Chapter 25

Preprocessing and Feature Engineering

Outline

- Transformers
- High-Level Transformers
- Working with Continuous Features
- Working with Categorical Features
- Text Data Transformers
- Feature Manipulation
- Feature Selection
- Advanced Topics

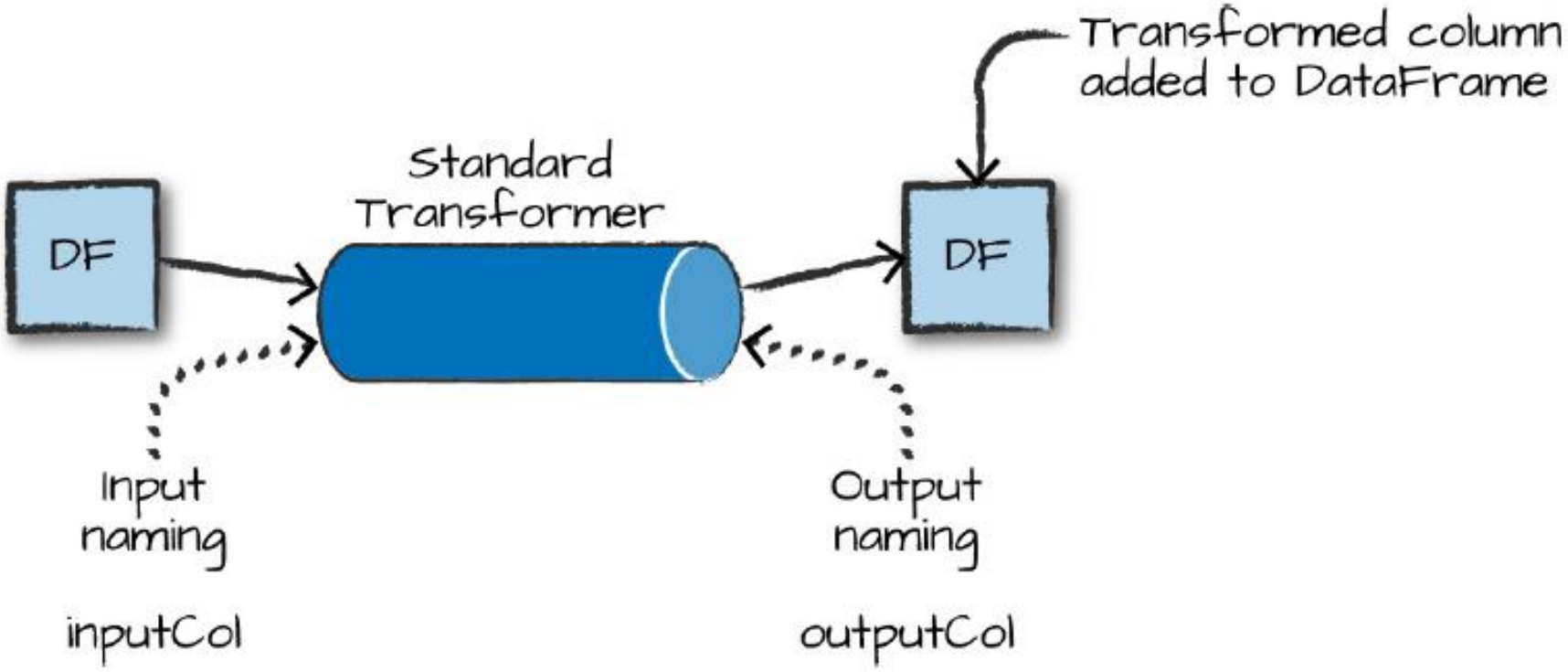
Transformers

- Transformers are primarily used in preprocessing or feature generation.
- Transformers are functions that convert raw data in different ways. This is used to create a new interaction variable (from two other variables), to normalize a column, or to simply turn it into a Double to be input into a model.
  - Spark provides a number of transformers as part of the org.apache.spark.ml.feature package. New transformers are constantly popping up in Spark. The most up-to-date information can be found on the Spark documentation site.(https://spark.apache.org/docs/latest/ml-features.html)

```scala
// in Scala
val sales = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("/data/retail-data/by-day/*.csv")
  .coalesce(5)
  .where("Description IS NOT NULL")
val fakeIntDF = spark.read.parquet("/data/simple-ml-integers")
var simpleDF = spark.read.json("/data/simple-ml")
val scaleDF = spark.read.parquet("/data/simple-ml-scaling")
```

Transformers (cont.)

