat	Eggs	Milk	Wine	NaN	NaN
3	Cheese	Meat	Eggs	Milk	Wine	NaN	NaN
4	Meat	Pencil	Wine	NaN	NaN	NaN	NaN

```
[3]: items = (df['0'].unique())
     items

[3]: array(['Bread', 'Cheese', 'Meat', 'Eggs', 'Wine', 'Bagel', 'Pencil',
            'Diaper', 'Milk'], dtype=object)
```



```
[4]: # transform dataset to one-hot encoded dataset
encoded_items = []
def onehotencoding():
    for index, row in df.iterrows():
        present = {}
        uncommons = list(set(items) - set(row))
        commons = list(set(items).intersection(row))
        for uc in uncommons:
            present[uc] = 0
        for com in commons:
            present[com] = 1
        encoded_items.append(present)
onehotencoding()
ohe_df = pd.DataFrame(encoded_items)
print(ohe_df)
```

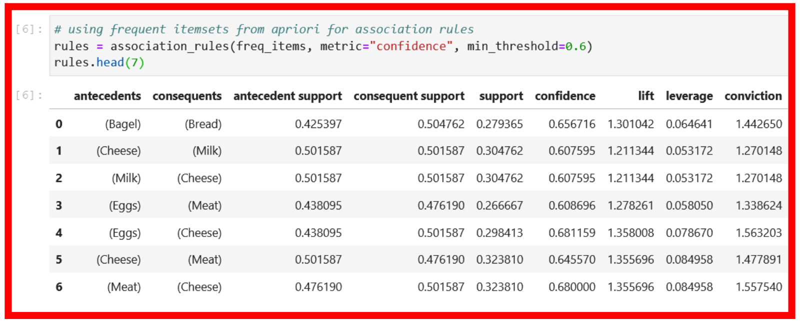
	Bagel	Milk	Bread	Eggs	Meat	Pencil	Diaper	Cheese	Wine
0	0	0	1	1	1	1	1	1	1
1	0	1	1	0	1	1	1	1	1
2	0	1	0	1	1	0	0	1	1
3	0	1	0	1	1	0	0	1	1
4	0	0	0	0	1	1	0	0	1
..	...	...	...	...	...	...	...	...	...
310	0	0	1	1	0	0	0	1	0
311	0	1	0	0	1	1	0	0	0
312	0	0	1	1	1	1	1	1	1
313	0	0	0	0	1	0	0	1	0
314	1	0	1	1	1	0	0	0	1

[315 rows x 9 columns]

(use configuration parameter to finetune the results)

```
[5]: # running apriori algorithm
freq_items = apriori(ohe_df, min_support=0.2, use_colnames=True, verbose=1)
freq_items.head(7)

Processing 141 combinations | Sampling itemset size 3
```

	support	itemsets
0	0.425397	(Bagel)
1	0.501587	(Milk)
2	0.504762	(Bread)
3	0.438095	(Eggs)
4	0.476190	(Meat)
5	0.361905	(Pencil)
6	0.406349	(Diaper)

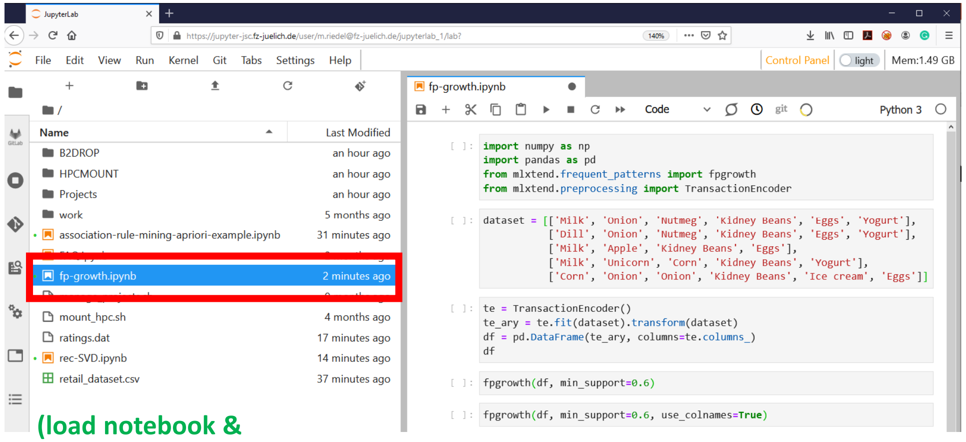


```
[6]: # using frequent itemsets from apriori for association rules
rules = association_rules(freq_items, metric="confidence", min_threshold=0.6)
rules.head(7)
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(Bagel)	(Bread)	0.425397	0.504762	0.279365	0.656716	1.301042	0.064641	1.442650
1	(Cheese)	(Milk)	0.501587	0.501587	0.304762	0.607595	1.211344	0.053172	1.270148
2	(Milk)	(Cheese)	0.501587	0.501587	0.304762	0.607595	1.211344	0.053172	1.270148
3	(Eggs)	(Meat)	0.438095	0.476190	0.266667	0.608696	1.278261	0.058050	1.338624
4	(Eggs)	(Cheese)	0.438095	0.501587	0.298413	0.681159	1.358008	0.078670	1.563203
5	(Cheese)	(Meat)	0.501587	0.476190	0.323810	0.645570	1.355696	0.084958	1.477891
