Analyzing social media data with Apache Spark using Python

David Vrba (Socialbakers)

22/2/2019

MLPrague Workshop

Socialbakers

- Founded in 2010, Pilsen, Czech Republic
    - Our product (SaaS): AI-Powered Social Media Marketing Suite
        - Analytics/Dashboard
        - Publishing
        - Influencers
        - Audiences

socialbakers

Lecturer

- David Vrba Ph.D.
- Data Scientist at Socialbakers (Prague office)
  - Analyzing data and building ML prototypes
  - Working with Spark
    - optimizing Spark jobs
    - productionalizing Spark applications
    - lecturing Spark trainings
- Get in touch on LinkedIn: www.linkedin.com/in/vrba-david

Teaching Assistants (TA’s)

- Peter Vasko
- Data Architect at Socialbakers

- Pavol Knapek
- Data Engineer at Socialbakers

Disclaimer

- Presented content is meant for educational purpose, it is not meant to be used in production setting.
- All used data is samples taken from social networks and they are modified and partially auto-generated for the purpose of this workshop. The samples are not representative and they should not be used to infer any conclusions.

Outline of the workshop

1) Interactive data analysis with DataFrame API

2) Cluster and predictive analysis with ML Pipelines

3) Advanced data analysis using graph processing algorithms with GraphFrames

Outline of the workshop

1. Theory to all three parts in the form of slide presentation
2. Hands-on
   a. we will solve prepared problems in the first two parts
      i. Answer some analytical questions about dataset
      ii. Build predictive model
      iii. Run cluster analysis

Data

1. Provided by Socialbakers
    a. Instagram influencers
    b. Facebook pages with posts
    c. Interactions of users with Facebook pages (affinities)

Environment

- Provided and sponsored by Databricks
- We prepared clusters with Spark 2.4
  - driver + 2 workers (16 cores each)
  - just attach a notebook and run your query!

https://mlprague2019.cloud.databricks.com

- If you don't have access:
  - **mlprague2019@socialbakers.com**

Workshop scope

- Machine learning with Spark is simple
    - Spark contains scalable version of algorithms
    - The API of ML Pipelines is user-friendly
- The difficult parts are
    - preprocessing the data
    - finding good features

Workshop scope

- Today we will
  - focus on the DataFrame API so you can preprocess your data well
  - see basic concepts of the Spark libraries
  - see how to analyze data and build ML prototypes in Spark
- Today we will not
  - cover the mathematical theory behind ML algorithms
  - look for the best features to end up with the most accurate model
  - build production ready ML applications

Part I

Interactive data analysis with DataFrame API

- Introduction to Spark and DataFrame API (30 mins)
  - Introduction to Spark
  - Show the API of DataFrames to write complex analytical queries
- Lab 1 (45 mins)
  - Introduce Databricks platform (10 mins)
  - Explain the influencers dataset
  - Run analytical queries on influencers data
  - Explain the solution

Outline for the theory in part I

- Introduction to Spark and DataFrame API
- Aggregations
- Window Functions
- User Defined Functions (with Pandas)
- Higher Order Functions (if there is time)

Unified framework

- ETL
- Ad-hoc analytics (has interactive console)
- Machine learning
- Graph processing
- Stream processing

Unified API for reading data

- Supports different datasources (parquet, csv, json, ...)

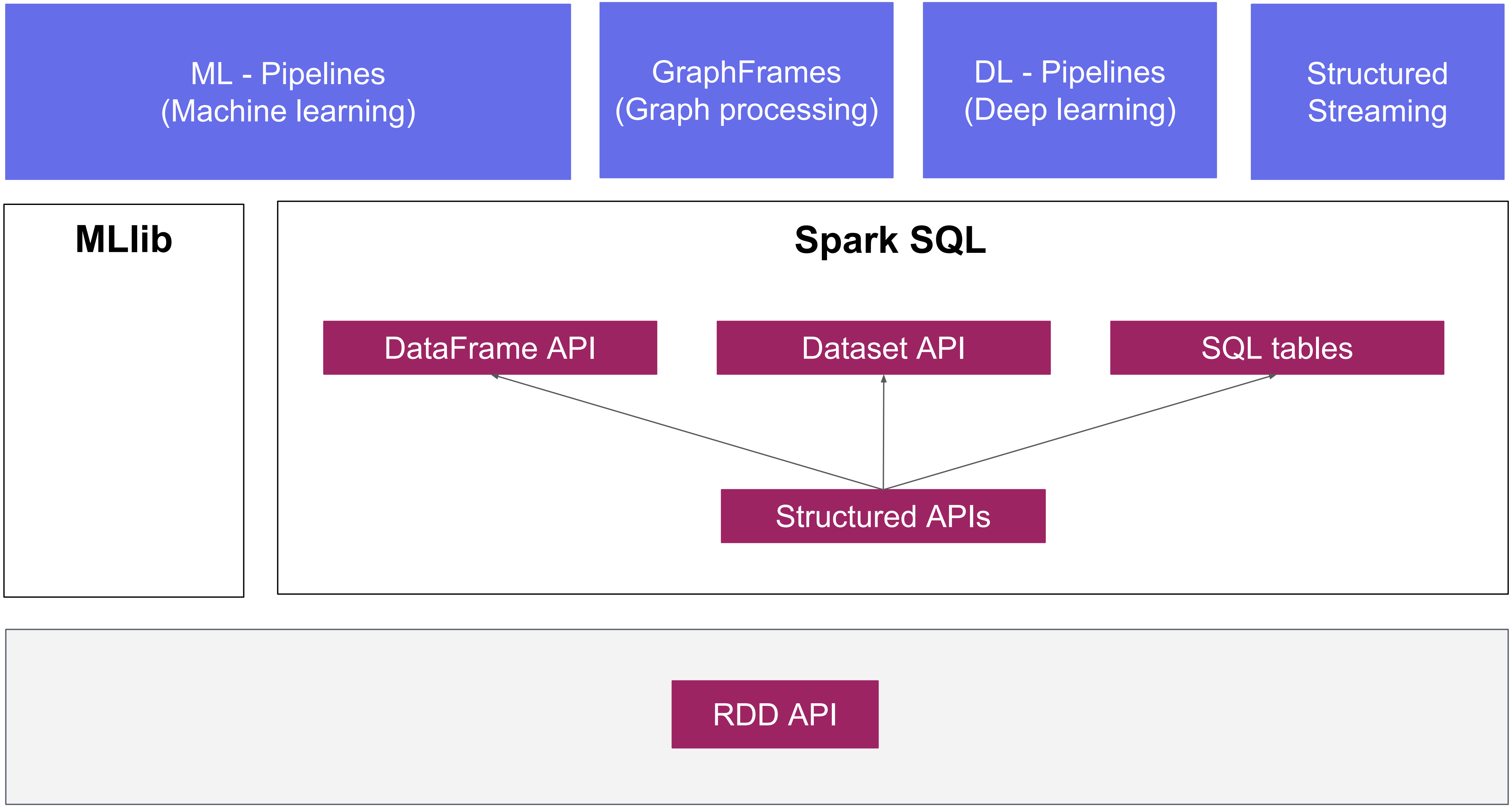
Spark

Easily accessible

- Supports multiple languages: Scala, Java, Python, SQL, R

Fast

- Can process data in parallel: is distributed and fault-tolerant
- Capable of in-memory computation
- Optimizes code under the cover


ML - Pipelines
(Machine learning)
GraphFrames
(Graph processing)
DL - Pipelines
(Deep learning)
Structured
Streaming
MLlib
Spark SQL
DataFrame API
Dataset API
SQL tables
Structured APIs
RDD API

What is a DataFrame?

- Representation of distributed data
- Represents data with some (tabular) structure
- Each record is a row and has schema (field name with data type)
- It has rows and columns

How to create a DataFrame?

Without metastore:

```
spark.read
      .format('data_format')
      .option('key', 'value')
      .schema(my_schema)
      .load()
```

With metastore:

```
spark.table(table_name)
```

All important information gets from the metastore

Operations in Spark

Transformations

- DataFrame transformations
- Column transformations
- Both of them are lazy
- Do not send query for execution

```
df.select('id', 'first_name')
df.dropDuplicates()
df.orderBy('age')
```

```
df.withColumn(
    'full_name', concat('first_name', lit(' '), 'last_name')
)
```

Operations in Spark

```
df.count()
df.show()
df.collect()
```

Actions

- Materialize the query
- Trigger computation
- Function in which you ask for output

Transformations

- DataFrame transformations
- Column transformations
- Both of them are lazy
- Do not send query for execution

```
df.select('id', 'first_name')
df.dropDuplicates()
df.orderBy('age')
```

```
df.withColumn(
    'full_name', concat('first_name', lit(' '), 'last_name')
)
```

Supported transformations

Filtering

```
df.filter(col('user_id').isNotNull()) \
   .filter(col('year') == 2018)
```

Joins

```
users.join(activities, 'user_id', 'left')

users.join(deleted, 'user_id', 'left_anti')
```

Deduplication

```
df.dropDuplicates(['user_id', 'snapshot_on'])
```

Nulls handling

```
df.fillna(0)

df.fillna({'age': 0, 'name': 'unknown'})
```

Aggregations

```
df.groupBy('age') \
  .agg(
      count('*').alias('age_frequency')
)
```

DataFrame API vs SQL API

```
df.filter(col('user_id').isNotNull()) \
   .filter(col('year') == 2018)
```

equivalent Spark queries

```
(
      df.filter(col('user_id').isNotNull())
      .filter(col('year') == 2018)
)
```

DataFrame API vs SQL API

```
df.filter(col('user_id').isNotNull()) \
   .filter(col('year') == 2018)
```

equivalent Spark queries

```
(
    df.filter(col('user_id').isNotNull())
    .filter(col('year') == 2018)
)
```

```
df.createOrReplaceTempView('my_view')

spark.sql(
    """
        SELECT *
        FROM my_view
        WHERE user_id IS NOT NULL
        AND year = 2018
    """
)
```

Aggregations and Window functions

- Contained in three classes
  - pyspark.sql.functions
  - pyspark.sql.DataFrameStatFunctions
  - pyspark.sql.GroupedData

Three types of aggregations

Aggregating whole DataFrame

```
df.select(count('id'))
```

```
df.agg(count('id'))
```

Needs to be imported

```
from pyspark.sql.functions import count
```

Both options support multiple aggregations:

```
df.select(count('id'), sum('id'))
```

Aggregations based on groups

- Using
  - groupBy()
  - cube()
  - rollup()

Returns instance of class GroupedData

You need to call some aggregation function on it

```
df.groupBy('id') \
  .count()
```

- Function is from class GroupedData
- Does not need to be imported
- You can not rename it using alias()!

```
df.groupBy('id') \
  .agg(count('*'))
```

- Function is from class spark.sql.functions
- Needs to be imported
- You can rename it using alias()

Aggregations based on frame

- Three types of window functions
  - Ranking functions: rank, ntile, row_number, ...
  - Analytic functions: lag, lead, ...
  - Aggregate functions: count, sum, ...