- This figure is a simple illustration. On the left is an input DataFrame with the column to be manipulated. On the right is the input DataFrame with a new column representing the output transformation.



Transformers (cont.)

- Tokenizer : It tokenizes a string, splitting on a given character

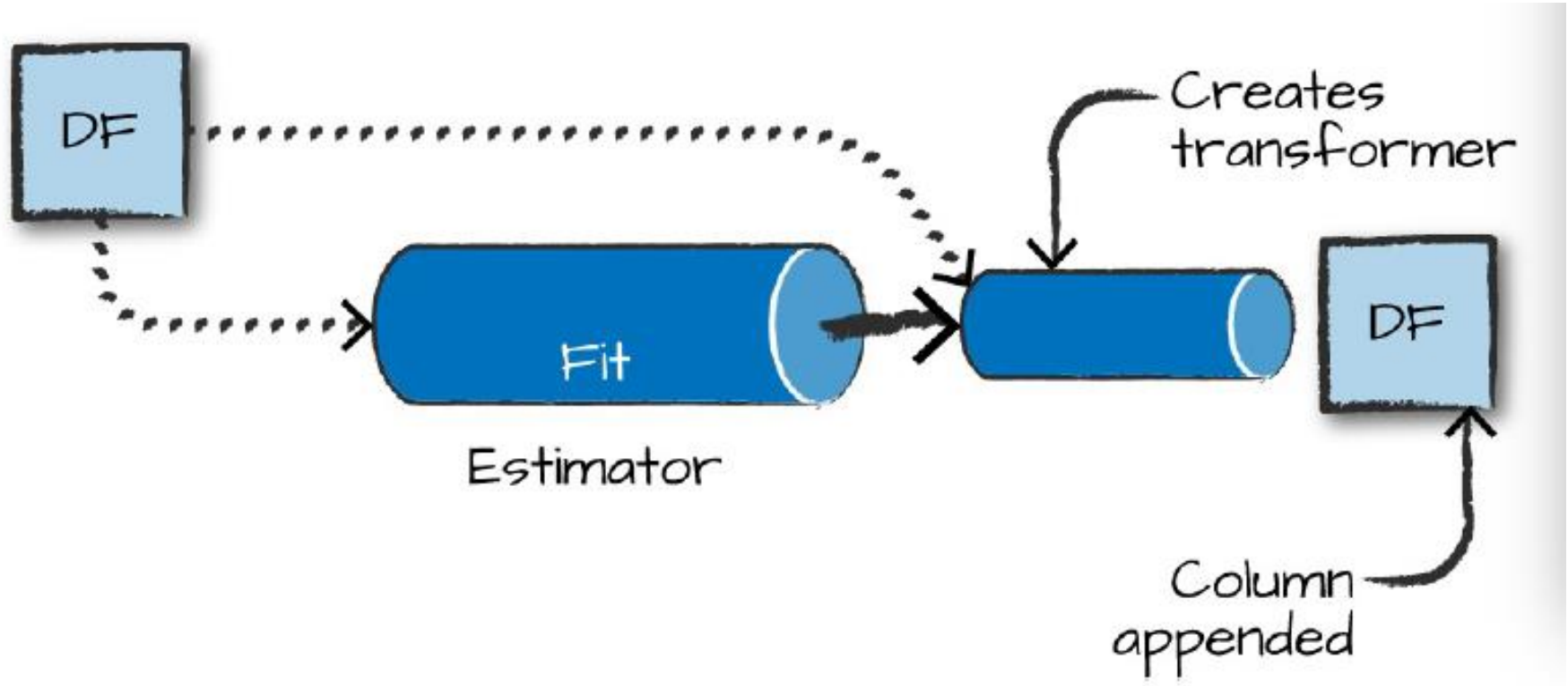
```
// in Scala
import org.apache.spark.ml.feature.Tokenizer
```

```
val tkn = new Tokenizer().setInputCol("Description")
tkn.transform(sales.select("Description")).show(false)
```

```
+-----------------------------------+------------------------------------------+
|Description                        |tok_7de4dfc81ab7__output                  |
+-----------------------------------+------------------------------------------+
|RABBIT NIGHT LIGHT                 |[rabbit, night, light]                    |
|DOUGHNUT LIP GLOSS                 |[doughnut, lip, gloss]                    |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion]  |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set, brown]  |
+-----------------------------------+------------------------------------------+
```

