An introduction to

COLLABORATIVE FILTERING IN PYTHON

and an overview of Surprise

WHAT SHOULD I READ?


WHAT SHOULD I
SEE?


WHERE SHOULD I
EAT?

TAXONOMY

- Content-based

  Leverage information about the items content

  Based on item profiles (metadata).
- Collaborative filtering

  Leverage social information (user-item interactions)

  E.g.: recommend items liked by my peers. Usually yields better results.
- Knowledge-based

  Leverage users requirements

  Used for cars, loans, real estate... Very task-specific.

- Collaborative filtering
  Leverage social information (user-item interactions)

  E.g.: recommend items liked by my peers. Usually yields better results.

RATING PREDICTION

$$\begin{pmatrix} ? & 2 & ? & 3 & 1 \\ 1 & 5 & 1 & 4 & ? \\ ? & 4 & ? & ? & ? \\ 2 & 3 & ? & 5 & 1 \\ 2 & ? & 4 & ? & 3 \\ ? & 1 & 4 & 5 & ? \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \text{Charlie} \\ \text{Daniel} \\ \text{Eric} \\ \text{Frank} \end{matrix}$$

Rows are users, columns are items

~ 99% of the entries are missing

Your mission: predict the ?

ABOUT ME

Nicolas Hug
PhD Student (University of Toulouse III)
Needed a Python library for RS research... Did it.

OUTLINE

1. THE NEIGHBORHOOD METHOD (K-NN)

OUTLINE

1. THE NEIGHBORHOOD METHOD (K-NN)
2. OVERVIEW OF SURPRISE

   (http://surpriselib.com)

OUTLINE

1. THE NEIGHBORHOOD METHOD (K-NN)
2. OVERVIEW OF SURPRISE

   (http://surpriselib.com)
3. MATRIX FACTORIZATION

1. THE NEIGHBORHOOD METHOD (K-NN)

NEIGHBORHOOD METHODS

NEIGHBORHOOD METHOD

We have a history of past ratings

We need to predict Alice's rating for Titanic

1. Find the users that have the same tastes as Alice (using the rating history)
2. Average their ratings for Titanic

That's it

1. Find the users that have the same tastes as Alice (using the rating history)

WE NEED A SIMILARITY MEASURE

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

WE NEED A SIMILARITY MEASURE

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

- $\mathbf{sim}(u, v) =$ number of common rated items

WE NEED A SIMILARITY MEASURE

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

- $\mathbf{sim}(u, v) =$ number of common rated items
- $\mathbf{sim}(u, v) =$ average absolute difference between ratings (it's actually a distance)

WE NEED A SIMILARITY MEASURE

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

- $\text{sim}(u, v) =$ number of common rated items
- $\text{sim}(u, v) =$ average absolute difference between ratings (it's actually a distance)
- $\text{sim}(u, v) =$ cosine angle between u and v

WE NEED A SIMILARITY MEASURE

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

- $\text{sim}(u, v) =$ number of common rated items
- $\text{sim}(u, v) =$ average absolute difference between ratings (it's actually a distance)
- $\text{sim}(u, v) =$ cosine angle between u and v
- $\text{sim}(u, v) =$ Pearson correlation coefficient between u and v

WE NEED A SIMILARITY MEASURE

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

- $\mathbf{sim}(u, v) =$ number of common rated items
- $\mathbf{sim}(u, v) =$ average absolute difference between ratings (it's actually a distance)
- $\mathbf{sim}(u, v) =$ cosine angle between u and v
- $\mathbf{sim}(u, v) =$ Pearson correlation coefficient between u and v
- About 3 millions other measures

NEIGHBORHOOD METHOD

We have a history of past ratings

We need to predict Alice's rating for Titanic

1. Find the users that have the same tastes as Alice (using the rating history)
2. Average their ratings for Titanic

That's it

2. Average their ratings for Titanic

AGGREGATING THE NEIGHBORS' RATINGS

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

Alice is close to Bob. Her rating for Titanic is probably close to Bob's (1). So $? \leftarrow 1$

AGGREGATING THE NEIGHBORS' RATINGS

$$\begin{pmatrix} ? & 2 & ? & 2 & 5 & ? & 4 \\ 1 & 1 & ? & 3 & 5 & ? & ? \\ & & & \vdots & & & \\ 4 & 5 & ? & ? & 1 & 4 & 2 \end{pmatrix} \begin{matrix} \text{Alice} \\ \text{Bob} \\ \vdots \\ \text{Zoe} \end{matrix}$$

Alice is close to Bob. Her rating for Titanic is probably close to Bob's (1). So $? \leftarrow 1$

But we should of course look at more than one neighbor!

