Generic RDD Transformations

- `map` and `flatMap` apply a given function to each RDD element **independently**
- `map` transforms an RDD of **length *n*** into another RDD of **length *n***.
- `flatMap` allows returning **0, 1 or more elements** from map function.

```
In [1]: data = sc.parallelize(range(10))
In [2]: squared_data = data.map(lambda x: x * x)
In [3]: print(squared_data.collect()) # this is an action
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [4]: squared_cubed_data_1 = data.map(lambda x: ( x * x, x * x * x))
In [5]: print(squared_cubed_data_1.collect()) # this is an action
[(0, 0), (1, 1), (4, 8), (9, 27), (16, 64), (25, 125), (36, 216), (49, 343),
(64, 512), (81, 729)]

In [6]: squared_cubed_data_2 = data.flatMap(lambda x: ( x * x, x * x * x))
In [7]: print(squared_cubed_data_2.collect()) # this is an action
[0, 0, 1, 1, 4, 8, 9, 27, 16, 64, 25, 125, 36, 216, 49, 343, 64, 512, 81,
729]
```

Generic RDD Transformations

- `sortBy` sorts an RDD
- `union` performs the **merging** of RDDs
- `intersection` performs the **set intersection** of RDD

```
In [1]: words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
In [2]: sorted_words = words.sortBy(lambda w: len(w))
In [3]: print(sorted_words.collect()) # this is an action
['di', 'nel', 'del', 'vita', 'mezzo', 'cammin', 'nostra']
In [4]: data1 = sc.parallelize(range(0,7))
In [5]: data2 = sc.parallelize(range(3,10))
In [6]: union = data1.union(data2)
[0, 1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7, 8, 9]
In [7]: print(union.collect()) # this is an action
In [8]: intersection = data1.intersection(data2)
In [9]: print(intersection.collect()) # this is an action
[3, 4, 5, 6]
```

Key-Value RDD Transformations

- In a `(k,v)` pair, `k` is the **key**, and `v` is the **value**
- To create a key-value RDD:
  - `map` over your current RDD to a basic key-value structure.
  - Use the `keyBy` to create a key from the current value.
  - Use the `zip` to zip together two RDD.

```
In [1]: words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
In [2]: keywords1 = words.map(lambda w: (w.upper(), 1))
In [3]: print(keywords1.collect()) # this is an action
[('NEL', 1), ('MEZZO', 1), ('DEL', 1), ('CAMMIN', 1), ('DI', 1), ('NOSTRA', 1), ('VITA', 1)]
In [4]: keywords2 = words.keyBy(lambda w: w[0].upper())
In [5]: print(keywords2.collect()) # this is an action
[('N', 'nel'), ('M', 'mezzo'), ('D', 'del'), ('C', 'cammin'), ('D', 'di'), ('N', 'nostra'), ('V', 'vita')]
In [6]: numbers = sc.parallelize(range(7))
In [7]: keywords3 = words.zip(numbers)
In [8]: print(keywords3.collect()) # this is an action
[('nel', 0), ('mezzo', 1), ('del', 2), ('cammin', 3), ('di', 4), ('nostra', 5), ('vita', 6)]
```

Key-Value RDD Transformations

- `keys` and `values` extract keys and values from the RDD, respectively
- `lookup` looks up the **list of values** for a particular key in an RDD

```
In [1]: words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
In [2]: keywords = words.keyBy(lambda w: w[0])
# [('n', 'nel'), ('m', 'mezzo'), ('d', 'del'), ('c', 'cammin'), ('d', 'di'), ('n', 'nostra'), ('v', 'vita')]
In [3]: k = keywords.keys()
# ['n', 'm', 'd', 'c', 'd', 'n', 'v']
In [4]: v = keywords.values()
# ['nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']
In [5]: look = keywords.lookup("n")
In [6]: print(look)
['nel', 'nostra']
```

Key-Value RDD Transformations

- `reduceByKey` combines values with the same key
  - Takes a **function** as input and uses it to **combine values** of the same key
- `sortByKey` returns an RDD sorted by the key

```
In [1]: words = sc.parallelize("fare o non fare non esiste provare".split(" "))
In [2]: wordcount = words.map(lambda w: (w, 1)).reduceByKey(lambda x, y: x + y)
In [3]: print(wordcount.collect()) # this is an action
[('provare', 1), ('fare', 2), ('non', 2), ('esiste', 1), ('o', 1)]
In [4]: sorted_wordcount = wordcount.sortByKey()
In [5]: print(sorted_wordcount.collect()) # this is an action
[('esiste', 1), ('fare', 2), ('non', 2), ('o', 1), ('provare', 1)]
```