Aggregations based on frame

- Three types of window functions
  - Ranking functions: rank, ntile, row_number, ...
  - Analytic functions: lag, lead, ...
  - Aggregate functions: count, sum, ...

```
from pyspark.sql.window import Window
```

```
w = Window.partitionBy(...) \
           .orderBy(...) \
           .frame_specification
```

```
df.select(sum(...).over(w))
```

Only order of rows matters

`rowsBetween(a, b)`

`rangeBetween(a, b)`

The column value matters

Aggregations based on frame – example

id	date
1	‘2019-01-10’
1	‘2019-01-01’
1	‘2019-01-20’
2	‘2019-01-15’
2	‘2019-01-07’
2	‘2019-01-30’

id	date	row_number
1	‘2019-01-01’	1
1	‘2019-01-07’	2
1	‘2019-01-10’	3
2	‘2019-01-07’	1
2	‘2019-01-15’	2
2	‘2019-01-30’	3

Aggregations based on frame – example

id	date
1	‘2019-01-10’
1	‘2019-01-01’
1	‘2019-01-20’
2	‘2019-01-15’
2	‘2019-01-07’
2	‘2019-01-30’

```
w = Window.partitionBy(‘id’).orderBy(‘date’)

data.withColumn(‘row_number’, row_number().over(w))
```

id	date	row_number
1	‘2019-01-01’	1
1	‘2019-01-07’	2
1	‘2019-01-10’	3
2	‘2019-01-07’	1
2	‘2019-01-15’	2
2	‘2019-01-30’	3

User Defined Functions (UDFs)

- Possibility to execute custom code on the DataFrame as a column transformation
- Extend the functionality through simple interface
- Very powerful but it comes with big responsibility

UDFs

User Defined Functions (UDFs)

- Possibility to execute custom code on the DataFrame as a column transformation
- Extend the functionality through simple interface
- Very powerful but it comes with big responsibility

UDFs

- Try to avoid using it as much as possible
- Any time you use it, you pay big performance penalty
    - It is quite ok if you use Scala or Java
    - But there is a big difference in Python

User Defined Functions (UDFs)

- Possibility to execute custom code on the DataFrame as a column transformation
- Extend the functionality through simple interface
- Very powerful but it comes with big responsibility

UDFs

- Integrated with Pandas
- Integrated with Apache Arrow
- Enables vectorized execution

- Try to avoid using it as much as possible
- Any time you use it, you pay big performance penalty
    - It is quite ok if you use Scala or Java
    - But there is a big difference in Python

User Defined Functions (UDFs)

- Possibility to execute custom code on the DataFrame as a column transformation
- Extend the functionality through simple interface
- Very powerful but it comes with big responsibility

- Consider using Scala if performance is crucial

UDFs

- Integrated with Pandas
- Integrated with Apache Arrow
- Enables vectorized execution

- Try to avoid using it as much as possible
- Any time you use it, you pay big performance penalty
    - It is quite ok if you use Scala or Java
    - But there is a big difference in Python

UDFs – define and register

```
@udf(IntegerType())
def my_python_udf(my_str):
    return len(my_str)
```

Spark SQL types:

- IntegerType
- LongType
- StringType
- DateType
- ArrayType
- StructType
- MapType

UDFs – define and register

```
@udf(IntegerType())
def my_python_udf(my_str):
    return len(my_str)
```

Spark SQL types:

- IntegerType
- LongType
- StringType
- DateType
- ArrayType
- StructType
- MapType

```
@pandas_udf(IntegerType(), PandasUDFType.SCALAR)
def my_pandas_udf(my_str):
    return my_str.str.len()
```

PandasUDFType

- SCALAR
- GROUPED_MAP
- GROUPED_AGG

Using UDFs

id	message
1	‘This is my text.’
2	‘How is my dog?’

id	message	str_len
1	‘This is my text.’	16
2	‘How is my dog?’	14

Using UDFs

id	message
1	‘This is my text.’
2	‘How is my dog?’

id	message	str_len
1	‘This is my text.’	16
2	‘How is my dog?’	14

Using UDF:

```
df.withColumn(‘str_len’, my_python_udf(‘message’))
```

Using UDFs

id	message
1	‘This is my text.’
2	‘How is my dog?’

id	message	str_len
1	‘This is my text.’	16
2	‘How is my dog?’	14

Using UDF:

```
df.withColumn(‘str_len’, my_python_udf(‘message’))
```

But it can be done also natively:

```
df.withColumn(‘str_len’, length(‘message’))
```

GROUPED_MAP PandasUDFType

id	value
1	10
1	11
1	12
1	13
2	1
2	2
2	3

id	value	gmean
1	10	11.45
1	11	11.45
1	12	11.45
1	13	11.45
2	1	1.82
2	2	1.82
2	3	1.82

- compute custom aggregation for each group
- it is convenient if the aggregation can be done using Pandas
- check the SciPy package

GROUPED_MAP PandasUDFType

```
schema = StructType(
    [
        StructField('id', IntegerType()),
        StructField('value', IntegerType()),
        StructField('gmean', DoubleType())
    ]
)

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def compute_gmean(pdf):
    pdf['gmean'] = sc.stats.gmean(pdf['value'])
    return pdf
```

```
df.groupBy('id') \
  .apply(compute_gmean)
```

Complex Data Types in DataFrame API

- Allow you to store complex and nested data structures
- ArrayType
  - Much better support since 2.4
    - A lot of functions for array manipulation
    - Higher order functions
- StructType
- MapType

Complex Data Types in DataFrame API

- Allow you to store complex and nested data structures
- ArrayType
  - Much better support since 2.4
    - A lot of functions for array manipulation
    - Higher order functions
- StructType
- MapType

```
from pyspark.sql.functions import array


df.select(array('col_1', 'col_2', 'col_3').alias('my_array_col'))

df.select(col('my_array_col')[1])
```

Complex Data Types in DataFrame API

Array

- Variable length
- Homogeneous
- Ordered
- Accessed by index

Complex Data Types in DataFrame API

Array	Struct
• Variable length • Homogeneous • Ordered • Accessed by index	• Fixed number of fields • Heterogeneous • Ordered • Accessed by name

Complex Data Types in DataFrame API

Array	Struct	Map
● Variable length ● Homogeneous ● Ordered ● Accessed by index	● Fixed number of fields ● Heterogeneous ● Ordered ● Accessed by name	● Variable number of key–value pairs ● All keys have the same type ● All values have the same type ● Accessed by key name

Working with arrays in Spark SQL

- size
- array_min
- array_max
- array_sort
- array_distinct
- array_join
- array_overlap
- array_remove
- flatten
- slice
- reverse
- element_at

Working with arrays in Spark SQL

- size
- array_min
- array_max
- array_sort
- array_distinct
- array_join
- array_overlap
- array_remove
- flatten
- slice
- reverse
- element_at

ArrayType

id	words
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]

id	words	word_count
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]	5

Working with arrays in Spark SQL

- size
- array_min
- array_max
- array_sort
- array_distinct
- array_join
- array_overlap
- array_remove
- flatten
- slice
- reverse
- element_at

ArrayType

id	words
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]

```
df.withColumn(‘word_count’, size(‘words’))
```

id	words	word_count
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]	5

Working with arrays in Spark SQL

id	words
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]

id	length_of_words
1	[4, 2, 2, 4, 7]

one array

another array

Working with arrays in Spark SQL

id	words
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]

id	length_of_words
1	[4, 2, 2, 4, 7]

In functional programming languages: **arr.map(func)**

Higher order function

Higher Order Functions in Spark 2.4

- TRANSFORM function in the SQL API

```
df.createOrReplaceTempView(‘my_data’)

spark.sql(
    “““““
    SELECT
        words,
        TRANSFORM(words, value -> length(value)) AS length_of_words
    FROM my_data
    “““““
)
```

Higher Order Functions in Spark 2.4

- TRANSFORM function in the SQL API

```
df.createOrReplaceTempView('my_data')

spark.sql(
    """
    SELECT
        words,
        TRANSFORM(words, value -> length(value)) AS length_of_words
    FROM my_data
    """
)
```

```
df.selectExpr(
    'words',
    TRANSFORM(words, value -> length(value)) AS length_of_words
)
```

Higher Order Functions in Spark 2.4

- TRANSFORM
- FILTER
- EXISTS
- AGGREGATE

Example with AGGREGATE

id	words
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]

id	text
1	[‘ this is my text message’]

Example with AGGREGATE

id	words
1	[‘This’, ‘is’, ‘my’, ‘text’, ‘message’]

```
df.selectExpr(
        ‘id’,
        AGGREGATE(words, ‘’, (buffer, value) -> concat(buffer, ‘ ’, value)) AS text
)
```

id	text
1	[‘ this is my text message’]

Time for hands-on
Lab I

Let's switch to Databricks and see the platform

https://mlprague2019.cloud.databricks.com

- If you don't have access:
    - **mlprague2019@socialbakers.com**

Part II

Predictive and cluster analysis using ML Pipelines

- Introduction to Machine learning in Spark (30 mins)
  - General introduction to binary classification
  - Basic concepts of ML Pipelines: Transformer, Estimator, Pipeline, Evaluator
  - Some examples
- Lab II (30 mins)
- Explain the solution of the problems (10 mins)

Machine learning in Spark

- Supported from the beginning through a native library MLlib – one of the Spark modules
- MLlib
  - contains scalable version of some algorithms for machine learning
  - Has two subpackages:
    - mllib – provides RDD API
    - ml (ML Pipelines) – provides DataFrame based API
- In current version of Spark the RDD based API is in maintenance mode and is expected to be deprecated in Spark 3.0. The name of the package stays MLlib.

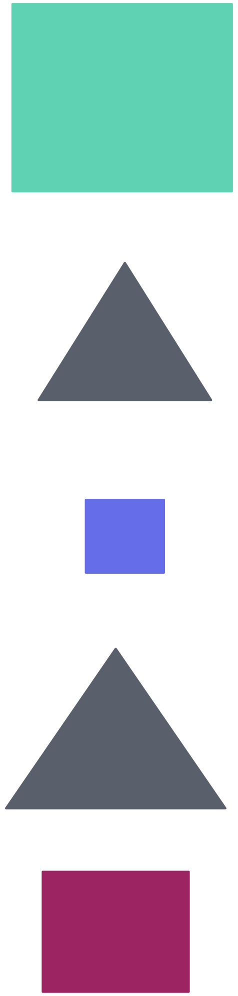
Submodules of ML package

- **classification** – logistic regression, decision tree, random forest, naive bayes, …
- **clustering** – k-means, gaussian mixture, LDA, …
- **regression** – generalized linear model, linear regression, …
- **recommendation** – ALS
- **tuning** – cross-validator, param grid builder
- **evaluation** – binary classification evaluator, multi-class classification evaluator, …
- **fpm** – PrefixSpan, FP-growth
- **linalg** – sparse vector, dense vector, sparse matrix, dense matrix
- **feature** – tokenizer, normalizer, one-hot encoder, count vectorizer, idf, bucketizer, …

Binary Classification in general

- All objects belong to one of 2 classes:
    - (0, 1)
    - (red, blue)
    - (positive, negative), …
- Feature engineering
    - describe each object by some attributes (predictors, features)

Binary Classification in general



object id = 1

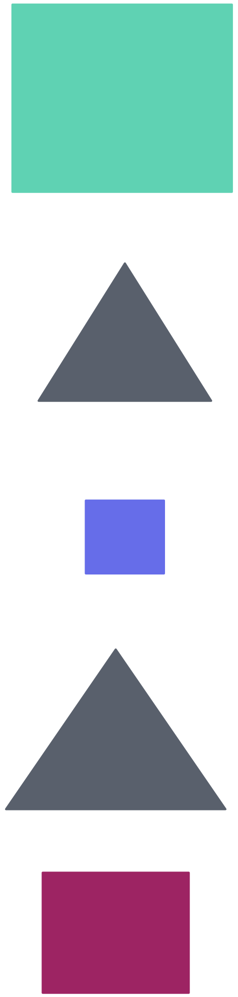
object id = 2

object id = 3

object id = 4

object id = 5

Binary Classification in general



object id = 1

object id = 2

object id = 3

object id = 4

object id = 5

object id
1
2
3
4
5

Binary Classification in general

object id = 1

object id = 2

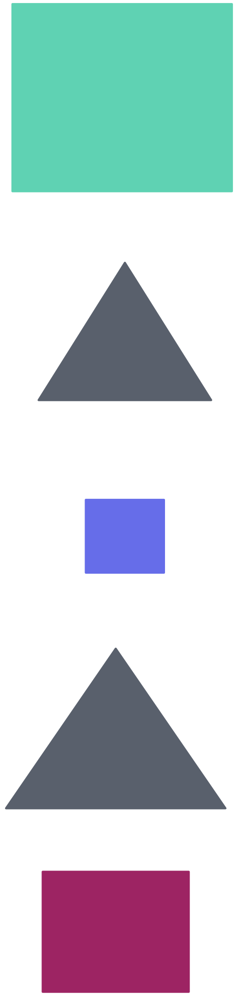
object id = 3

object id = 4

object id = 5

object id	area
1	0.93
2	0.72
3	0.2
4	0.8
5	0.8

Binary Classification in general



object id = 1

object id = 2

object id = 3

object id = 4

object id = 5

object id	area	color
1	0.93	green
2	0.72	grey
3	0.2	blue
4	0.8	grey
5	0.8	red

Binary Classification in general

object id = 1

object id = 2

object id = 3

object id = 4

object id = 5

object id	area	color	vertices
1	0.93	green	4
2	0.72	grey	3
3	0.2	blue	4
4	0.8	grey	3
5	0.8	red	4