LET'S CODE!

```
def predict_rating(u, i):
    '''Return estimated rating of user u for item i.
       rating_hist is a list of tuples (user, item, rating)'''

    # Retrieve users having rated i
    neighbors = [(sim[u, v], r_vj)
                 for (v, j, r_vj) in rating_hist if (i == j)]
    # Sort them by similarity with u
    neighbors.sort(key=lambda tple: tple[0], reversed=True)
    # Compute weighted average of the k-NN's ratings
    num = sum(sim_uv * r_vi for (sim_uv, r_vi) in neighbors[:k])
    denum = sum(sim_uv for (sim_uv, _) in neighbors[:k])

    return num / denum
```

LET'S CODE!

```
def predict_rating(u, i):
    '''Return estimated rating of user u for item i.
       rating_hist is a list of tuples (user, item, rating)'''

    # Retrieve users having rated i
    neighbors = [(sim[u, v], r_vj)
                 for (v, j, r_vj) in rating_hist if (i == j)]
    # Sort them by similarity with u
    neighbors.sort(key=lambda tple: tple[0], reversed=True)
    # Compute weighted average of the k-NN's ratings
    num = sum(sim_uv * r_vi for (sim_uv, r_vi) in neighbors[:k])
    denum = sum(sim_uv for (sim_uv, _) in neighbors[:k])

    return num / denum
```

$$\hat{r}_{ui} = \frac{\sum_{v \in k\text{NN}(u)} \text{sim}(u,v) \times r_{vi}}{\sum_{v \in k\text{NN}(u)} \text{sim}(u,v)}$$

NEIGHBORHOOD METHOD

We have a history of past ratings
We need to predict Alice's rating for Titanic

1. Find the users that have the same tastes as Alice (using the rating history)
2. Average their ratings for Titanic

That's it...

NEIGHBORHOOD METHOD

We have a history of past ratings
We need to predict Alice's rating for Titanic

1. Find the users that have the same tastes as Alice (using the rating history)
2. Average their ratings for Titanic

That's it... ?

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation
- Discount similarities (give them more or less confidence)

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation
- Discount similarities (give them more or less confidence)
- Use item-item similarity instead

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation
- Discount similarities (give them more or less confidence)
- Use item-item similarity instead
- Or use both kinds of similarities!

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation
- Discount similarities (give them more or less confidence)
- Use item-item similarity instead
- Or use both kinds of similarities!
- Cluster users and/or items

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation
- Discount similarities (give them more or less confidence)
- Use item-item similarity instead
- Or use both kinds of similarities!
- Cluster users and/or items
- **Learn** the similarities

THERE ARE (APPROXIMATELY) HALF A BILLION VARIANTS

- Normalize the ratings
- Remove bias (some users are mean)
- Use a fancier aggregation
- Discount similarities (give them more or less confidence)
- Use item-item similarity instead
- Or use both kinds of similarities!
- Cluster users and/or items
- **Learn** the similarities
- Blah blah blah...

OUTLINE

1. THE NEIGHBORHOOD METHOD (K-NN)
2. OVERVIEW OF SURPRISE

   (http://surpriselib.com)
3. MATRIX FACTORIZATION

2. OVERVIEW OF SURPRISE

(http://surpriselib.com)

SURPRISE

A Python library for recommender systems

(Or rather: a Python library for rating prediction algorithms)

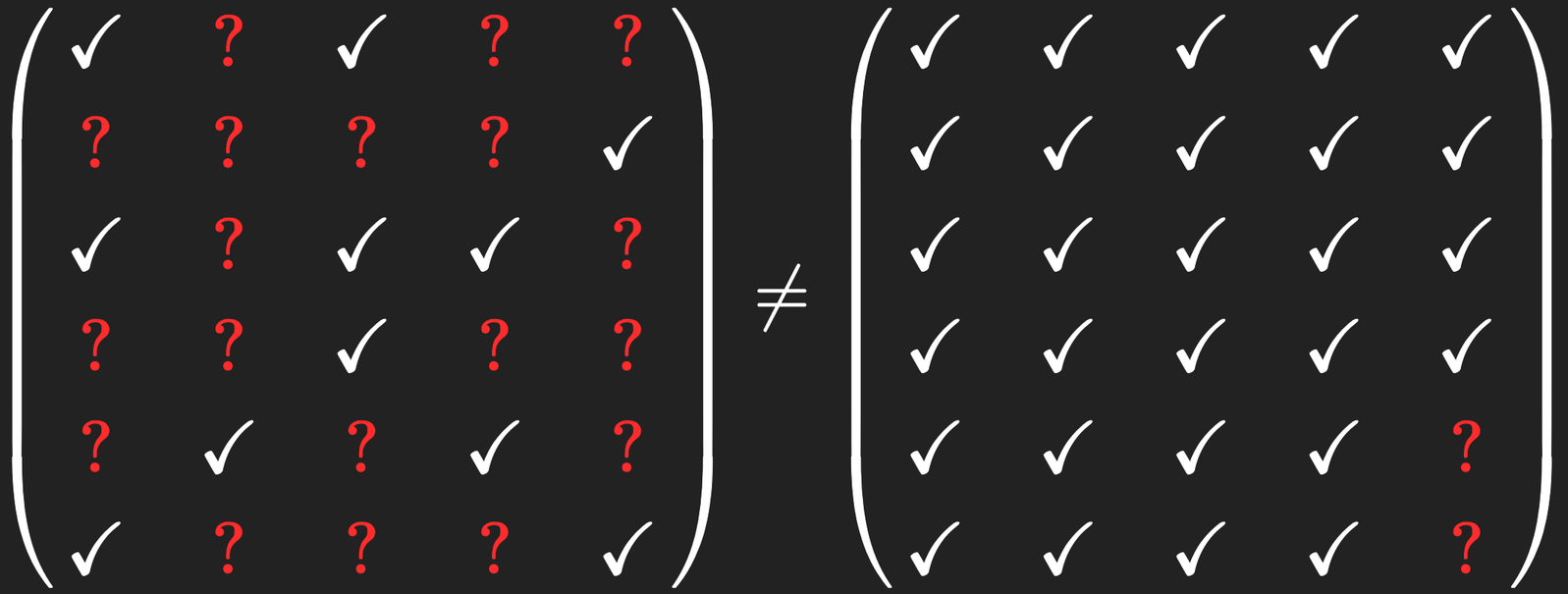
WHY?

