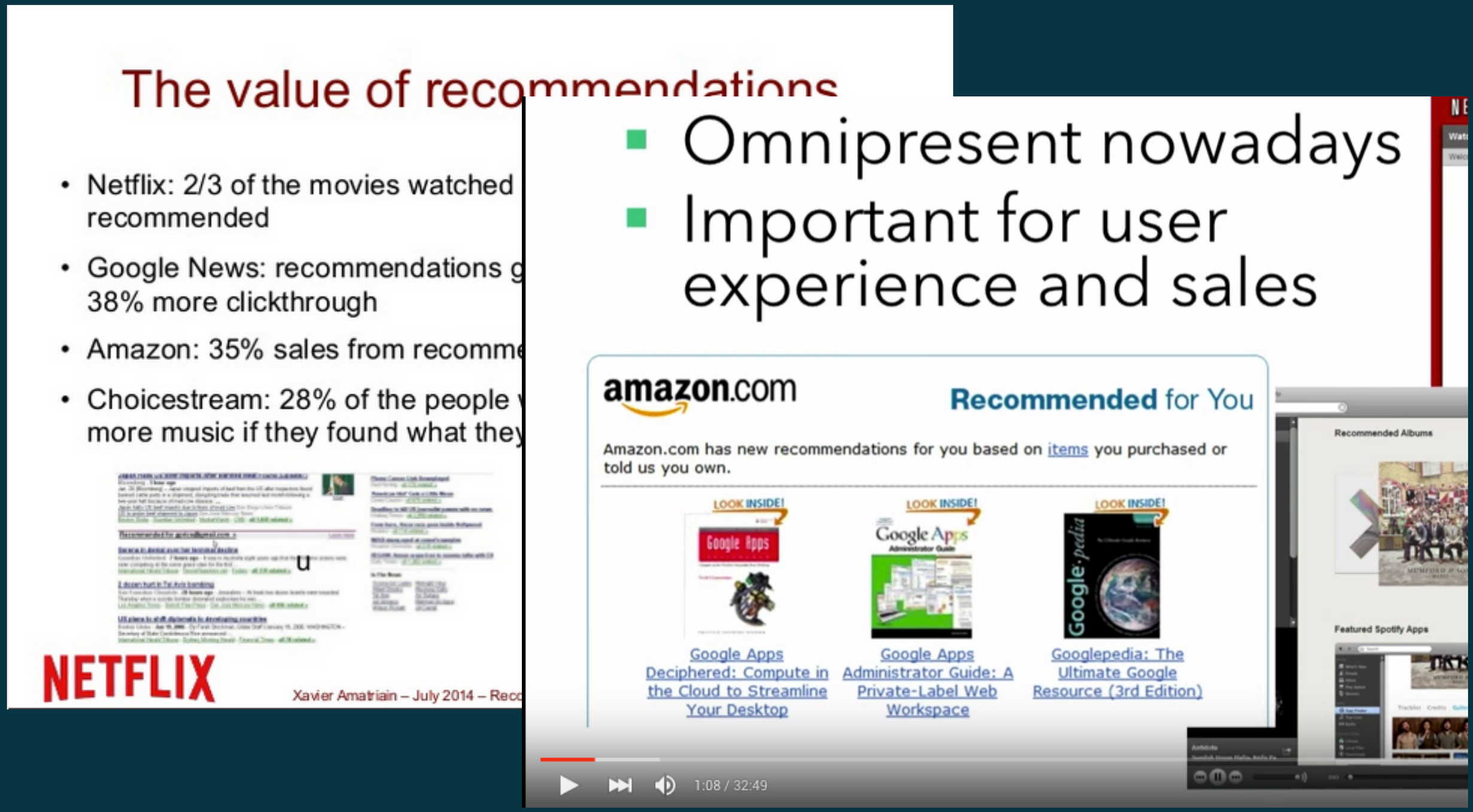
Recommender Systems at Scale


```
{"name": "Eoin Hurrell",
 "twitter": "@eoinhurrell",
 "github": "eoinhurrell",
 "email": "eoin.hurrell@gmail.com"}
```

Slides About Recommender Systems Being Everywhere are Everywhere



http://technocalifornia.blogspot.ie/2014/08/introduction-to-recommender-systems-4.html

https://www.youtube.com/watch?v=CmHPxASIj6Y

The Recommendation Task

We have a catalogue of items. Given what we know about the user what will they like best?

Companies want to get a prediction of the subjective rating a user would have of every item in their collection.