Binary Classification in general

object id = 1

object id = 2

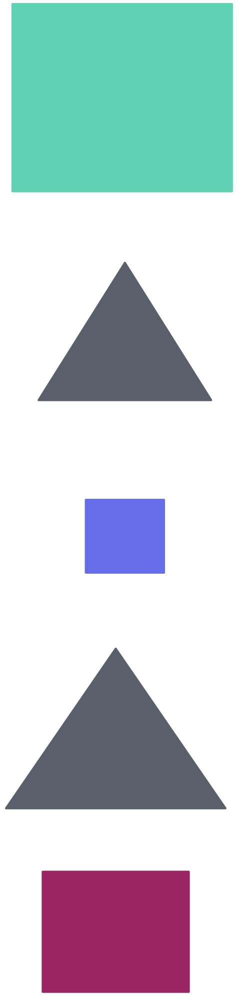
object id = 3

object id = 4

object id = 5

object id	area	color	vertices	label
1	0.93	green	4	square
2	0.72	grey	3	triangle
3	0.2	blue	4	square
4	0.8	grey	3	triangle
5	0.8	red	4	square

Binary Classification in general



object id = 1

object id = 2

object id = 3

object id = 4

object id = 5

object id	area	color	vertices	label
1	0.93	green	4	square
2	0.72	grey	3	triangle
3	0.2	blue	4	square
4	0.8	grey	3	triangle
5	0.8	red	4	square

$$f : (x1, x2, x3, ...) \rightarrow y$$

Two challenges

1. We need an algorithm that can learn the function from the data
   a. we will skip this part and rely on existing solutions
      i. logistic regression, decision tree, random forest, neural network, …
2. We need a way how to do it in Spark
   a. that is what we will do in the next 30 mins

ML Pipelines – abstraction for ML in Spark

- Basic concepts
  - fundamental structural types:
    - Transformer
    - Estimator
    - Evaluator
    - Pipeline
  - Low level data types
    - Dense Vector
    - Sparse Vector

Vectors

- Way how to represent our objects mathematically
- They will serve as input to learning algorithms
- Suppose that we have these predictors:

height	weight	price	age
10	8	0	1

Vectors

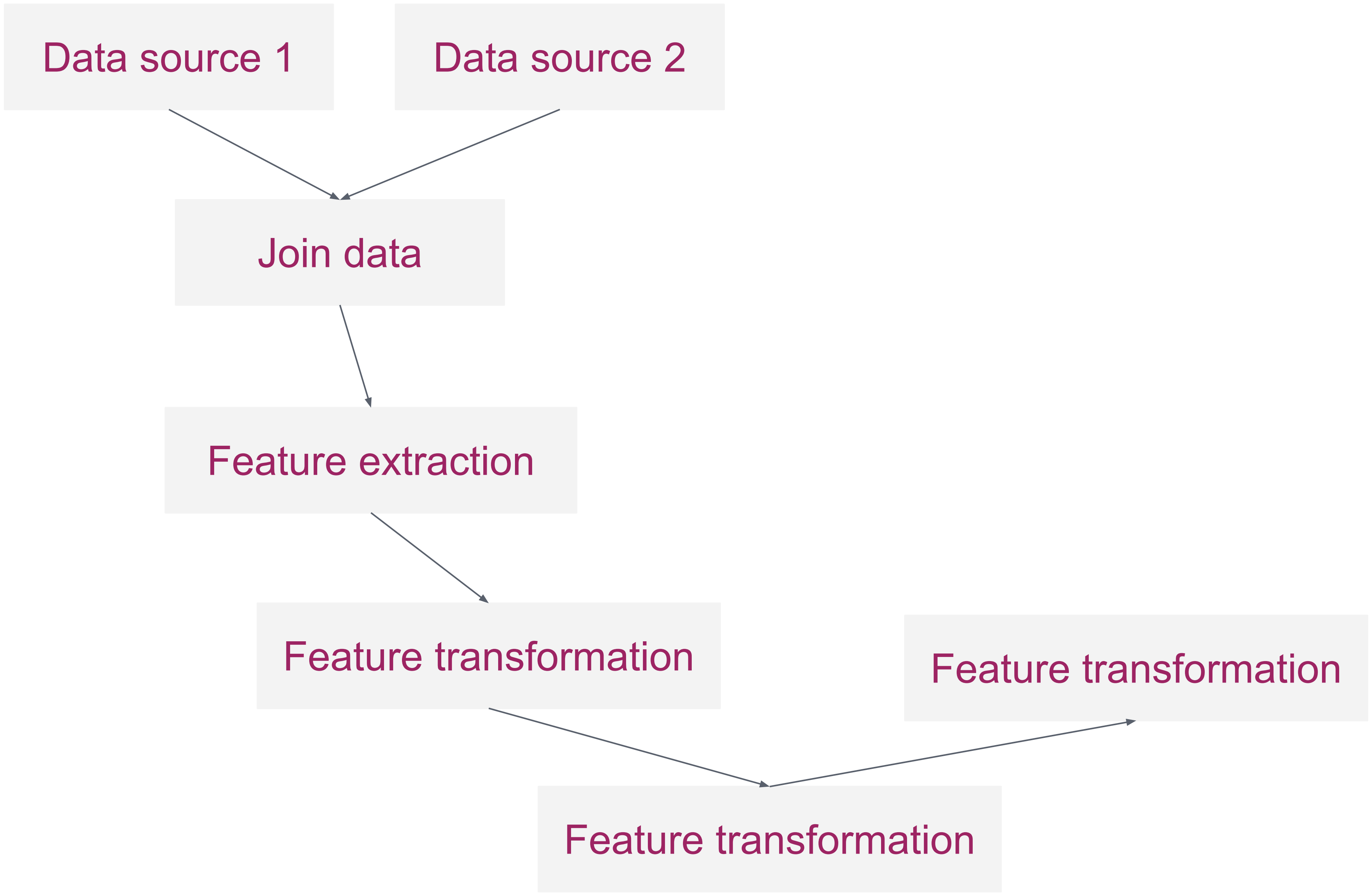
- Way how to represent our objects mathematically
- They will serve as input to learning algorithms
- Suppose that we have these predictors:

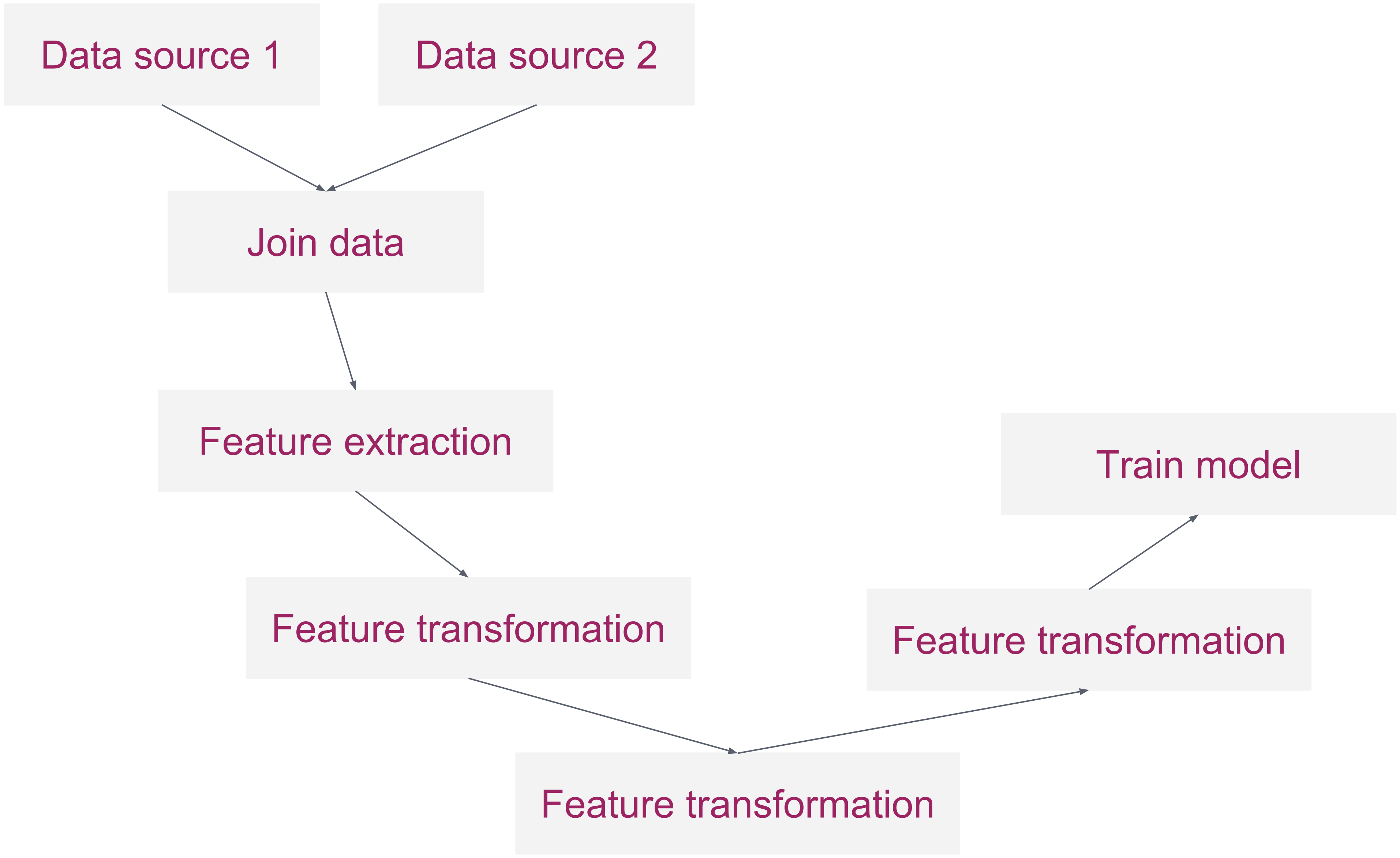
height	weight	price	age
10	8	0	1

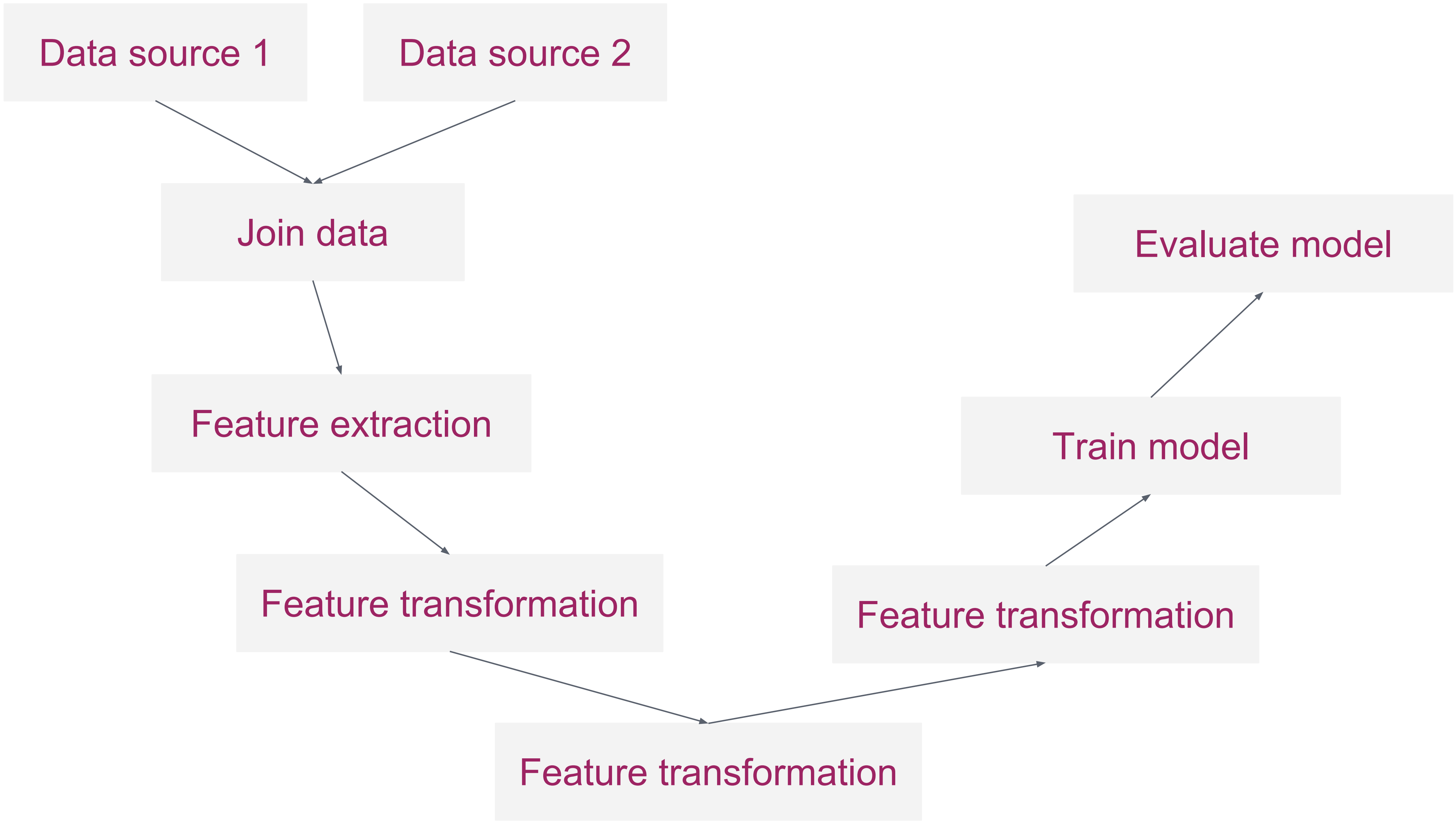
```
Vectors.dense(10.0, 8.0, 0.0, 1.0)
```

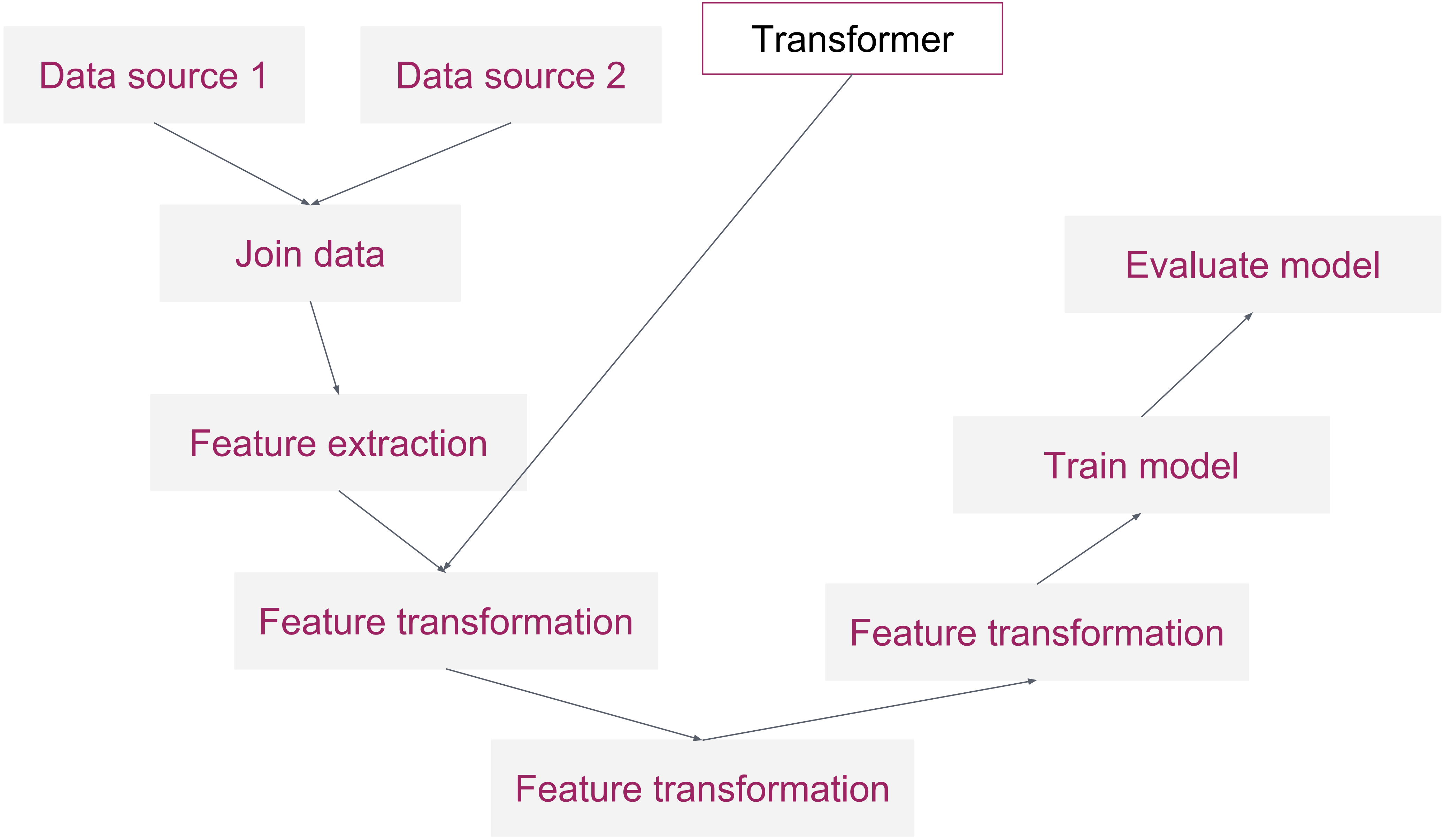
```
Vectors.sparse(4, [0, 1, 3], [10.0, 8.0, 1.0])
```