- Needed a Python lib for quick and easy prototyping
- Needed to control my experiments

SO WHY NOT SCIKIT-LEARN?

SO WHY NOT SCIKIT-LEARN?

Rating prediction ≠ regression or classification



YES :)

```
clf = MyClassifier()
clf.fit(X_train, y_train)
clf.predict(X_test)
```

NO :(

```
rec = MyRecommender()
rec.fit(X_train, y_train)
rec.predict(X_test)
```

BASIC USAGE

```
from surprise import KNNBasic
from surprise import Dataset

data = Dataset.load_builtin('ml-100k')  # download dataset
trainset = data.build_full_trainset()  # build trainset

algo = KNNBasic()  # use built-in prediction algorithm

algo.train(trainset)  # fit data

algo.predict('Alice', 'Titanic')
algo.predict('Bob', 'Toy Story')
# ...
```

MAIN FEATURES

- Easy dataset handling
- Built-in prediction algorithms (SVD, k-NN, many others)
- Built-in similarity measures (Cosine, Pearson...)
- Custom algorithms are easy to implement
- Cross-validation, grid-search
- Built-in evaluation measures (RMSE, MAE...)
- Model persistence

DATASET HANDLING

```
# Built-in datasets (movielens, Jester)
data = Dataset.load_builtin('ml-100k')

# Custom datasets (from file)
file_path = os.path.expanduser('./my_data')
reader = Reader(line_format='user item rating timestamp',
                sep='\t')
data = Dataset.load_from_file(file_path, reader=reader)

# Custom datasets (from pandas dataframe)
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(df[['uid', 'iid', 'r_ui']],
                            reader)
```

MAIN FEATURES

- Easy dataset handling
- Built-in prediction algorithms (SVD, k-NN, many others)
- Built-in similarity measures (Cosine, Pearson...)
- Custom algorithms are easy to implement
- Cross-validation, grid-search
- Built-in evaluation measures (RMSE, MAE...)
- Model persistence

BUILT-IN ALGORITHMS AND SIMILARITIES

- SVD, SVD++, NMF, k-NN (with variants), Slope One, some baselines...
- Cosine, Pearson, MSD (between users or items)

```
sim_options = {'name': 'cosine',
               'user_based': False,
               'min_support': 10
               }
algo = KNNBasic(sim_options=sim_options)
```

MAIN FEATURES

- Easy dataset handling
- Built-in prediction algorithms (SVD, k-NN, many others)
- Built-in similarity measures (Cosine, Pearson...)
- Custom algorithms are easy to implement
- Cross-validation, grid-search
- Built-in evaluation measures (RMSE, MAE...)
- Model persistence

CUSTOM PREDICTION ALGORITHM

```
class MyRatingPredictor(AlgoBase):

    def estimate(self, u, i):
        return 3


algo = MyRatingPredictor()
algo.train(trainset)
pred = algo.predict('Alice', 'Titanic')  # will call estimate
```

CUSTOM PREDICTION ALGORITHM

```
class MyRatingPredictor(AlgoBase):

    def train(self, trainset):
        '''Fit your data here.'''

        self.mean = np.mean([r_ui for (u, i, r_ui)
                             in trainset.all_ratings()])

    def estimate(self, u, i):
        return self.mean
```

CUSTOM PREDICTION ALGORITHM

```
class MyRatingPredictor(AlgoBase):

    def train(self, trainset):
        '''Fit your data here.'''

        self.mean = np.mean([r_ui for (u, i, r_ui)
                             in trainset.all_ratings()])

    def estimate(self, u, i):
        return self.mean
```

trainset object:

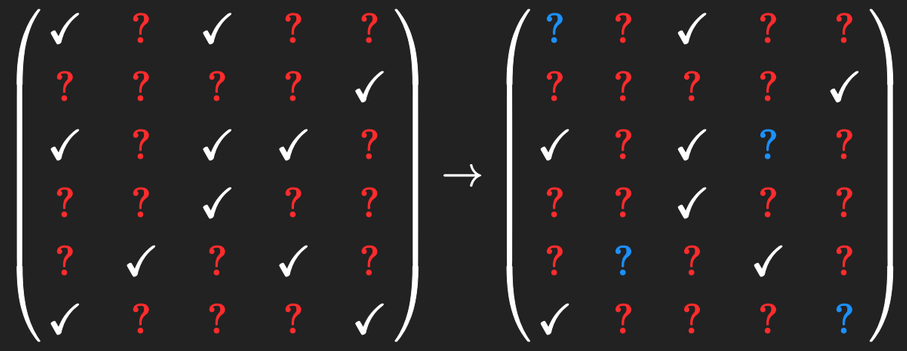
- Iterators over the rating history (user-wise, item-wise, or unordered)
- Iterators over users and items ids
- Other useful methods/attributes

```
class MyKNN(AlgoBase):

    def train(self, trainset):
        self.trainset = trainset
        self.sim = ...  # Compute similarity matrix

    def estimate(self, u, i):

        neighbors = [(self.sim[u, v], r_vi) for (v, r_vj)
                     in self.trainset.ur[u]]
        neighbors = sorted(neighbors,
                           key=lambda tple: tple[0],
                           reverse=True)

        sum_sim = sum_ratings = 0
        for (sim_uv, r_vi) in neighbors[:self.k]:
            sum_sim += sim
            sum_ratings += sim * r

        return sum_ratings / sum_sim
```

MAIN FEATURES

- Easy dataset handling
- Built-in prediction algorithms (SVD, k-NN, many others)
- Built-in similarity measures (Cosine, Pearson...)
- Custom algorithms are easy to implement
- Cross-validation, grid-search
- Built-in evaluation measures (RMSE, MAE...)
- Model persistence

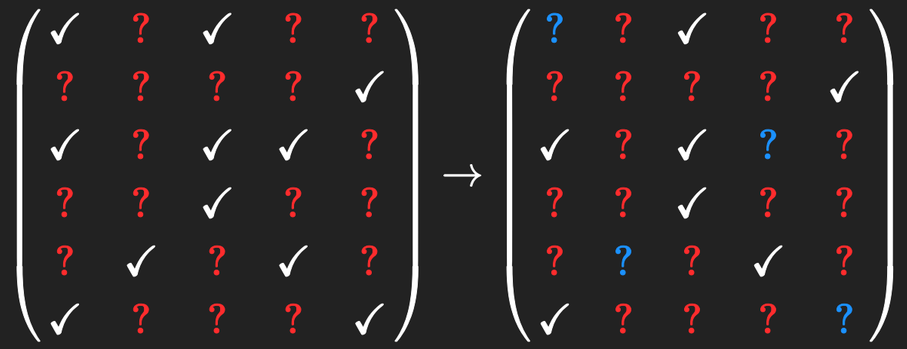
TYPICAL EVALUATION PROTOCOL



Hide some of the ✓ (they become ?)

Use the remaining ✓ to predict the ?

TYPICAL EVALUATION PROTOCOL



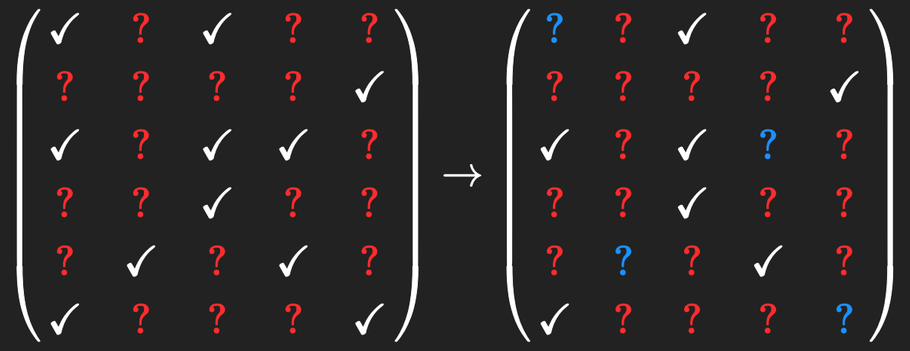
Hide some of the ✓ (they become ?)

Use the remaining ✓ to predict the ?

$$\text{RMSE} = \sqrt{\sum_{?\in R} (r_{ui} - \hat{r}_{ui})^2}$$

The average error, where big errors are heavily penalized (squared)

TYPICAL EVALUATION PROTOCOL



Hide some of the ✓ (they become ?)