Collaborative Filtering

We can use what we know about what others like to predict what a user will like, but hasn't seen.

"You're a lot like Alice, and you haven't seen Alice's favourite film, you might like it"

Matrix Factorization

Matrix Factorization and Latent Factors

Matrix (Rating Database)

ITEM	U1	U2	...	UN
I1	3		...	
I2		4	...	
...	...	...	...	...
IN	2		...	

User Factors

Item Factors

Matrix Factorization

Latent Factors are variables contributing to the rating. They are often opaque, but think of them as:

User Factors — E.g. "I rarely watch films"

Item Factors — E.g. "This film has lots of explosions."

Matrix Factorization

Latent Factors can rebuild the matrix

User Factors

Item Factors

Matrix with Predictions

ITEM	U1	U2	...	UN
I1	3	1	...	2
I2	3	4	...	4
...	...	...	...	...
IN	2	5	...	5

Matrix Factorization

Problems:

We only partially know the User and Item Latent Factors.

Estimating Latent Factors can be computationally expensive and difficult to scale.

Alternating Least Squares

A method of matrix factorization that can easily be computed in parallel

http://bugra.github.io/work/notes/2014-04-19/alternating-least-squares-method-for-collaborative-filtering/

Alternating Least Squares

* Estimate Item Factor matrix using current User Factors
* Estimate User Factor matrix using current Item Factors
* Iterate to convergence (either User or Item matrix no longer changes)

http://bugra.github.io/work/notes/2014-04-19/alternating-least-squares-method-for-collaborative-filtering/

From Theory to Practice

How do we implement this in a way that can deal with lots of data?

Apache Spark

What is Spark?

"Apache Spark™ is a fast and general engine for large-scale data processing."

Spark **runs on Hadoop**, Mesos, **standalone**, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

You can run Spark using its standalone cluster mode, on EC2, on Hadoop YARN, or on Apache Mesos. Access data in HDFS, Cassandra, HBase, Hive, Tachyon, and any Hadoop data source.

Using Spark to Scale

Spark is fast and has support for exactly this type of operation, with libraries such as MLLib

The Approach

Start small, with a small amount of data locally.
Then scale up to run in the cloud.

We can do both with Spark

The Code: Python

Python is sort of a second-class citizen in Spark.

I chose Python because it's Data Science Esperanto

The Data: MovieLens

http://grouplens.org/datasets/movielens/

Dataset	Size	Users	Movies	Tags
10K	5 MB	1,000	1,700	-
1M	6 MB	6,000	4,000	-
10M	63 MB	72,000	10,000	100,000
20M	132 MB	138,000	27,000	465,000
Latest (21M)	144 MB	230,000	30,000	510,000

```python
""" Example PySpark ALS application
"""
from pyspark import SparkContext  # pylint: disable=import-error
from pyspark.mllib.recommendation import ALS, Rating  # pylint: disable=import-error


def parse_rating(line):
    """ Parse Movielens Rating line to Rating object.
        UserID::MovieID::Rating::Timestamp
    """
    line = line.split('::')
    return Rating(int(line[0]), int(line[1]), float(line[2]))


def parse_movie(line):
    """ Parse Movielens Movie line to Movie tuple.
        MovieID::Title::Genres
    """
    line = line.split('::')
    return (line[0], line[1])


def main():
    """ Train and evaluate an ALS recommender.
    """
    # Set up environment
    sc = SparkContext("local[*]", "RecSys")

    # Load and parse the data
    data = sc.textFile("./data/ratings.dat")
    ratings = data.map(parse_rating)

    # Build the recommendation model using Alternating Least Squares
    rank = 10
    iterations = 50
    model = ALS.train(ratings, rank, iterations)

    movies = sc.textFile("./data/movies.dat")\
               .map(parse_movie)
    # Evaluate the model on training data
    testdata = ratings.map(lambda p: (p[0], p[1]))
    predictions = model.predictAll(testdata)\
                       .map(lambda r: ((r[0], r[1]), r[2]))
    rates_and_preds = ratings.map(lambda r: ((r[0], r[1]), r[2]))\
                             .join(predictions)
    MSE = rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
    print("Mean Squared Error = " + str(MSE))
```

```python
""" Example PySpark ALS application
"""
from pyspark import SparkContext  # pylint: disable=import-error
from pyspark.mllib.recommendation import ALS, Rating  # pylint: disable=import-error


def parse_rating(line):
    """ Parse Movielens Rating line to Rating object.
        UserID::MovieID::Rating::Timestamp
    """
    line = line.split('::')
    return Rating(int(line[0]), int(line[1]), float(line[2]))


def parse_movie(line):
    """ Parse Movielens Movie line to Movie tuple.
        MovieID::Title::Genres
    """
    line = line.split('::')
    return (line[0], line[1])


def main():
    """ Train and evaluate an ALS recommender.
    """
    # Set up environment
    sc = SparkContext("local[*]", "RecSys")

    # Load and parse the data
    data = sc.textFile("./data/ratings.dat")
    ratings = data.map(parse_rating)

    # Build the recommendation model using Alternating Le
    rank = 10
    iterations = 50
    model = ALS.train(ratings, rank, iterations)

    movies = sc.textFile("./data/movies.dat")\
               .map(parse_movie)
    # Evaluate the model on training data
    testdata = ratings.map(lambda p: (p[0], p[1]))
    predictions = model.predictAll(testdata)\
                       .map(lambda r: ((r[0], r[1]), r[2]))
    rates_and_preds = ratings.map(lambda r: ((r[0], r[1]), r[2]))\
                             .join(predictions)
    MSE = rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
    print("Mean Squared Error = " + str(MSE))
```