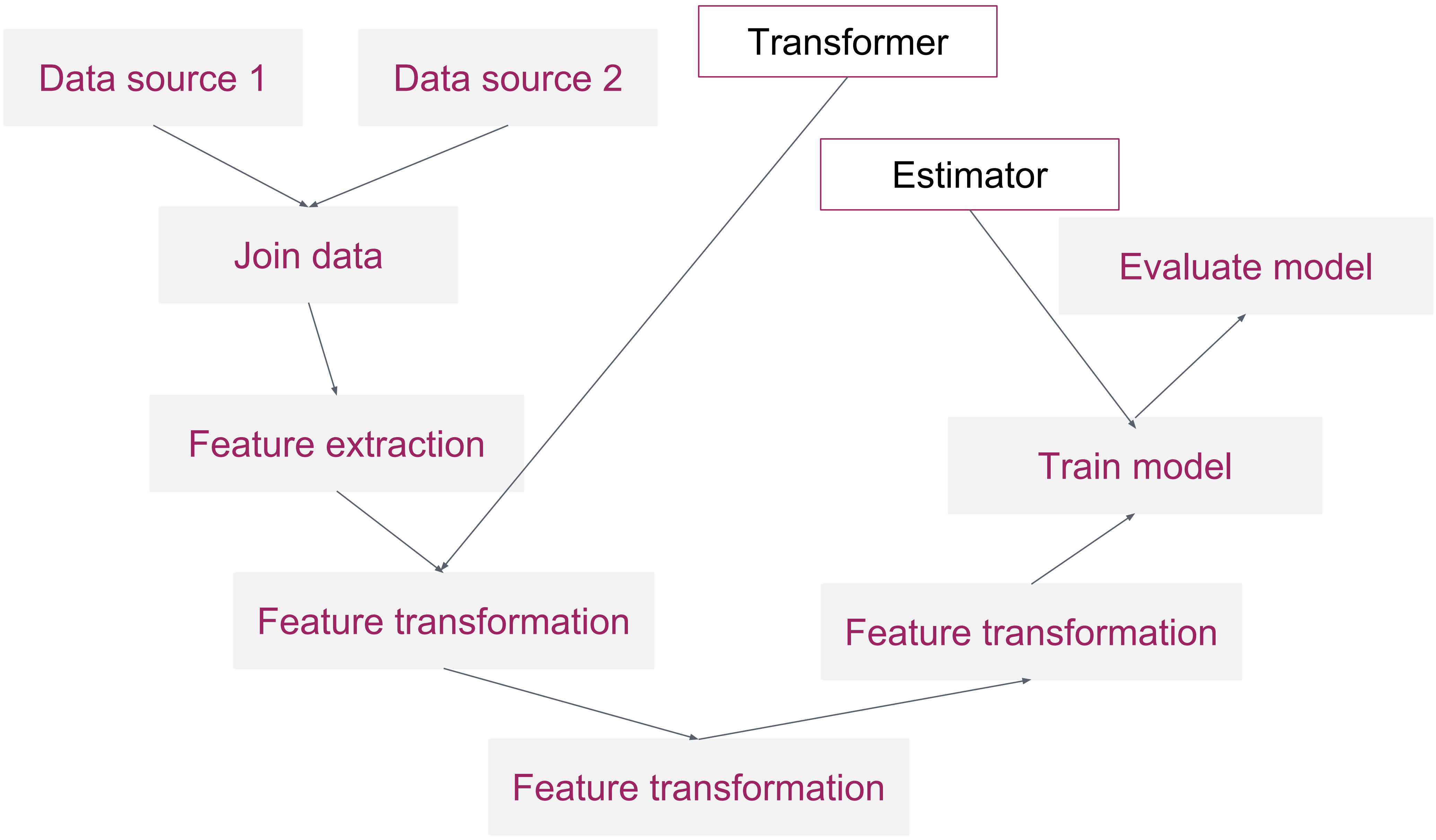

Data source 1
Data source 2
Join data

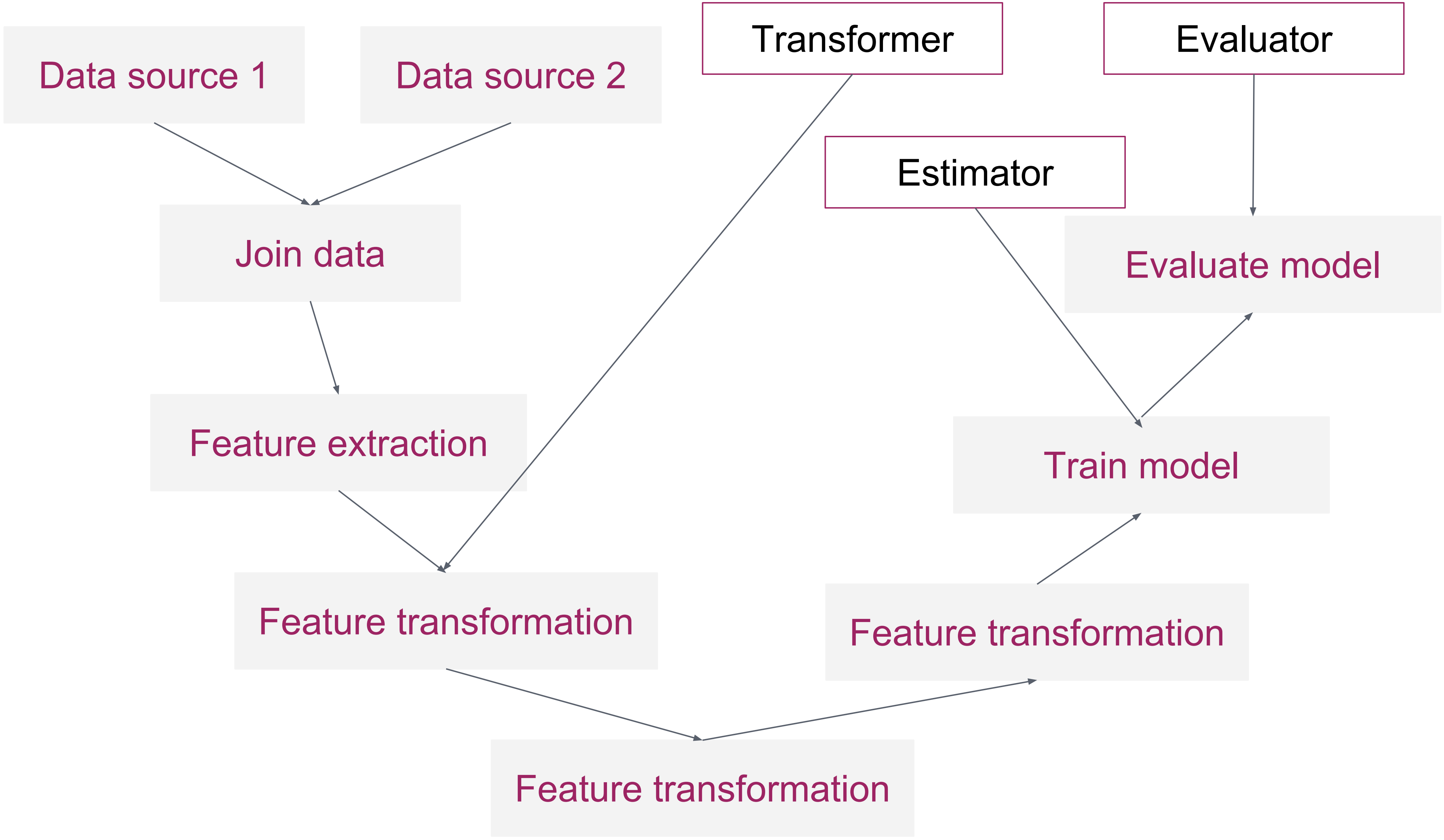

Data source 1
Data source 2
Join data
Feature extraction
Feature transformation
Feature transformation
Feature transformation

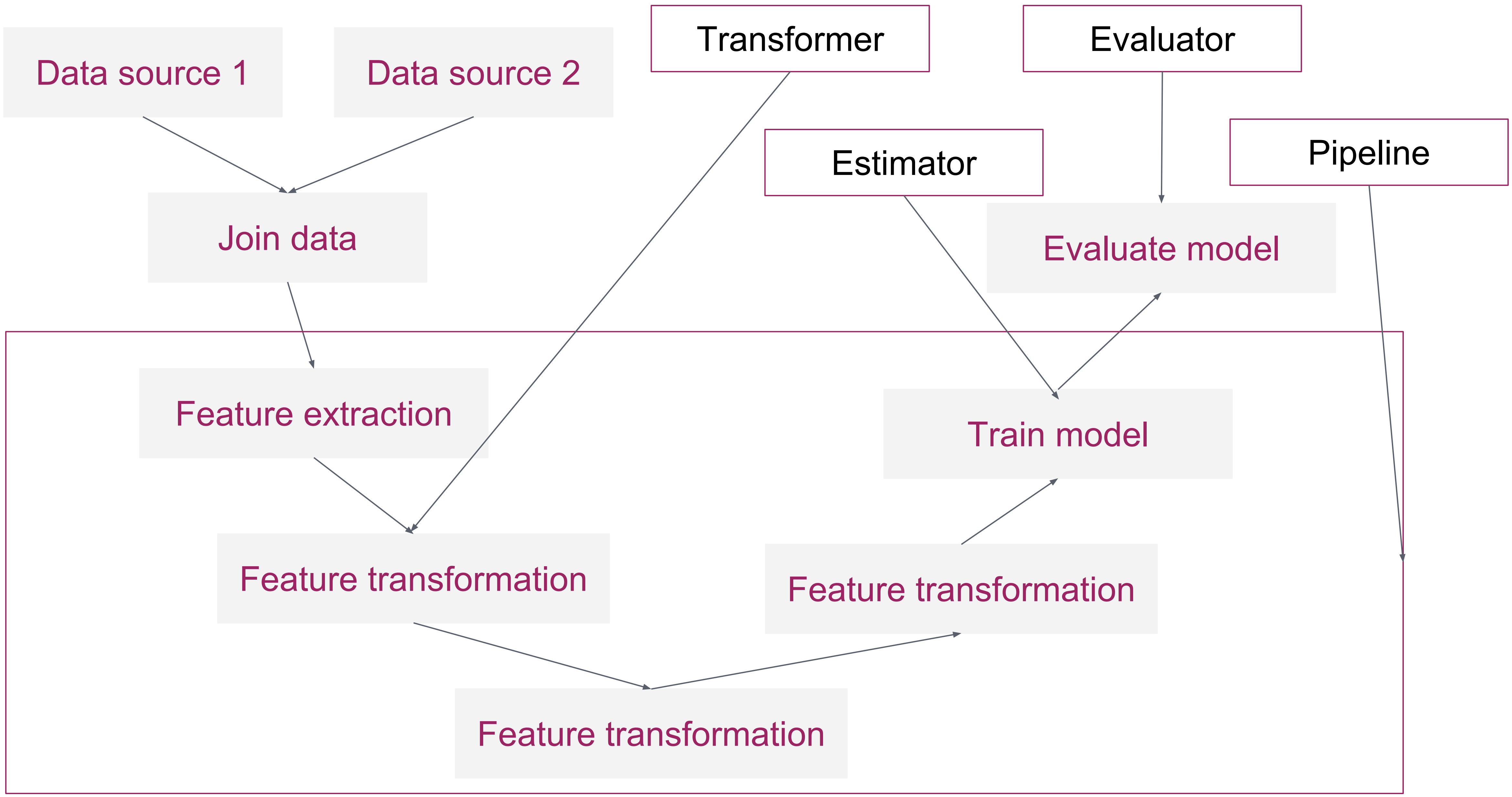

Data source 1
Data source 2
Join data
Feature extraction
Feature transformation
Feature transformation
Feature transformation
Train model


Data source 1
Data source 2
Join data
Evaluate model
Feature extraction
Train model
Feature transformation
Feature transformation
Feature transformation


Data source 1
Data source 2
Transformer
Join data
Evaluate model
Feature extraction
Train model
Feature transformation
Feature transformation
Feature transformation


Data source 1
Data source 2
Transformer
Estimator
Join data
Evaluate model
Feature extraction
Train model
Feature transformation
Feature transformation
Feature transformation


Data source 1
Data source 2
Transformer
Evaluator
Estimator
Join data
Evaluate model
Feature extraction
Train model
Feature transformation
Feature transformation
Feature transformation


Data source 1
Data source 2
Transformer
Evaluator
Estimator
Pipeline
Join data
Evaluate model
Feature extraction
Train model
Feature transformation
Feature transformation
Feature transformation

Transformer

- Transforms the DataFrame
- Adds a new column to the DataFrame by transforming another column
- Each transformer has a **transform** method
- UnaryTransformer
  - transforms exactly one column and appends one column to the DataFrame

Let’s see some examples that will be useful in the hands-on part

Tokenizer

- Transformer
- converts to lower-case
- splits text on whitespace
- adds ArrayType column

Tokenizer

- Transformer
- converts to lower-case
- splits text on whitespace
- adds ArrayType column

```
tokenizer = Tokenizer(inputCol='message', outputCol='words')

tokenized_df = tokenizer.transform(data)
```

Tokenizer

- Transformer
- converts to lower-case
- splits text on whitespace
- adds ArrayType column

```
tokenizer = Tokenizer(inputCol='message', outputCol='words')

tokenized_df = tokenizer.transform(data)
```

id	message
1	'This is my text'
2	'How are you?'

id	message	words
1	'This is my text'	['this', 'is', 'my', 'text']
2	'How are you?'	['how', 'are', 'you?']

StopWordsRemover

- Transformer
- removes words specified in stopWords list
- important params
  - stopWords

```
remover = StopWordsRemover(inputCol=’words’, outputCol=’noStopWords’, stopWords = [‘this’, ‘that’])

stopWordsRemoved_df = remover.transform(tokenized_df)
```

Normalizer

- Transformer
- rescales a vector to size 1

```
normalizer = Normalizer(inputCol='inputVec', outputCol='normVec')

normalized_df = normalizer.transform(data)
```

OneHotEncoder

- Transformer
- to handle categorical features

id	color
1	red
2	blue
3	black
4	green

OneHotEncoder

- Transformer
- to handle categorical features

id	color
1	red
2	blue
3	black
4	green

1	0	0
0	1	0
0	0	1
0	0	0

OneHotEncoder

- Transformer
- to handle categorical features

id	color
1	red
2	blue
3	black
4	green

1	0	0
0	1	0
0	0	1
0	0	0

Sparse Vector

3	[0]	[1.0]
3	[1]	[1.0]
3	[2]	[1.0]
3	[]	[]

VectorAssembler

- Transformer
- collects all columns to a single vector that can be used as input for learning algorithm

```
vectorAssembler = VectorAssembler(inputCols=['height', 'weight', 'price', 'age'], outputCol='features')

assembled_df = vectorAssembler.transform(data)
```

VectorAssembler

- Transformer
- collects all columns to a single vector that can be used as input for learning algorithm

```
vectorAssembler = VectorAssembler(inputCols=['height', 'weight', 'price', 'age'], outputCol='features')

assembled_df = vectorAssembler.transform(data)
```

height	weight	price	age
10	8	0	1

height	weight	price	age	features
10	8	0	1	[10.0, 8.0, 0.0, 1.0]

Estimator

- Takes in a DataFrame and returns a model which is a transformer
- Represents a learning algorithm
- Implements fit method
- Each Estimator has a corresponding model that is a transformer

CountVectorizer

- Estimator
- creates CountVectorizerModel
- creates a vocabulary from documents
- computes term frequency (TF) for each word in the vocabulary
- you can set the vocabulary size
- parameters
  - vocabSize
  - minDF
  - maxDF

CountVectorizer

```
countVectorizer = CountVectorizer(inputCol=words, outputCol='counts', minDF=2)
model_count = countVectorizer.fit(data)
count_df = model_count.transform(data)
```

id	message	words
1	‘This is my text’	[‘this’, ‘is’, ‘my’, ‘text’]
2	‘How is my dog?’	[‘how’, ‘is’, ‘my’, ‘dog?’]