6	(Meat)	(Cheese)	0.476190	0.501587	0.323810	0.680000	1.355696	0.084958	1.557540

[7] mlxtend lib, Apriori

Using FP-Growth Algorithm with Retail Shopping Data Example



(load notebook & no data)

(simple transaction generation)

[1]:
```python
import numpy as np
import pandas as pd
from mlxtend.frequent_patterns import fpgrowth
from mlxtend.preprocessing import TransactionEncoder
```

[2]:
```python
dataset = [['Milk', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
           ['Dill', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
           ['Milk', 'Apple', 'Kidney Beans', 'Eggs'],
           ['Milk', 'Unicorn', 'Corn', 'Kidney Beans', 'Yogurt'],
           ['Corn', 'Onion', 'Onion', 'Kidney Beans', 'Ice cream', 'Eggs']]
```

[3]:
```python
te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)
df
```

[3]:

	Apple	Corn	Dill	Eggs	Ice cream	Kidney Beans	Milk	Nutmeg	Onion	Unicorn	Yogurt
0	False	False	False	True	False	True	True	True	True	False	True
1	False	False	True	True	False	True	False	True	True	False	True
2	True	False	False	True	False	True	True	False	False	False	False
3	False	True	False	False	False	True	True	False	False	True	True
4	False	True	False	True	True	True	False	False	True	False	False

[4]:
```python
fpgrowth(df, min_support=0.6)
```

[4]:

	support	itemsets
0	1.0	(5)
1	0.8	(3)
2	0.6	(10)
3	0.6	(8)
4	0.6	(6)
5	0.8	(3, 5)
6	0.6	(10, 5)
7	0.6	(8, 3)
8	0.6	(8, 5)
9	0.6	(8, 3, 5)
10	0.6	(5, 6)

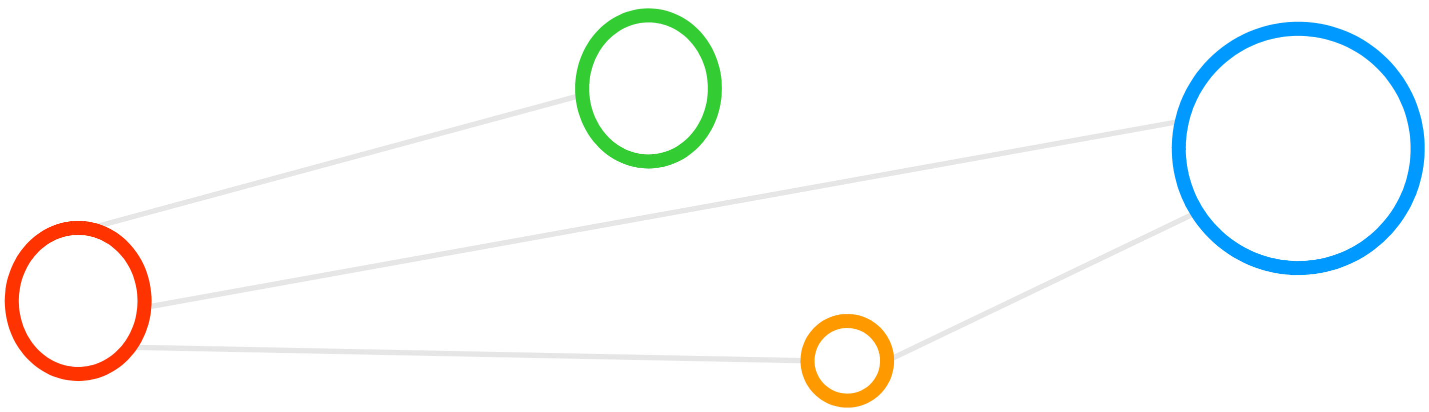
[5]:
```python
fpgrowth(df, min_support=0.6, use_colnames=True)
```

[5]:

	support	itemsets
0	1.0	(Kidney Beans)
1	0.8	(Eggs)
2	0.6	(Yogurt)
3	0.6	(Onion)
4	0.6	(Milk)
5	0.8	(Eggs, Kidney Beans)
6	0.6	(Yogurt, Kidney Beans)
7	0.6	(Onion, Eggs)
8	0.6	(Onion, Kidney Beans)
9	0.6	(Onion, Eggs, Kidney Beans)
10	0.6	(Milk, Kidney Beans)

[7] mlxtend lib, FP-Growth

Lecture Bibliography



Lecture Bibliography (1)

- [1] Google Colab Exercise ‘Build a Movie Recommendation System‘, Online: https://developers.google.com/machine-learning/recommendation/labs/movie-rec-programming-exercise
- [2] Project Jupyter, Online: https://jupyter.org/
- [3] Jupyter @ Juelich Supercomputing Centre, Online: https://jupyter-jsc.fz-juelich.de/