Performing 50 iterations of ALS

```python
""" Example PySpark ALS application
"""
from pyspark import SparkContext  # pylint: disable=import-error
from pyspark.mllib.recommendation import ALS, Rating  # pylint: disable=import-error


def parse_rating(line):
    """ Parse Movielens Rating line to Rating object.
        UserID::MovieID::Rating::Timestamp
    """
    line = line.split('::')
    return Rating(int(line[0]), int(line[1]), float(line[2]))


def parse_movie(line):
    """ Parse Movielens Movie line to Movie tuple.
        MovieID::Title::Genres
    """
    line = line.split('::')
    return (line[0], line[1])


def main():
    """ Train and evaluate an ALS recommender.
    """
    # Set up environment
    sc = SparkContext("local[*]", "RecSys")

    # Load and parse the data
    data = sc.textFile("./data/ratings.dat")
    ratings = data.map(parse_rating)

    # Build the recommendation model using Alternating Least Squares
    rank = 10
    iterations = 50
    model = ALS.train(ratings, rank, iterations)

    movies = sc.textFile("./data/movies.dat")\
               .map(parse_movie)
    # Evaluate the model on training data
    testdata = ratings.map(lambda p: (p[0], p[1]))
    predictions = model.predictAll(testdata)\
                       .map(lambda r: ((r[0], r[1]), r[2]))
    rates_and_preds = ratings.map(lambda r: ((r[0], r[1]), r[2]))\
                             .join(predictions)
    MSE = rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
    print("Mean Squared Error = " + str(MSE))
```

Evaluating accuracy

User 1 likes:

Sympathy for Lady Vengeance (2005),
Boys Don't Cry (2000),
District B13 (2004),
BloodRayne (2005),
Last Holiday (2006),
Underworld: Evolution (2006)

User 1 would like:

Volver (2006), predicted rating: 4.5

github.com/eoinhurrell/python-spark-recsys

Running this at scale

Spark cluster computing runs from 1 to N nodes.

Spark on AWS

AWS EC2 - spark-ec2 commandline tool allows you to do this trivially.

AWS EMR - This can be done through the AWS Console. Even though the UI mentions 'application JARs' you can use Python code.

Spark on AWS

Cost: >$10 to predict items for 1,000 users.
Amazon make 35% of their sales from recommendations [1].
If 5% of those users buy something
18 of those sales will be from recommendations.

If you make more than $0.56 per sale on average recommenders are worth investigating.

[1] http://technocalifornia.blogspot.ie/2014/08/introduction-to-recommender-systems-4.html

Conclusion

* ALS Matrix Factorization is efficient for recommendation
* Spark has built-in support for such high-level operations
* Working with Spark is easy at any scale, with no code changes

github.com/eoinhurrell/python-spark-recsys


```
{"name": "Eoin Hurrell",
 "twitter": "@eoinhurrell",
 "github": "eoinhurrell",
 "email": "eoin.hurrell@gmail.com"}
```

Questions?

eoin.hurrell@gmail.com

github.com/eoinhurrell/python-spark-recsys


```
{"name": "Eoin Hurrell",
 "twitter": "@eoinhurrell",
 "github": "eoinhurrell",
 "email": "eoin.hurrell@gmail.com"}
```