id	message	words	counts
1	‘This is my text’	[‘this’, ‘is’, ‘my’, ‘text’]	(2, [0, 1], [1.0, 1.0])
2	‘How is my dog?’	[‘how’, ‘is’, ‘my’, ‘dog?’]	(2, [0, 1], [1.0, 1.0])

```
model_count.vocabulary
```

StringIndexer

- Estimator
- to handle categorical features

```
stringIndex = StringIndexer(inputCol='color', outputCol='indexedColor')
indexed_df = stringIndexer.fit(data).transform(data)
```

id	color
1	red
2	blue
3	red
4	green

id	color	indexedColor
1	red	0.0
2	blue	1.0
3	red	0.0
4	green	2.0

IDF

- Estimator
- computes inverse document frequency * term frequency

Number of documents

$$\text{IDF} = \ln \frac{|D| + 1}{\text{DF} + 1}$$

Document frequency (in how many documents the term is)

IDF

'this is my cat'

'this is my dog - this dog is hungry'

'my dog is in this car'

$$IDF(dog) = \ln \frac{|D| + 1}{DF + 1} = \ln \frac{3 + 1}{2 + 1} = 0.29$$

$$IDF(cat) = \ln \frac{|D| + 1}{DF + 1} = \ln \frac{3 + 1}{1 + 1} = 0.69$$

$$IDF(this) = \ln \frac{|D| + 1}{DF + 1} = \ln \frac{3 + 1}{3 + 1} = 0$$

IDF

‘this is my cat’

‘this is my dog - this dog is hungry’

‘my dog is in this car’

$$IDF(dog) = \ln \frac{|D| + 1}{DF + 1} = \ln \frac{3 + 1}{2 + 1} = 0.29$$

$$IDF(cat) = \ln \frac{|D| + 1}{DF + 1} = \ln \frac{3 + 1}{1 + 1} = 0.69$$

$$IDF(this) = \ln \frac{|D| + 1}{DF + 1} = \ln \frac{3 + 1}{3 + 1} = 0$$

TFIDF(dog, 1) = IDF * TF = 0.29 * 0 = 0

TFIDF(dog, 2) = IDF * TF = 0.29 * 2 = 0.58

TFIDF(dog, 3) = IDF * TF = 0.29 * 1 = 0.29

RandomForestClassifier

- Estimator
- supervised learning algorithm used for classification
- params: numTrees, maxDepth

```
rf = RandomForestClassifier(featuresCol='features', labelCol='label')
model = rf.fit(train_data)
predictions = model.transform(test_data)
```

RandomForestClassifier

- Estimator
- supervised learning algorithm used for classification
- params: numTrees, maxDepth

```
(train_data, test_data) = data.randomSplit([0.7, 0.3])
```

```
rf = RandomForestClassifier(featuresCol=’features’, labelCol=’label’)
model = rf.fit(train_data)
predictions = model.transform(test_data)
```

LogisticRegression

- Estimator
- supervised learning algorithm used for binary classification

```
lr = LogisticRegression(featuresCol='features', labelCol='label')
model = lr.fit(train_data)
predictions = model.transform(test_data)
```

KMeans

- Estimator
- unsupervised learning algorithm used for data clustering
- requires to set number of clusters k

```
kmeans = KMeans(featuresCol='features', k=4)
model = kmeans.fit(data)
predictions = model.transform(data)
```

LDA (Latent Dirichlet Allocation)

- Estimator
- unsupervised learning algorithm used for topic modelling
- requires to set number of clusters k
- assumes that each document is composed of multiple topics
- each topic is characterized by some words

```
lda = LDA(featuresCol='features', k=4)
model = lda.fit(data)
predictions = model.transform(data)
```

Pipeline

- allows to chain transformers and estimators

```
tokenizer = Tokenizer(inputCol='message', outputCol='words')
stopWordsRemover = StopWordsRemover(inputCol='words', outputCol='noStopWords')
countVectorizer = CountVectorizer(inputCol='noStopWords', outputCol='wordCounts')
```

```
df1 = tokenizer.transform(data)
df2 = stopwordsRemover.transform(df1)
vectorizerModel = countVectorizer.fit(df2)
```

Pipeline

- allows to chain transformers and estimators

```
tokenizer = Tokenizer(inputCol='message', outputCol='words')
stopWordsRemover = StopWordsRemover(inputCol='words', outputCol='noStopWords')
countVectorizer = CountVectorizer(inputCol='noStopWords', outputCol='wordCounts')
```

```
df1 = tokenizer.transform(data)
df2 = stopwordsRemover.transform(df1)
vectorizerModel = countVectorizer.fit(df2)
```

- these are your transformers & estimators
- this is one possible way to run it
- or you can use Pipeline

```
model = Pipeline(stages=[tokenizer, stopWordsRemover, countVectorizer]).fit(data)
```

Evaluator

- Allows you to evaluate your model
- method evaluate()
- BinaryClassificationEvaluator
- RegressionEvaluator
- MulticlassClassificationEvaluator
- ClusteringEvaluator

BinaryClassificationEvaluator

- method evaluate()
    - takes DataFrame with at least 2 columns: label, rowPrediction
- metricName: areaUnderROC, areaUnderPR

```
pipeline = Pipeline(stages=[indexer, assembler, rf_cls])
model = pipeline.fit(data)
```

```
predictions = model.transform(data)
evaluator = BinaryClassificationEvaluator(labelCol='label', metricName='areaUnderROC')
evaluator.evaluate(predictions)
```

Hyperparameter tuning

- Gridsearch
- Allows you to train the model for all combinations of specified parameters
  - Selects the best model based on the evaluation metric

Hyperparameter tuning

```
rf = RandomForestClassifier(labelCol='label', featuresCol='features')
pipeline = Pipeline(stages[..., rf])
```

```
paramGrid = ParamGridBuilder() \
    .addGrid(rf.maxDepth, [3, 5, 8]) \
    .addGrid(rf.numTrees, [50, 100, 150]) \
    .build()
```

```
evaluator = BinaryClassificationEvaluator(labelCol='label', metricName='areaUnderROC')
model = CrossValidator(estimator=pipeline, evaluator=evaluator, estimatorParamMaps=paramGrid).fit(data)
best_model = model.bestModel
```

Time for hands-on
Lab II

https://mlprague2019.cloud.databricks.com

- If you don't have access:
    - **mlprague2019@socialbakers.com**

Part III

Advanced data analysis with GraphFrames

- Graph processing in Spark (10 mins)
- Demo in the notebook (if there is time, 10 mins)

Graph processing in Spark

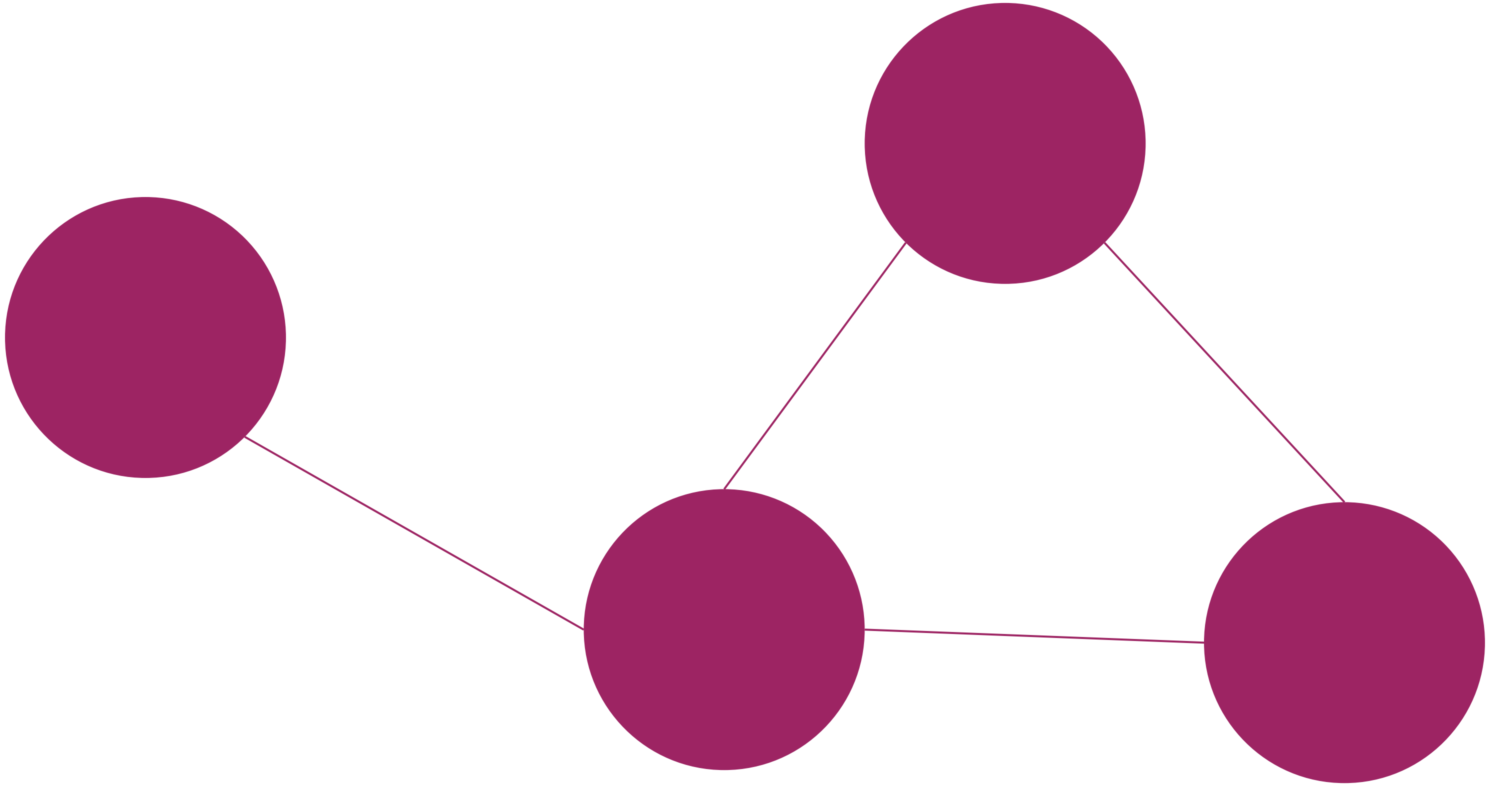
- GraphX – native library for graph processing – provides only RDD API
- GraphFames – DataFrame based API
    - external library
    - calls GraphX under the hood

Why graph processing?

- If data has a network or graph structure it may give you different perspective on the data

What is a graph?