Estimators for Preprocessing

- This figure is a simple illustration of an estimator fitting to a particular input dataset, generating a transformer that is then applied to the input dataset to append a new column (of the transformed data).



Estimators for Preprocessing (cont.)

- An example of estimator is the *StandardScaler*, which scales your input column according to the range of values in that column to have a zero mean and a variance of 1 in each dimension. For that reason it must first perform a pass over the data to create the transformer.

```
// in Scala
import org.apache.spark.ml.feature.StandardScaler
val ss = new StandardScaler().setInputCol("features")
ss.fit(scaleDF).transform(scaleDF).show(false)
```

```
+---+--------------+------------------------------------------------------------+
|id |features      |stdScal_d66fbeac10ea__output                                |
+---+--------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[1.1952286093343936,0.02337622911060922,-0.5976143046671968]|
...
|1  |[3.0,10.1,3.0]|[3.5856858280031805,2.3609991401715313,1.7928429140015902]  |
+---+--------------+------------------------------------------------------------+
```

High-Level Transformers

- High-level allow you to concisely specify a number of transformations in one.
- Allow you to avoid doing data manipulations or transformations one by one.
- In general, you should try to use the highest level transformers you can, in order to minimize the risk of error and help you focus on the business problem instead of the smaller details of implementation.

High-Level Transformers

- RFormula
- SQL Transformers
- VectorAssembler

RFormula

- The RFormula is the easiest transformer to use when you have “conventionally” formatted data.
- Spark borrows this transformer from the R language to make it simple to declaratively specify a set of transformations for your data.
- With this transformer, values can be either numerical or categorical and you do not need to extract values from strings or manipulate them in any way.
- The RFormula will automatically handle categorical inputs (specified as strings) by performing function called *one-hot encoding*. In brief, it converts a set of values into a set of binary columns. (we’ll discuss more later in this chapter).
- With the RFormula, numeric columns will be cast to Double. If the label column is of type String, it will be first transformed to Double with **StringIndexer**. (we’ll discuss more later this chapter).

High-Level Transformers (cont.)

- RFormula
- SQL Transformers
- VectorAssembler

SQL Transformers

- A SQLTransformer allows you to leverage Spark's vast library of SQL-related manipulations
- Any SELECT statement you can use in SQL is a valid transformation.
- The only thing you need to change is that instead of using the table name, you should just use the keyword **THIS.**
- Use SQLTransformer to formally codify the DataFrame manipulation as a preprocessing step, or try different SQL expressions for features during hyperparameter tuning.
  - Note that the output of this transformation will be appended as a column to the output DataFrame.

SQL Transformers (cont.)

- The following is a basic example of using SQLTransformer:

```scala
// in Scala
import org.apache.spark.ml.feature.SQLTransformer

val basicTransformation = new SQLTransformer()
  .setStatement("""
    SELECT sum(Quantity), count(*), CustomerID
    FROM __THIS__
    GROUP BY CustomerID
  """)

basicTransformation.transform(sales).show()
```

```python
# in Python
from pyspark.ml.feature import SQLTransformer

basicTransformation = SQLTransformer()\
  .setStatement("""
    SELECT sum(Quantity), count(*), CustomerID
    FROM __THIS__
    GROUP BY CustomerID
  """)

basicTransformation.transform(sales).show()
```

```
+-------------+--------+----------+
|sum(Quantity)|count(1)|CustomerID|
+-------------+--------+----------+
|          119|      62|   14452.0|
...
|          138|      18|   15776.0|
+-------------+--------+----------+
```

High-Level Transformers (cont.)