- [4] PRACE Machine Learning Tutorial using JuypterLab & JupyterHub, Online: http://www.morrisriedel.de/prace-tutorial-parallel-and-scalable-machine-learning-introduction
- [5] Juelich Supercomputing Centre Account Setup via JUDOOR, Online: https://judoor.fz-juelich.de/login
- [6] HDF-Cloud, Online: https://www.fz-juelich.de/ias/jsc/EN/Expertise/SciCloudServices/HDFCloud/_node.html
- [7] MLxtend Library, Association Rules & Apriori: http://rasbt.github.io/mlxtend/user_guide/frequent_patterns/association_rules/
- [8] Collaborative Filtering Recommender System on Colab, Online: https://colab.research.google.com/github/SJD1882/Big-Data-Recommender-Systems/blob/master/notebooks/MovieLens27M-ALS-Recommender-System.ipynb
- [9] Google Colaboratory, Online: https://colab.research.google.com
- [10] YouTube Video, ‘Visualization of k-means clustering‘, Online: https://www.youtube.com/watch?v=nXY6PxAaOk0
- [11] MLxtend Library, FP-Growth, Online: http://rasbt.github.io/mlxtend/user_guide/frequent_patterns/fpgrowth/

Lecture Bibliography (2)

- [12] Big Data Tips, Association Rules, Online:
  http://www.big-data.tips/association-rules
- [13] M.S. Mythili et al., 'Performance Evaluation of Apriori & FP-Growth Algorithms', *Int. Journal of Computer Application*, (79), 10, 2013 October 2013, Online:
  https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.1361&rep=rep1&type=pdf
- [14] Pieper.de Duefte, Online:
  https://www.pieper.de/duefte/
- [15] Big Data Tips, Recommender Systems, Online:
  http://www.big-data.tips/recommender-systems
- [16] Various Implementations of Collaborative Filtering, Online:
  https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0
- [17] Collaborative Filtering and Embeddings — Part 1, Online:
  https://towardsdatascience.com/collaborative-filtering-and-embeddings-part-1-63b00b9739ce
- [18] Python scikit-surprise, Online:
  http://surpriselib.com/
- [19] fast.ai library, Online:
  https://www.fast.ai/
- [20] CRISP-DM Model, Online:
  https://www.semanticscholar.org/paper/Crisp-dm%3A-towards-a-standard-process-modell-for-Wirth-Hipp/48b9293cfd4297f855867ca278f7069abc6a9c24
- [21] Altair Visualization Library, Online:
  https://altair-viz.github.io/
- [22] MovieLens Dataset, Online:
  https://grouplens.org/datasets/movielens/

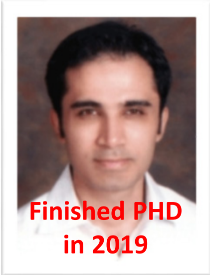
Lecture Bibliography (3)