Use the remaining ✓ to predict the ?

$$\text{RMSE} = \sqrt{\sum_{?\in R} (r_{ui} - \hat{r}_{ui})^2}$$

The average error, where big errors are heavily penalized (squared)

Do that many times with different subsets: this is cross-validation

CROSS VALIDATION

```
data = Dataset.load_builtin('ml-100k')  # download dataset
data.split(n_folds=3)  # 3-folds cross validation

algo = SVD()

for trainset, testset in data.folds():
    algo.train(trainset)  # fit data
    predictions = algo.test(testset)  # predict ratings
    accuracy.rmse(predictions, verbose=True)  # print RMSE
```

GRID-SEARCH

```
param_grid = {'n_epochs': [5, 10], 'lr_all': [0.002, 0.005],
              'reg_all': [0.4, 0.6]}
grid_search = GridSearch(SVD, param_grid,
                         measures=['RMSE', 'FCP'])

data = Dataset.load_builtin('ml-100k')
data.split(n_folds=3)

grid_search.evaluate(data)  # will evaluate each combination

print(grid_search.best_score['RMSE'])
print(grid_search.best_params['RMSE'])
algo = grid_search.best_estimator['RMSE'])
```

MAIN FEATURES

- Easy dataset handling
- Built-in prediction algorithms (SVD, k-NN, many others)
- Built-in similarity measures (Cosine, Pearson...)
- Custom algorithms are easy to implement
- Cross-validation, grid-search
- Built-in evaluation measures (RMSE, MAE...)
- Model persistence

EVALUATION TOOLS

```
for trainset, testset in data.folds():
    algo.train(trainset)  # fit data
    predictions = algo.test(testset)  # predict ratings

    accuracy.rmse(predictions)
    accuracy.mae(predictions)
    accuracy.fcp(predictions)
```

Can also compute Recall, Precision and top-N related measures easily

MAIN FEATURES

- Easy dataset handling
- Built-in prediction algorithms (SVD, k-NN, many others)
- Built-in similarity measures (Cosine, Pearson...)
- Custom algorithms are easy to implement
- Cross-validation, grid-search
- Built-in evaluation measures (RMSE, MAE...)
- Model persistence

MODEL PERSISTENCE

```
# ... grid search
algo = grid_search.best_estimator['RMSE']

# dump algorithm
file_name = os.path.expanduser('~/dump_file')
dump.dump(file_name, algo=algo)

# ...
# Close Python, go to bed
# ...

# Wake up, reload algorithm, profit
_, algo = dump.load(file_name)
```

Can also dump the predictions to analyze and carefully compare algorithms with pandas

OUTLINE

1. THE NEIGHBORHOOD METHOD (K-NN)
2. OVERVIEW OF SURPRISE

   (http://surpriselib.com)
3. MATRIX FACTORIZATION

3. MATRIX FACTORIZATION

MATRIX FACTORIZATION

MATRIX FACTORIZATION

- A lot of hype during the Netflix Prize (2006-2009: *improve our system, get rich*)
- Model the ratings in an insightful way
- Takes its root in dimensional reduction and SVD

BEFORE SVD: PCA

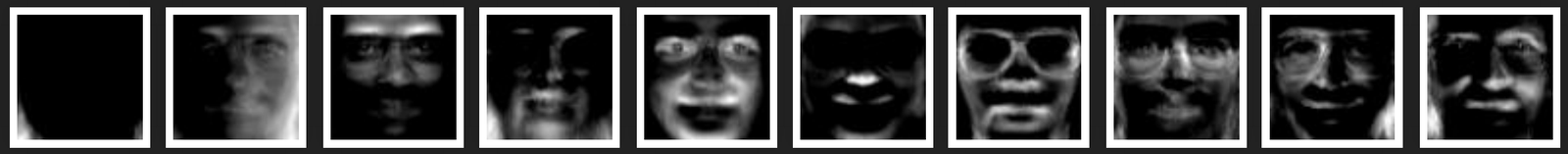
- Here are 400 greyscale images (64 x 64):

- Put them in a 400 x 4096 matrix X:

$$X = \begin{pmatrix} \text{—} & \text{Face 1} & \text{—} \\ \text{—} & \text{Face 2} & \text{—} \\ & \vdots & \\ \text{—} & \text{Face 400} & \text{—} \end{pmatrix}$$

BEFORE SVD: PCA

PCA will *reveal* 400 of those creepy typical guys:



These guys can build back all of the original faces

$$\begin{aligned}\text{Face 1} = &\alpha_1 \cdot \text{Creepy guy } \#1 \\ &+ \alpha_2 \cdot \text{Creepy guy } \#2 \\ &+ \cdots \\ &+ \alpha_{400} \cdot \text{Creepy guy } \#400\end{aligned}$$

PCA also gives you the α_i.

In advance: you don't need all the 400 guys

PCA ON A RATING MATRIX? SURE!

Assume all ratings are known

$$X = \begin{pmatrix} \_\_ & \textbf{Face 1} & \_\_ \\ \_\_ & \textbf{Face 2} & \_\_ \\ & \vdots & \\ \_\_ & \textbf{Face 400} & \_\_ \end{pmatrix} \quad R = \begin{pmatrix} \_\_ & \textbf{Alice} & \_\_ \\ \_\_ & \textbf{Bob} & \_\_ \\ & \vdots & \\ \_\_ & \textbf{Zoe} & \_\_ \end{pmatrix}$$

Exact same thing! We just have ratings instead of pixels.