- RFormula
- SQL Transformers
- VectorAssembler

VectorAssembler

- The VectorAssembler helps concatenate all your features into one big vector then the user can pass it into an estimator.
- It’s used typically in the last step of a machine learning pipeline.
- This is particularly helpful if you’re going to perform a number of manipulations using a variety of transformers and need to gather all of those results together.

VectorAssembler (cont.)

```scala
// in Scala
import org.apache.spark.ml.feature.VectorAssembler
val va = new VectorAssembler().setInputCols(Array("int1", "int2", "int3"))
va.transform(fakeIntDF).show()
```

```python
# in Python
from pyspark.ml.feature import VectorAssembler
va = VectorAssembler().setInputCols(["int1", "int2", "int3"])
```

```
va.transform(fakeIntDF).show()
```

```
+----+----+----+-----------------------------------------+
|int1|int2|int3|VectorAssembler_403ab93eacd5585ddd2d__output|
+----+----+----+-----------------------------------------+
|   1|   2|   3|                            [1.0,2.0,3.0]|
|   4|   5|   6|                            [4.0,5.0,6.0]|
|   7|   8|   9|                            [7.0,8.0,9.0]|
+----+----+----+-----------------------------------------+
```

Working with Continuous Features

- Bucketing
- Scaling and Normalization
- StandardScaler

Bucketing

- The most straightforward approach to bucketing or binning is using the **Bucketizer**.
- This will split a given continuous feature into the buckets of your designation. You specify how buckets should be created via an array or list of Double values.
  - This is useful because you may want to simplify the features in your dataset or simplify their representations for interpretation later on.
  - For example, imagine you have a column that represents a person's weight and you would like to predict some value based on this information. In some cases, it might be simpler to create three buckets of "overweight," "average," and "underweight."

Bucketing(cont.)

- When specifying your bucket points, the values you pass into splits must satisfy **three** requirements:
  - The **minimum** value in your splits array must be less than the minimum value in your DataFrame.
  - The **maximum** value in your splits array must be greater than the maximum value in your DataFrame.
  - You need to specify at a minimum **three values** in the splits array, which creates **two buckets**.
    - For example, setting splits to 5.0, 10.0, 250.0 on **contDF** will actually fail because it doesn't cover all possible input ranges.

```
val contDF = spark.range(20).selectExpr("cast(id as double)")
```

Bucketing(cont.)

- scala.Double.NegativeInfinity or scala.Double.PositiveInfinity to cover all possible ranges outside of the inner splits

- In order to handle null or NaN values, we must specify the handleInvalid parameter as a certain value.

Bucketing(cont.)

- Here is an example :

```scala
// in Scala
import org.apache.spark.ml.feature.Bucketizer
val bucketBorders = Array(-1.0, 5.0, 10.0, 250.0, 600.0)
val bucketer = new Bucketizer().setSplits(bucketBorders).setInputCol("id")
bucketer.transform(contDF).show()
```

```python
# in Python
from pyspark.ml.feature import Bucketizer
bucketBorders = [-1.0, 5.0, 10.0, 250.0, 600.0]
bucketer = Bucketizer().setSplits(bucketBorders).setInputCol("id")
bucketer.transform(contDF).show()
```

```
+----+----------------------------------------+
|  id|Bucketizer_4cb1be19f4179cc2545d__output|
+----+----------------------------------------+
| 0.0|                                     0.0|
...
|10.0|                                     2.0|
|11.0|                                     2.0|
...
+----+----------------------------------------+
```

Bucketing(cont.)