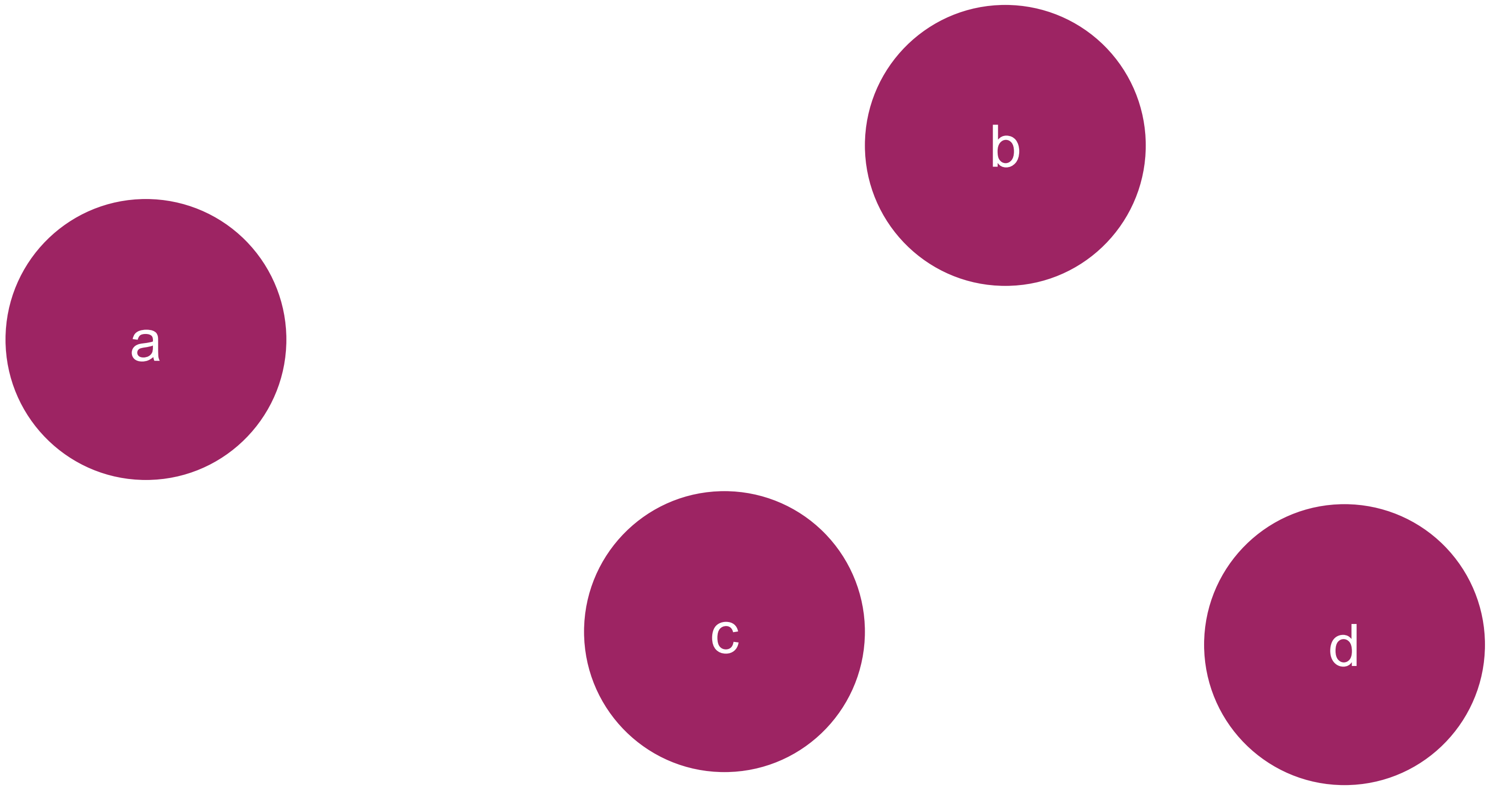
- Structure composed of
  - Vertices
  - Edges



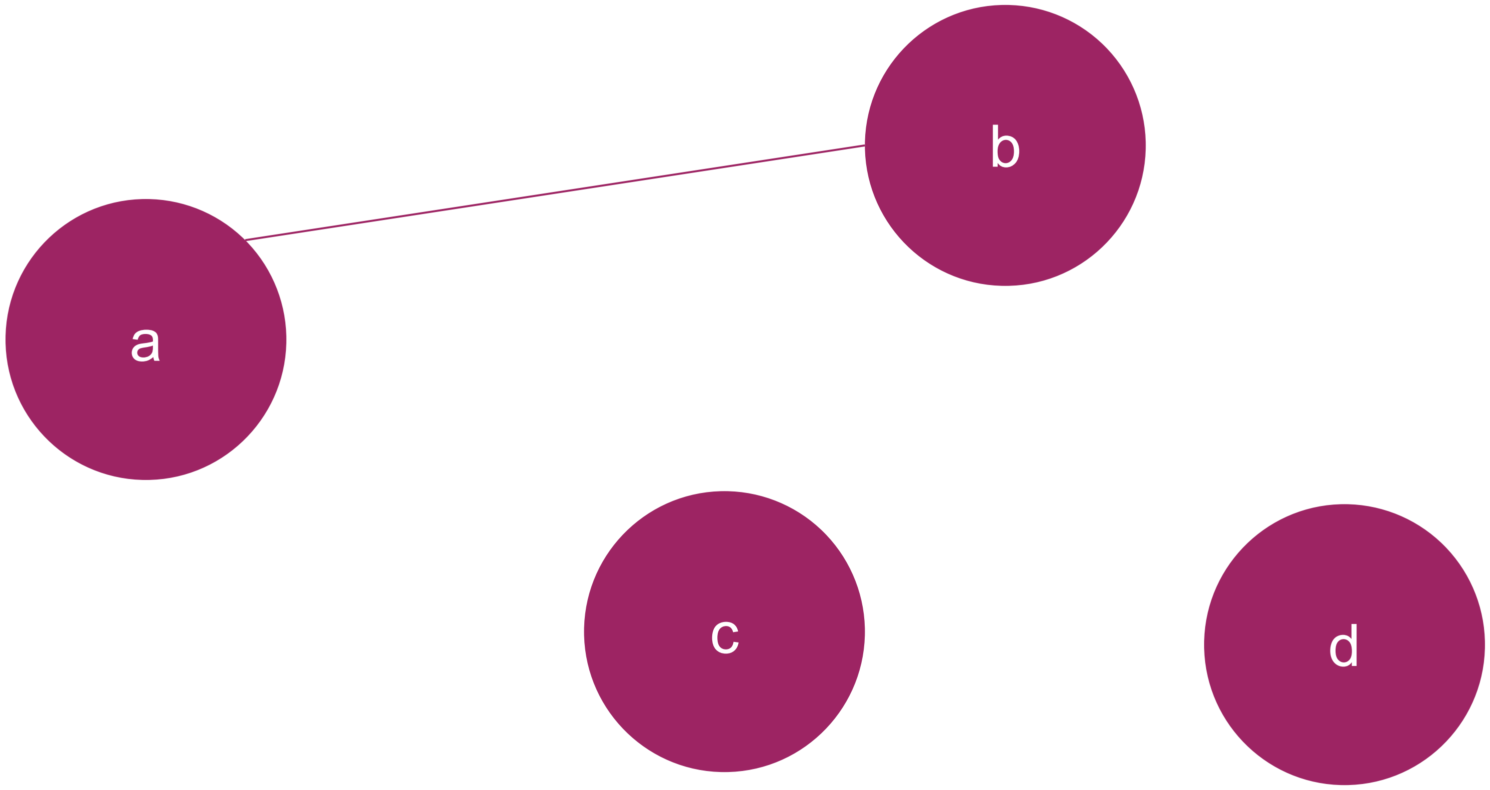
Affinities – user_pages data

- For each page we have a list of users that interacted with this page
- We can model the data as a graph
  - Each page is a vertex
  - The vertices are connected with edge if the pages were visited by the same user

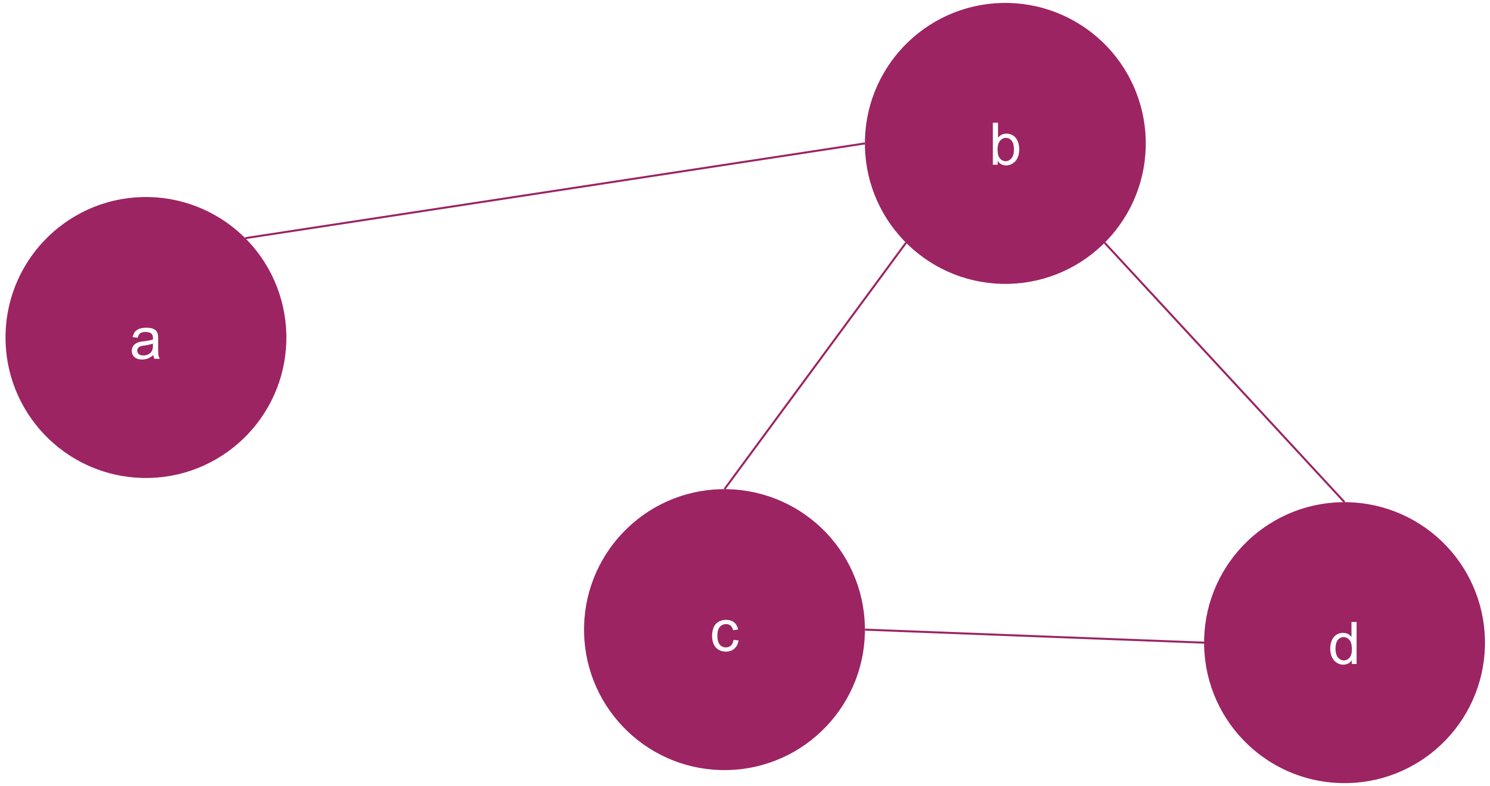
user_id	page_id
1	a
1	b
2	b
2	c
2	d
3	a
3	c