PCA will reveal typical users.

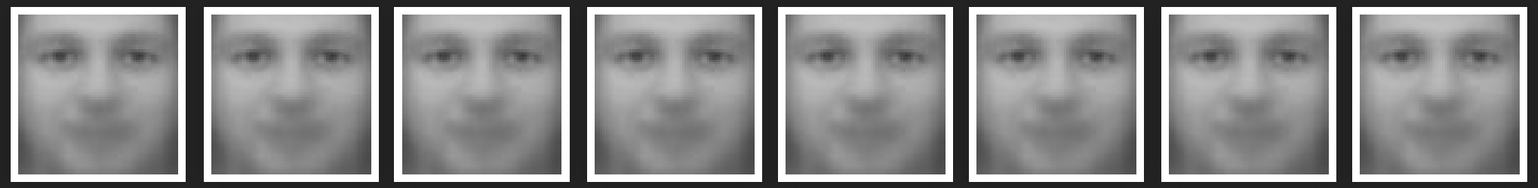
PCA ON A RATING MATRIX? SURE!

Assume all ratings are known

$$X = \begin{pmatrix} \_\_ & \text{Face 1} & \_\_ \\ \_\_ & \text{Face 2} & \_\_ \\ & \vdots & \\ \_\_ & \text{Face 400} & \_\_ \end{pmatrix} \qquad R = \begin{pmatrix} \_\_ & \text{Alice} & \_\_ \\ \_\_ & \text{Bob} & \_\_ \\ & \vdots & \\ \_\_ & \text{Zoe} & \_\_ \end{pmatrix}$$

Exact same thing! We just have ratings instead of pixels.

PCA will reveal typical users.



$$\begin{aligned} \text{Alice} &= 10\% \text{ Action fan} + 10\% \text{ Comedy fan} + 50\% \text{ Romance fan} + \cdots \\ \text{Bob} &= 50\% \text{ Action fan} + 30\% \text{ Comedy fan} + 10\% \text{ Romance fan} + \cdots \\ \text{Zoe} &= \cdots \end{aligned}$$

PCA ON A RATING MATRIX? SURE!

Assume all ratings are known. Transpose the matrix

$$X = \begin{pmatrix} \_\_ & \text{Face 1} & \_\_ \\ \_\_ & \text{Face 2} & \_\_ \\ & \vdots & \\ \_\_ & \text{Face 400} & \_\_ \end{pmatrix} \qquad R^T = \begin{pmatrix} \_\_ & \text{Titanic} & \_\_ \\ \_\_ & \text{Toy Story} & \_\_ \\ & \vdots & \\ \_\_ & \text{Fargo} & \_\_ \end{pmatrix}$$

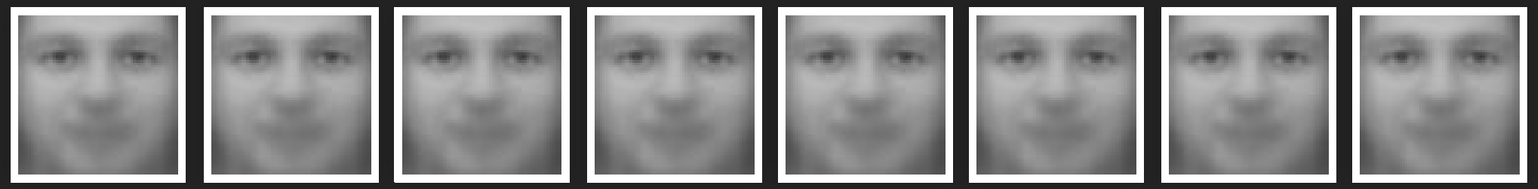
Exact same thing! PCA will reveal typical movies.

PCA ON A RATING MATRIX? SURE!

Assume all ratings are known. Transpose the matrix

$$X = \begin{pmatrix} — & \text{Face 1} & — \\ — & \text{Face 2} & — \\ & \vdots & \\ — & \text{Face 400} & — \end{pmatrix} \qquad R^T = \begin{pmatrix} — & \text{Titanic} & — \\ — & \text{Toy Story} & — \\ & \vdots & \\ — & \text{Fargo} & — \end{pmatrix}$$

Exact same thing! PCA will reveal typical movies.



$$\text{Titanic} = 20\% \text{ Action} + 0\% \text{ Comedy} + 70\% \text{ Romance} + \cdots$$
$$\text{Toy Story} = 30\% \text{ Action} + 60\% \text{ Comedy} + 0\% \text{ Romance} + \cdots$$

Note: in practice, the factors semantic is not clearly defined.

SVD IS PCA^2

- PCA on R gives you the typical users U
- PCA on R^T gives you the typical movies M
- SVD gives you both in one shot!