Key-Value RDD Transformations

- `join` performs an inner-join on the key
- Other types of join:
  - `fullOuterJoin`
  - `leftOuterJoin, rightOuterJoin`
  - `cartesian`

```
In [1]: cars = sc.parallelize(["Ferrari", "Porsche", "Mercedes"])
In [2]: colors = sc.parallelize(["red", "black", "pink"])
In [3]: joined = cars.cartesian(colors)
In [4]: print(joined.collect())
[('Ferrari', 'red'), ('Ferrari', 'black'), ('Ferrari', 'pink'), ('Porsche',
'red'), ('Porsche', 'black'), ('Porsche', 'pink'), ('Mercedes', 'red'),
('Mercedes', 'black'), ('Mercedes', 'pink')]
In [5]: cars = sc.parallelize([(1,"Ferrari"), (1, "Porsche"), (2, "Mercedes")])
In [6]: colors = sc.parallelize([(1, "red"), (2, "black"), (3, "pink")])
In [7]: joined = cars.join(colors)
In [8]: print(joined.collect())
[(1, ('Ferrari', 'red')), (1, ('Porsche', 'red')), (2, ('Mercedes', 'black'))]
```

Word Count (I)

1. Load "`comedies.txt`" text file into Spark
2. Transform the lines RDD into a words RDD
3. Transform each `word` into a `(word, 1)` pair
4. Reduce words by key to sum up word occurrences
5. Save results as text file

Word Count (II)

```
$> cd $HOME/hpsa
$> pyspark
...

In [1]: text   = sc.textFile("data/comedies.txt")
In [2]: words  = text.flatMap(lambda x: x.split(" "))
In [3]: ones   = words.map(lambda w: ( w, 1 ))
In [4]: counts = ones.reduceByKey(lambda x, y: x + y)
In [5]: counts.saveAsTextFile("data/comedies_wordcount.txt")
```

Bigram Count (I)

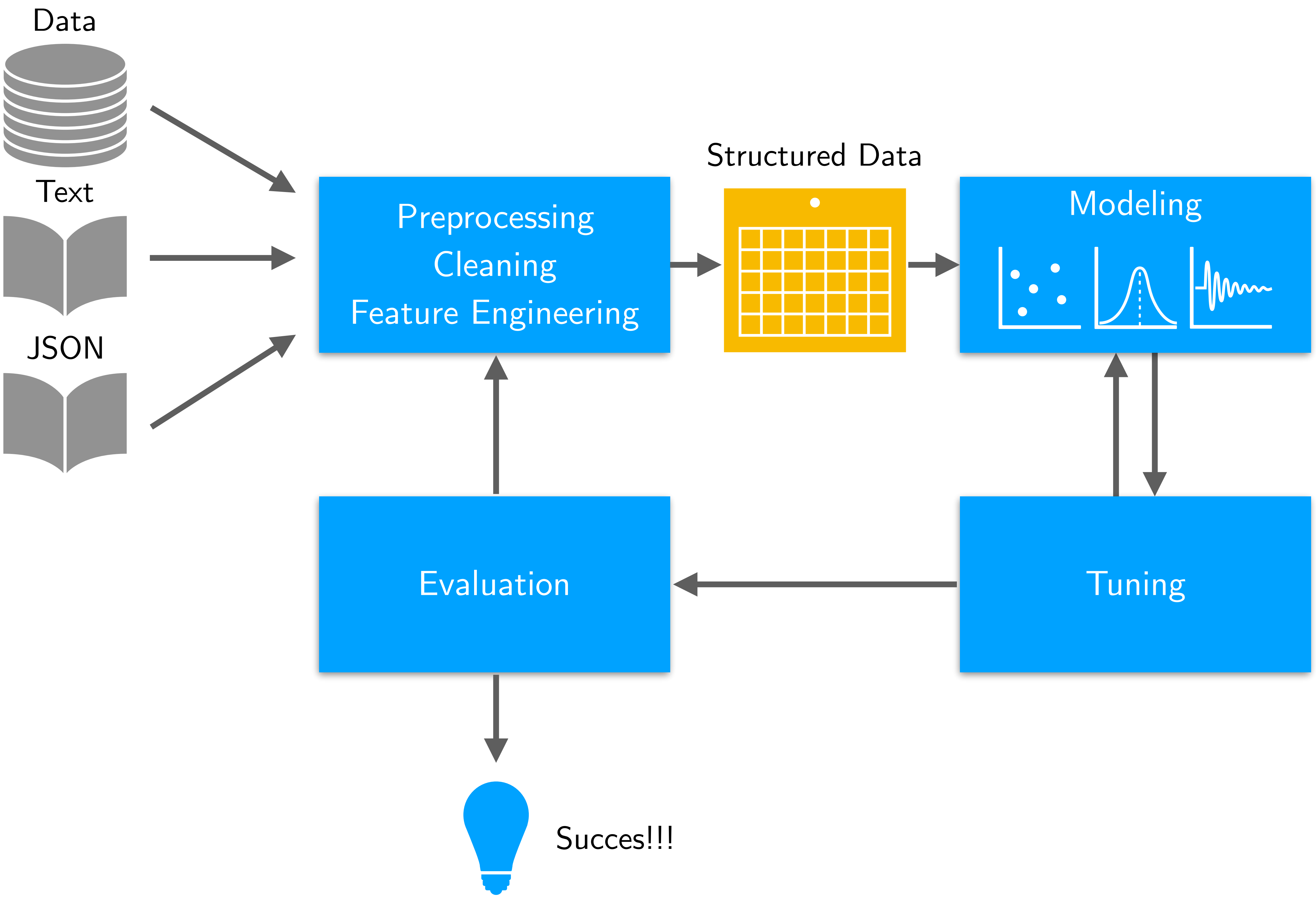
1. Define a function extracting all bigrams from a string of words.
2. Load "`comedies.txt`" text file into Spark
3. Transform the lines RDD into a bigrams RDD
4. Transform each bigram into a (bigram, 1) pair
5. Reduce bigrams by key to sum up bigram occurrences
6. Save results as text file