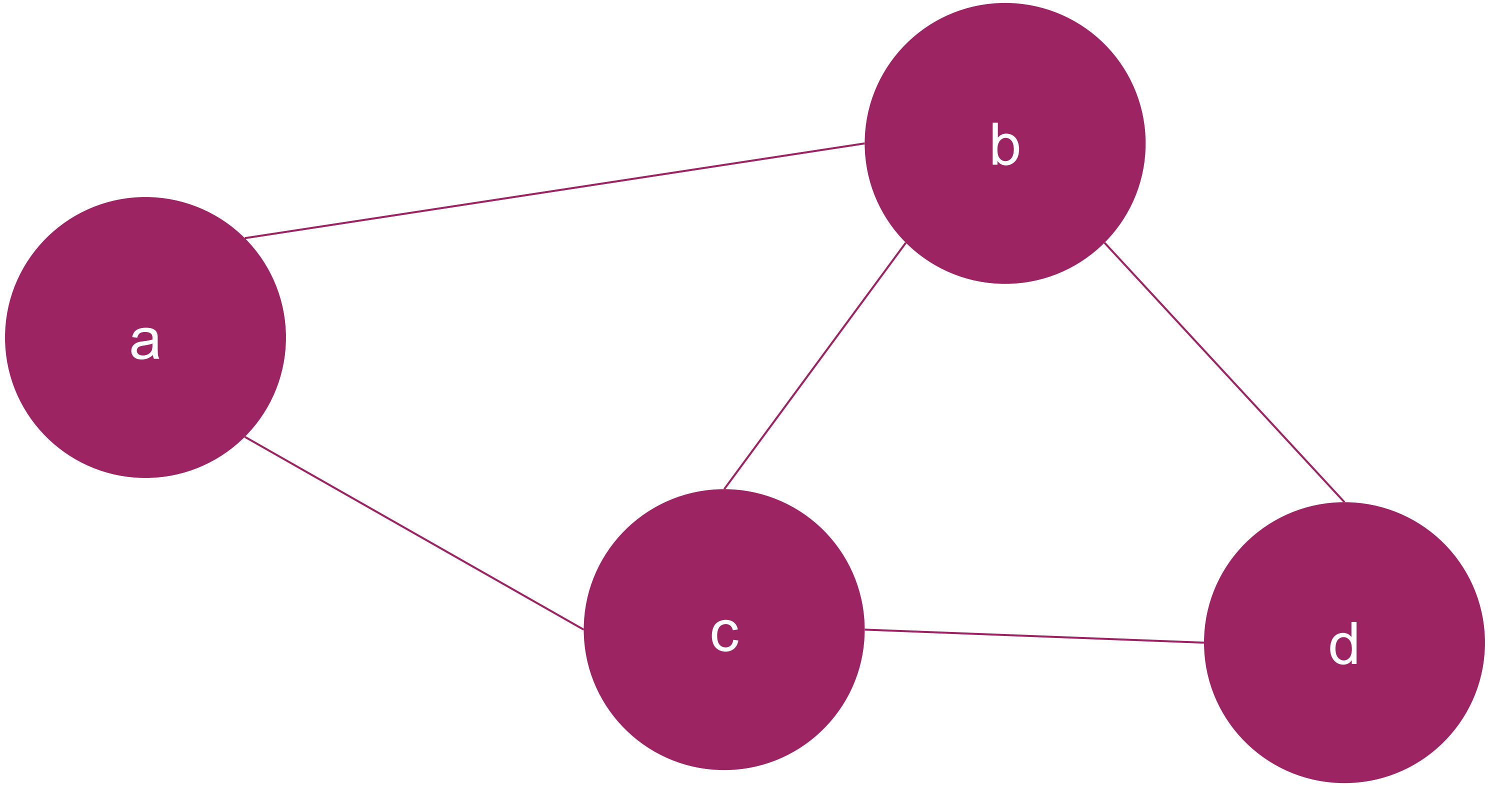
user_id	page_id
1	a
1	b
2	b
2	c
2	d
3	a
3	c



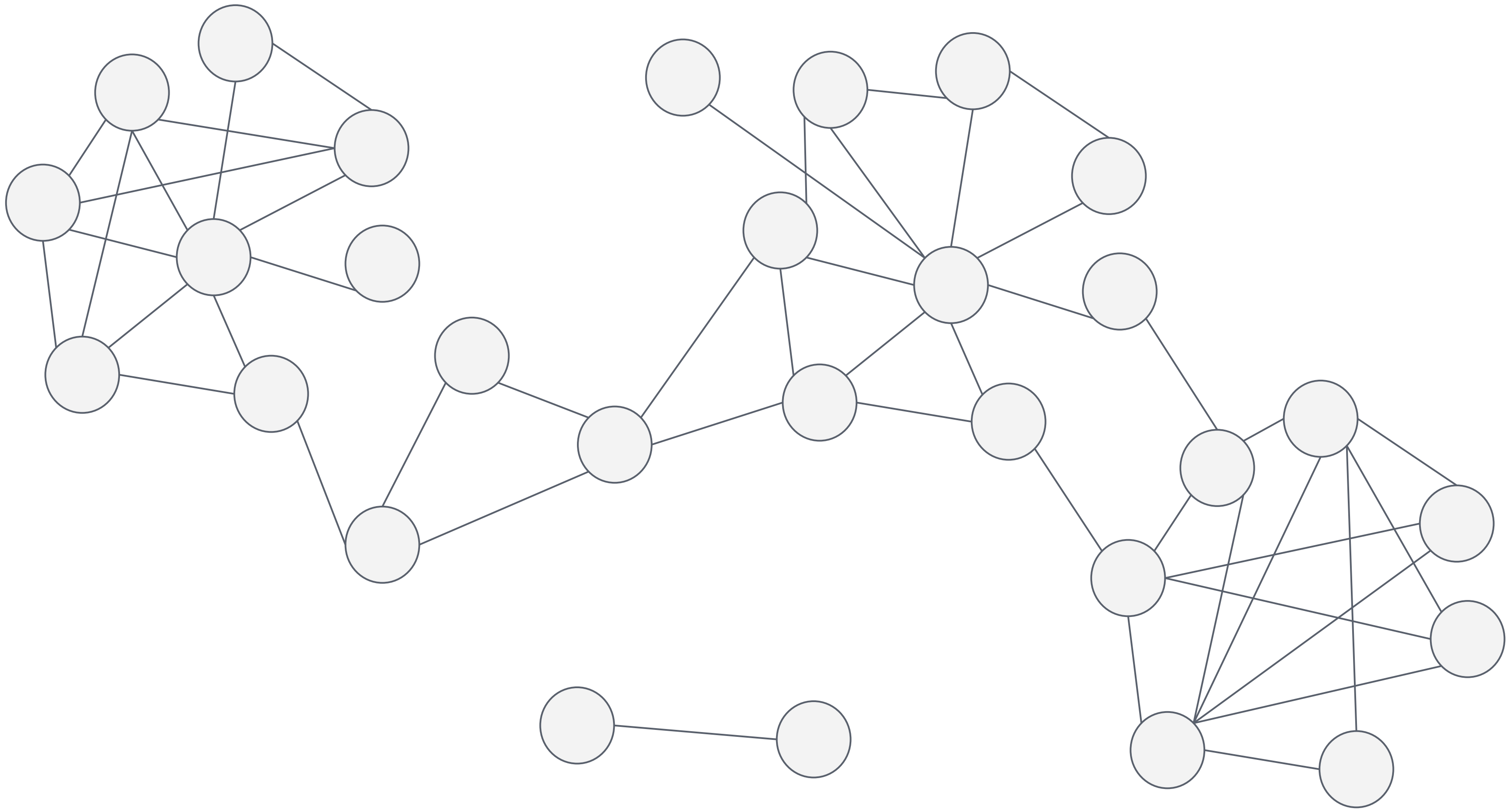
user_id	page_id
1	a
1	b
2	b
2	c
2	d
3	a
3	c



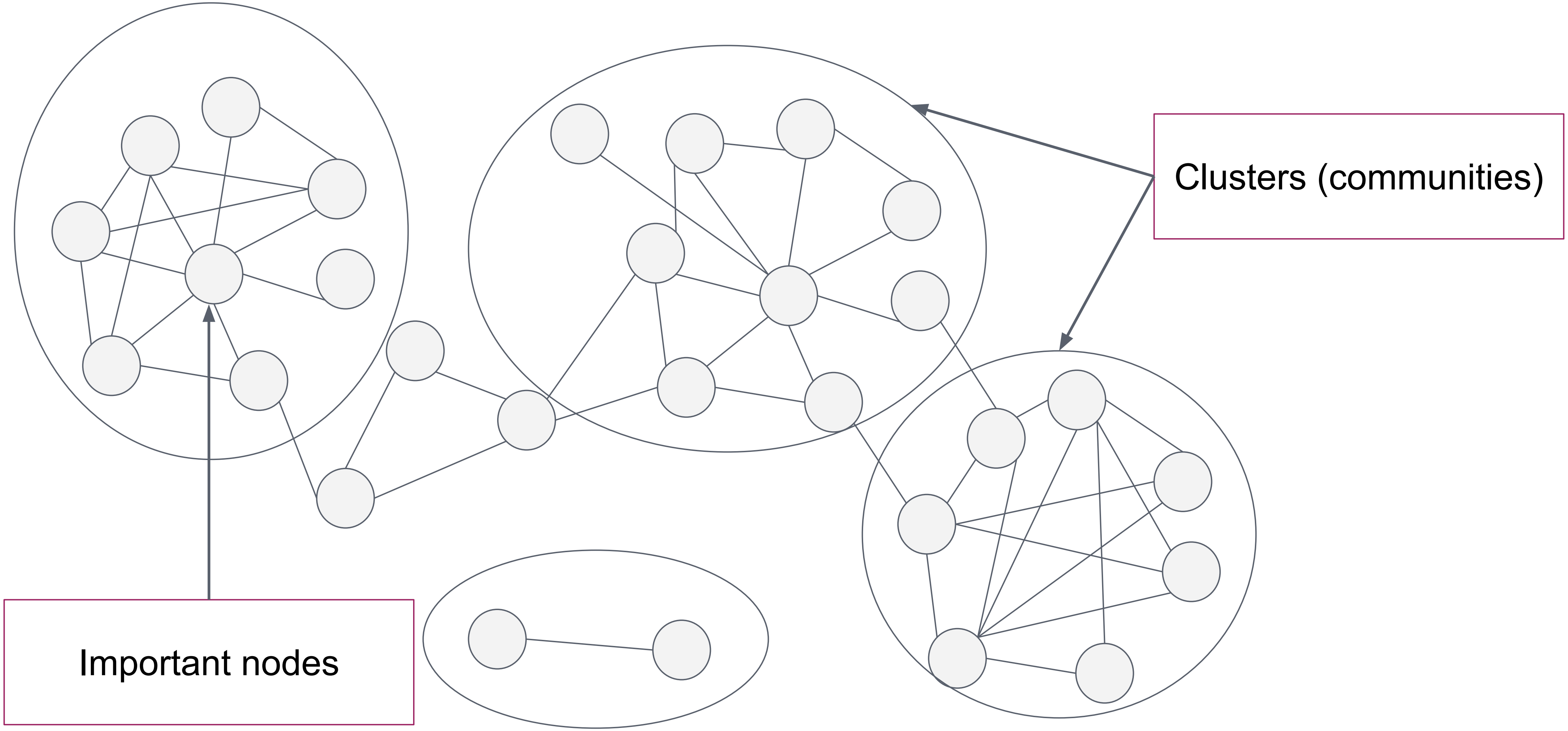
user_id	page_id
1	a
1	b
2	b
2	c
2	d
3	a
3	c



What can you find out from the graph?



What can you find out from the graph?



Algorithms offered by GraphFrames

- Label Propagation (LPA)
  - standard community detection algorithm
  - computationally inexpensive
  - convergence is not guaranteed
  - it can end up with trivial solution (all nodes in a single community)

Algorithms offered by GraphFrames

- Page Rank (PR)
  - detects important nodes in the network

Constructing the graph in GraphFrames

- Create two DataFrames
  - vertices
    - only one column named ‘id’ with all page_ids
  - edges
    - two columns: ‘src’, ‘dst’

```
graph = GraphFrame(vertices, edges)
```

How to create edges from user_pages data

user_id	page_id
1	a
1	b
2	b
2	c
2	d
3	a
3	c

- just make a self-join on user_id
- rename page_id to 'src' and 'dst'
- filter out records in which src == dst
    - this corresponds to the case where page is connected with itself

Demo

Let's see some analysis in the notebook

Conclusion

Today we explored:

- DataFrame API for interactive data analytics
- ML Pipelines for machine learning
- GraphFrames for graph processing

Thank you for your attention!