SVD IS PCA[2]

- PCA on R gives you the typical users U
- PCA on R^T gives you the typical movies M
- SVD gives you both in one shot!

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

Σ is diagonal, it's *just* a scaler.

$$\boxed{R = MU^T}$$

This is our matrix factorization!

THE MODEL OF SVD

$$\boxed{R = MU^T}$$

$$\begin{pmatrix} & r_{ui} & \end{pmatrix} = \begin{pmatrix} & — \quad p_u \quad — & \end{pmatrix} \begin{pmatrix} & | & \\ & q_i & \\ & | & \end{pmatrix}$$

$$r_{ui} = p_u \cdot q_i$$

THE MODEL OF SVD

$$R = MU^T$$

$$\begin{pmatrix} & r_{ui} & \end{pmatrix} = \begin{pmatrix} & — \quad p_u \quad — & \end{pmatrix} \begin{pmatrix} & | & \\ & q_i & \\ & | & \end{pmatrix}$$

$$r_{ui} = p_u \cdot q_i$$

$$r_{ui} = \sum_{c \in \text{concepts}} \text{affinity of } u \text{ for } c \times \text{affinity of } i \text{ for } c$$

THE MODEL OF SVD

$$\boxed{R = MU^T}$$

$$\begin{pmatrix} & r_{ui} & \end{pmatrix} = \begin{pmatrix} & — \quad p_u \quad — & \end{pmatrix} \begin{pmatrix} & | & \\ & q_i & \\ & | & \end{pmatrix}$$

$$r_{ui} = p_u \cdot q_i$$

$$r_{ui} = \sum_{c \in \text{concepts}} \text{affinity of } u \text{ for } c \times \text{affinity of } i \text{ for } c$$

$$\begin{aligned} \text{Titanic} &= 20\% \text{ Action} + 0\% \text{ Comedy} + 70\% \text{ Romance} + \cdots \\ \text{Alice} &= 15\% \text{ Action} + 0\% \text{ Comedy} + 80\% \text{ Romance} + \cdots \\ \text{Bob} &= 10\% \text{ Action} + 80\% \text{ Comedy} + 5\% \text{ Romance} + \cdots \end{aligned}$$

THE MODEL OF SVD

$$\boxed{R = MU^T}$$

$$\begin{pmatrix} & r_{ui} & \end{pmatrix} = \begin{pmatrix} & — \quad p_u \quad — & \end{pmatrix} \begin{pmatrix} & | & \\ & q_i & \\ & | & \end{pmatrix}$$

$$r_{ui} = p_u \cdot q_i$$

$$r_{ui} = \sum_{c \in \text{concepts}} \text{affinity of } u \text{ for } c \times \text{affinity of } i \text{ for } c$$

$$\begin{aligned} \text{Titanic} &= 20\% \text{ Action} + 0\% \text{ Comedy} + 70\% \text{ Romance} + \cdots \\ \text{Alice} &= 15\% \text{ Action} + 0\% \text{ Comedy} + 80\% \text{ Romance} + \cdots \\ \text{Bob} &= 10\% \text{ Action} + 80\% \text{ Comedy} + 5\% \text{ Romance} + \cdots \end{aligned}$$

Rating(Alice, Titanic) will be high

Rating(Bob, Titanic) will be low

SO HOW TO COMPUTE M AND U?

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

- Columns of M are the eigenvectors of RR^T
- Columns of U are the eigenvectors of $R^T R$
- Associated eigenvalues make up the diagonal of Σ
- There are very efficient algorithms in the wild

SO HOW TO COMPUTE M AND U?

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

- Columns of M are the eigenvectors of RR^T
- Columns of U are the eigenvectors of $R^T R$
- Associated eigenvalues make up the diagonal of Σ
- There are very efficient algorithms in the wild
- Alternate option: find the p_u s and the q_i s that minimize the reconstruction error

$$\sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

(With some orthogonality constraints). There are also very efficient algorithms in the wild

SO HOW TO COMPUTE M AND U?

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

- Columns of M are the eigenvectors of RR^T
- Columns of U are the eigenvectors of $R^T R$
- Associated eigenvalues make up the diagonal of Σ
- There are very efficient algorithms in the wild
- Alternate option: find the p_u s and the q_i s that minimize the reconstruction error

$$\sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

(With some orthogonality constraints). There are also very efficient algorithms in the wild

So, piece of cake right?


OOPS!