- Another option is to split based on **percentiles** in our data. This is done with QuantileDiscretizer, which will bucket the values into user-specified buckets with the splits being determined by approximate quantiles values.
  - For instance, the 90th quantile is the point in your data at which 90% of the data is below that value.
  - You can control how **finely** the buckets should be split. Spark does this is by allowing you to **specify the number of buckets** you would like out of the data and it will split up your data accordingly.

Bucketing(cont.)

```scala
// in Scala
import org.apache.spark.ml.feature.QuantileDiscretizer
val bucketer = new QuantileDiscretizer().setNumBuckets(5).setInputCol("id")
```

```scala
val fittedBucketer = bucketer.fit(contDF)
fittedBucketer.transform(contDF).show()
```

```python
# in Python
from pyspark.ml.feature import QuantileDiscretizer
bucketer = QuantileDiscretizer().setNumBuckets(5).setInputCol("id")
fittedBucketer = bucketer.fit(contDF)
fittedBucketer.transform(contDF).show()
```

```
+----+----------------------------------------+
|  id|quantileDiscretizer_cd87d1a1fb8e__output|
+----+----------------------------------------+
| 0.0|                                     0.0|
...
| 6.0|                                     1.0|
| 7.0|                                     2.0|
...
|14.0|                                     3.0|
|15.0|                                     4.0|
...
+----+----------------------------------------+
```

Working with Continuous Features

- Bucketing
- Scaling and Normalization
- StandardScaler

Scaling and Normalization

- Another common task is to scale and normalize continuous data.
- Use this when the data contains a number of columns based on different scales.
  - For instance, say we have a DataFrame with two columns: weight (in ounces) and height (in feet). If we don't scale or normalize, the algorithm will be less sensitive to variations in height because height values in feet are much lower than weight values in ounces.
- Normalization might involve transforming the data so that each point's value is a representation of **its distance from the mean** of that column.
  - For example, if we want to know how far a given individual's height is from the mean height.

Scaling and Normalization (cont.)

- The fundamental goal—we want our data on the same scale so that values can easily be compared to one another in a sensible way.
- For example, the following **vectors** in a column:
  - 1,2
  - 3,4
  - When we apply our scaling function, the “3” and the “1” will be adjusted
  - while the “2” and the “4” will be adjusted according to one another.
- This is commonly referred to as component-wise comparisons.

Working with Continuous Features

- Bucketing
- Scaling and Normalization
- StandardScaler

StandardScaler

- The StandardScaler standardizes a set of features to have zero mean and a standard deviation of 1.

```
// in Scala
import org.apache.spark.ml.feature.StandardScaler
val sScaler = new StandardScaler().setInputCol("features")
sScaler.fit(scaleDF).transform(scaleDF).show()

# in Python
from pyspark.ml.feature import StandardScaler
sScaler = StandardScaler().setInputCol("features")
sScaler.fit(scaleDF).transform(scaleDF).show()
```

```
+---+-------------+------------------------------------------------------------+
|id |features     |StandardScaler_41aaa6044e7c3467adc3__output                 |
+---+-------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[1.1952286093343936,0.02337622911060922,-0.5976143046671968]|
...

|1  |[3.0,10.1,3.0]|[3.5856858280031805,2.3609991401715313,1.7928429140015902]  |
+---+-------------+------------------------------------------------------------+
```

StandardScaler (cont.)

- MinMaxScaler
- MaxAbsScaler
- ElementwiseProduct
- Normalizer

StandardScaler (cont.)

- MinMaxScaler

```
// in Scala
import org.apache.spark.ml.feature.MinMaxScaler
val minMax = new MinMaxScaler().setMin(5).setMax(10).setInputCol("features")
val fittedminMax = minMax.fit(scaleDF)
fittedminMax.transform(scaleDF).show()

# in Python
from pyspark.ml.feature import MinMaxScaler
minMax = MinMaxScaler().setMin(5).setMax(10).setInputCol("features")
fittedminMax = minMax.fit(scaleDF)
fittedminMax.transform(scaleDF).show()
```

```
+---+--------------+----------------------------------------+
| id|      features|MinMaxScaler_460cbafafbe6b9ab7c62__output|
+---+--------------+----------------------------------------+
|  0|[1.0,0.1,-1.0]|                           [5.0,5.0,5.0]|
...
|  1|[3.0,10.1,3.0]|                        [10.0,10.0,10.0]|
+---+--------------+----------------------------------------+
```

StandardScaler (cont.)