Bigram Count (II)

```
def create_bigrams(line):
    pairs = []
    words = line.lower().split()
    for i in range(len(words) -1):
        pairs.append(words[i] + "_" + words[i + 1])
    return pairs

text = sc.textFile("data/comedies.txt")
bigrams = text.flatMap(create_bigrams)
ones = bigrams.map(lambda b: ( b, 1 ))
counts = ones.reduceByKey(lambda x, y: x + y)
counts.saveAsTextFile("data/comedies_bigramscount.txt")
```

Data Analytics Process



Machine Learning with Spark

- Spark provides support for **statistics** and **machine learning**
- **Supervised learning**
  - Using labeled historical data and training a model to predict the values of those labels based on various features of the data points.
  - **Classification** (categorical values)
    - E.g., predicting disease, classifying images, ...
  - **Regression** (continuous values)
    - E.g., predicting sales, predicting height, ...
- **Unsupervised learning**
  - No label to predict.
  - Trying to **find patterns** or discover the underlying structure in a given set of data.
    - E.g., Clustering, anomaly detection, ...

MLlib

- **MLlib** is a package built **on Spark**
- It provides **interfaces** for:
  - **Gathering** and **cleaning** data
  - **Feature engineering** and feature selection
  - **Training** and **tuning** large-scale supervised and unsupervised machine learning models
  - Using those models in **production**
- MLlib consists of **two packages**
  - `org.apache.spark.mllib`: uses RDDs
    - It is in maintenance mode (only receives bug fixes, not new features)
  - `org.apache.spark.ml`: uses DataFrames
    - Offers a high-level interface for building machine learning pipelines

Why MLlib?

- Many tools for performing machine learning on **a single machine**, e.g., *scikit-learn* and *TensorFlow*
- These single-machine tools are usually **complementary** to MLlib
- Take advantage of Spark, when you hit **scalability issues**
  - Use Spark for **preprocessing** and **feature generation**, before giving data to single-machine learning libraries.
  - Use Spark, when input **data or model size** become too difficult to put on one machine.

MLlib Data Types

- MLlib contains a few specific data types, located in the `pyspark.mllib` package
- `Vector`
  - A **mathematical** vector.
  - **Dense** vectors, where every entry is stored
  - **Sparse** vectors, where only the nonzero entries are stored to save space
- `LabeledPoint`
  - A labeled data point for supervised learning algorithms such as classification and regression.
  - Includes a **feature vector** and a **label** (which is a floating-point value).
- `Model` classes
  - Each `Model` is the result of a **training algorithm**, and typically has a `predict()` method for applying the model to a new data point or to an RDD of new data points.