WE ASSUMED THAT R WAS DENSE

But it's not! 99% are missing

RR^T and $R^T R$ are not even defined

So M and U are not defined either

There is no SVD $R = MU^T$

WE ASSUMED THAT R WAS DENSE

But it's not! 99% are missing

RR^T and $R^T R$ are not even defined

So M and U are not defined either

There is no SVD $R = MU^T$

Two options:

- Fill the missing entries with a simple heuristic
- *" Let's just not give a damn "* -- Simon Funk (ended-up top-3 of the Netflix Prize for some time)

APPROXIMATION

Dense case: find the p_us and the q_is that minimize the total reconstruction error

$$\sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

(With orthogonality constraints)

Sparse case: find the p_us and the q_is that minimize the partial reconstruction error

$$\sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

(Forget about orthogonality)

APPROXIMATION

Dense case: find the p_us and the q_is that minimize the total reconstruction error

$$\sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

(With orthogonality constraints)

Sparse case: find the p_us and the q_is that minimize the partial reconstruction error

$$\sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

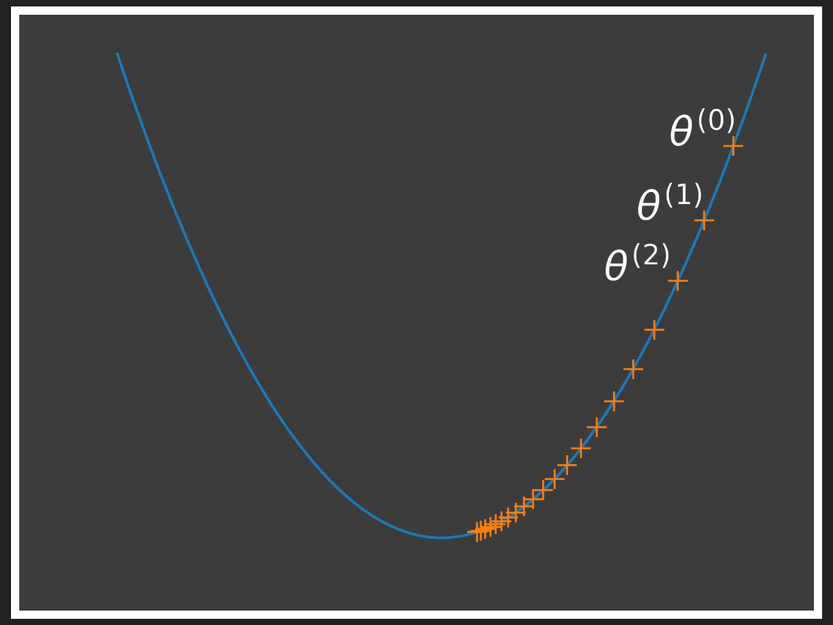
(Forget about orthogonality)

Basically the same, except we don't have all the ratings (and we don't care)

MINIMIZATION BY GRADIENT DESCENT

We want the value θ such that $f(\theta)$ is minimal:

- Compute $\frac{\partial f}{\partial \theta}$
- Randomly initialize θ
- $\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{\partial f}{\partial \theta}$ (do this until you're fed up)



MINIMIZATION BY (STOCHASTIC) GRADIENT DESCENT

Our parameter θ is (p_u, q_i) for all u and i. The function f to be minimized is:

$$f(p, q) = \sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2$$

```
def compute_SVD():
    '''Fit pu and qi to known ratings by SGD'''
    p = np.random.normal(size=F)
    q = np.random.normal(size=F)
    for iter in range(n_max_iter):
        for u, i, r_ui in rating_hist:
            err = r_ui - np.dot(p[u], q[i])
            p[u] = p[u] + learning_rate * err * q[i]
            q[i] = q[i] + learning_rate * err * p[u]

def estimate_rating(u, i):
    return np.dot(p[u], q[i])
```

SOME LAST DETAILS

Unbias the ratings, add regularization: you get "*SVD*":

$$\min_{p_u, q_i, b_u, b_i} \sum_{r_{ui} \in R} \left(\begin{array}{c} \left[r_{ui} - (\mu + b_u + b_i + p_u^T q_i) \right]^2 \\ +\lambda \left(||p_u||^2 + ||q_i||^2 + b_u^2 + q_i^2 \right) \end{array} \right)$$

SOME LAST DETAILS

Unbias the ratings, add regularization: you get "*SVD*":

$$\min_{p_u, q_i, b_u, b_i} \sum_{r_{ui} \in R} \left(\begin{array}{c} \left[r_{ui} - (\mu + b_u + b_i + p_u^T q_i) \right]^2 \\ +\lambda \left(||p_u||^2 + ||q_i||^2 + b_u^2 + q_i^2 \right) \end{array} \right)$$

Good ol' fashioned ML paradigm:

- Assume a model for your data ($R = MU^T$)
- Fit your model parameters to the observed data
- Praise the Lord and hope for the best

A FEW REMARKS

- Surprise is mostly a tool for algorithms that predict ratings. RS do much more than that.
- Focus on explicit ratings (implicit ratings algorithms will come, hopefully)
- Not aimed to be a complete tool for building RS from A to Z (check out Mangaki)
- Not aimed to be the most efficient implementation (check out LightFM)

SOME REFERENCES

- Can't recommend enough (pun intended) Aggarwal's Recommender Systems - The Textbook
- Jeremy Kun's blog (great insights on PCA and SVD)

THANKS!