- MaxAbsScaler

```
// in Scala
import org.apache.spark.ml.feature.MaxAbsScaler
val maScaler = new MaxAbsScaler().setInputCol("features")
val fittedmaScaler = maScaler.fit(scaleDF)
fittedmaScaler.transform(scaleDF).show()

# in Python
from pyspark.ml.feature import MaxAbsScaler
maScaler = MaxAbsScaler().setInputCol("features")
fittedmaScaler = maScaler.fit(scaleDF)
fittedmaScaler.transform(scaleDF).show()
```

```
+---+--------------+------------------------------------------------------------+
|id |features      |MaxAbsScaler_402587e1d9b6f268b927__output                   |
+---+--------------+------------------------------------------------------------+
|0  |[1.0,0.1,-1.0]|[0.3333333333333333,0.009900990099009901,-0.3333333333333]|
...
|1  |[3.0,10.1,3.0]|[1.0,1.0,1.0]                                               |
+---+--------------+------------------------------------------------------------+
```

StandardScaler (cont.)

- ElementwiseProduct
  - multiplies each input vector by a provided “weight” vector.
  - In other words, it scales each column of the dataset by a scalar multiplier.

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \circ \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_N w_N \end{pmatrix}$$

StandardScaler (cont.)

- ElementwiseProduct

```scala
// in Scala
import org.apache.spark.ml.feature.ElementwiseProduct
import org.apache.spark.ml.linalg.Vectors
val scaleUpVec = Vectors.dense(10.0, 15.0, 20.0)
val scalingUp = new ElementwiseProduct()
  .setScalingVec(scaleUpVec)
  .setInputCol("features")
scalingUp.transform(scaleDF).show()
```

```python
# in Python
from pyspark.ml.feature import ElementwiseProduct
from pyspark.ml.linalg import Vectors
scaleUpVec = Vectors.dense(10.0, 15.0, 20.0)
scalingUp = ElementwiseProduct()\
  .setScalingVec(scaleUpVec)\
  .setInputCol("features")
scalingUp.transform(scaleDF).show()
```

```
+---+--------------+-----------------------------------------------+
| id|      features|ElementwiseProduct_42b29ea5a55903e9fea6__output|
+---+--------------+-----------------------------------------------+
|  0|[1.0,0.1,-1.0]|                                [10.0,1.5,-20.0]|
...
|  1|[3.0,10.1,3.0]|                               [30.0,151.5,60.0]|
+---+--------------+-----------------------------------------------+
```

StandardScaler (cont.)

- Normalizer

```
// in Scala
import org.apache.spark.ml.feature.Normalizer
val manhattanDistance = new Normalizer().setP(1).setInputCol("features")
manhattanDistance.transform(scaleDF).show()
```

```
# in Python
from pyspark.ml.feature import Normalizer
manhattanDistance = Normalizer().setP(1).setInputCol("features")
manhattanDistance.transform(scaleDF).show()
```

```
+---+--------------+------------------------------+
| id|      features|normalizer_1bf2cd17ed33__output|
+---+--------------+------------------------------+
|  0|[1.0,0.1,-1.0]|           [0.47619047619047...|
|  1| [2.0,1.1,1.0]|           [0.48780487804878...|
|  0|[1.0,0.1,-1.0]|           [0.47619047619047...|
|  1| [2.0,1.1,1.0]|           [0.48780487804878...|
|  1|[3.0,10.1,3.0]|           [0.18633540372670...|
+---+--------------+------------------------------+
```