- [23] gspread Python API for Google Sheets, Online: https://gspread.readthedocs.io/en/latest/
- [24] Project Jupyter, Online: https://jupyter.org/
- [25] Python Programming Language, Online: https://www.python.org/
- [26] How to Use t-SNE Effectively, Online: https://distill.pub/2016/misread-tsne/

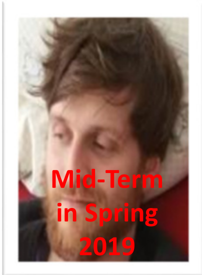
Acknowledgements – High Productivity Data Processing Research Group

PD Dr.
G. Cavallaro

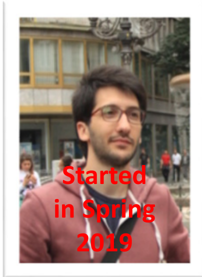
Senior PhD
Student A.S. Memon



Senior PhD
Student M.S. Memon



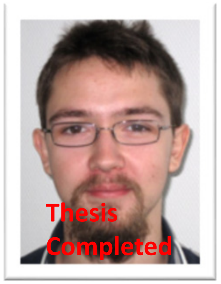
PhD Student
E. Erlingsson



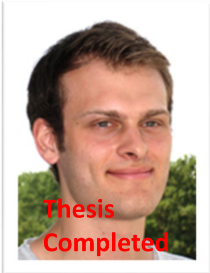
PhD Student
S. Bakarat

PhD Student
R. Sedona

Dr. M. Goetz
(now KIT)

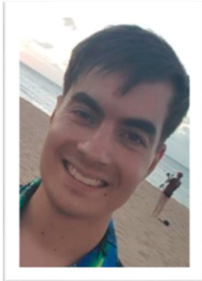


MSc M.
Richerzhagen
(now other division)



MSc
P. Glock
(now INM-1)

MSc
C. Bodenstein
(now
Soccerwatch.tv)



MSc Student
G.S. Guðmundsson
(Landsverkjun)

This research group has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 763558 (DEEP-EST EU Project)


services
measurement
within
fusion
new
funding
Policy-based
concepts
forms
cross-disciplinary
resources
device
Analysis
Centre
climate
Computer
expertise
Approaches
construction
Fusion
year
Computational
Test
E.g
Computing
disciplines
results
Enable
different
like
Cross-Disciplinary
methods
big
PB
community
Environment
Modelling
EU
High
nodes
Key
scientific
important
files
Science
centers
storage
basis
Infrastructure
model
etc
DKRZ
Use
simulations
ENES
technologies
international
formats
physical
IPCC
engineering
References
brain
increasing
EUDAT
via
Big
Project
Data
processing
Common
TB
SMAQ
Methods
MapReduce
create
computing
Using
European
databases
GB
Supercomputing
codes
Work
images
analysis
Large
Summary
data-intensive
Service
Scientific
levels
often
analyze
stored
Resources
DLCL
external
use
Level
Research
JSC
computational
Matters
Concrete
performance
research
e.g
science
HPC
Juelich
datasets
system
management
acquisition
compute
Landscape
System
Climate
modelling
AR5
Hardware
access
hundreds
Nutshell
Services
large-scale
cores
network
DLCLs
Understanding
Technologies
Earth
Structure
distributed
current
structures
Germany
directory
project
General
Simulation
Provide
NASA
Energy
systems
day
Health
Processing