Working with Categorical Features

- StringIndexer
- Indexing in Vectors
- One-Hot Encoding

StringIndexer

- Maps strings to different numerical IDs.
- Spark's StringIndexer creates metadata attached to the Dataframe that specify what inputs correspond to what outputs.
- StringIndexer can be applied to columns that are not strings, in which case, they will be converted to strings before being indexed.

```scala
// in Scala
import org.apache.spark.ml.feature.StringIndexer
val lblIndxr = new StringIndexer().setInputCol("lab").setOutputCol("labelInd")
val idxRes = lblIndxr.fit(simpleDF).transform(simpleDF)
idxRes.show()
```

```python
# in Python
from pyspark.ml.feature import StringIndexer
lblIndxr = StringIndexer().setInputCol("lab").setOutputCol("labelInd")
idxRes = lblIndxr.fit(simpleDF).transform(simpleDF)
idxRes.show()
```

```
+-----+----+------+------------------+--------+
|color| lab|value1|            value2|labelInd|
+-----+----+------+------------------+--------+
|green|good|     1|14.386294994851129|     1.0|
...
|  red| bad|     2|14.386294994851129|     0.0|
```

Converting Indexed Values Back to Text

- **IndexToString** converts indexed values back to text.
- We do not have to input our value to the String key; Spark's MLlib maintains this metadata .

```
// in Scala
import org.apache.spark.ml.feature.IndexToString
val labelReverse = new IndexToString().setInputCol("labelInd")
labelReverse.transform(idxRes).show()
# in Python
from pyspark.ml.feature import IndexToString
labelReverse = IndexToString().setInputCol("labelInd")
labelReverse.transform(idxRes).show()
```

```
+-----+----+------+-----------------+--------+-------------------------------+
|color| lab|value1|           value2|labelInd|IndexToString_415...2a0d__output|
+-----+----+------+-----------------+--------+-------------------------------+
|green|good|     1|14.386294994851129|     1.0|                           good|
...
|  red| bad|     2|14.386294994851129|     0.0|                            bad|
+-----+----+------+-----------------+--------+-------------------------------+
```

Indexing in Vectors

- **VectorIndexer** is a helpful tool for working with categorical variables that are already found inside of vectors in the dataset.
- This tool will automatically find categorical features inside of the input vectors and convert them to numerical features with **zero-based** category indices.

```
import org.apache.spark.ml.feature.VectorIndexer

import org.apache.spark.ml.linalg.Vectors

val idxIn = spark.createDataFrame(Seq((Vectors.dense(1, 2, 3),1), (Vectors.dense(2, 5, 6),2),(Vectors.dense(1, 8, 9),3) )).toDF("features", "label")

val indxr = new VectorIndexer().setInputCol("features")
.setOutputCol("idxed").setMaxCategories(2)

indxr.fit(idxIn).transform(idxIn).show
```

```
+-------------+-----+-------------+
|     features|label|        idxed|
+-------------+-----+-------------+
|[1.0,2.0,3.0]|    1|[0.0,2.0,3.0]|
|[2.0,5.0,6.0]|    2|[1.0,5.0,6.0]|
|[1.0,8.0,9.0]|    3|[0.0,8.0,9.0]|
+-------------+-----+-------------+
```

One –Hot Encoding

- Indexing does not always represent our categorical variables in the correct way for downstream models to process.
- For instance, when we index our “color” column, some colors have a higher value (or index number) than others (in our case, blue is 1 and green is 2).

One –Hot Encoding

- Indexing does not always represent our categorical variables in the correct way for downstream models to process.
- For instance, when we index our “color” column, some colors have a higher value (or index number) than others (in our case, blue is 1 and green is 2).
- This is **incorrect** because it gives the mathematical appearance that the input to the machine learning algorithm seems to specify that green > blue.
- OneHotEncoder, which will convert each distinct value to a Boolean flag (1 or 0) as a **component in a vector**. no longer ordered.

One –Hot Encoding

```
import org.apache.spark.ml.feature.{StringIndexer, OneHotEncoder}
val lblIndxr = new StringIndexer().setInputCol("color").setOutputCol("colorInd")
val colorLab = lblIndxr.fit(simpleDF).transform(simpleDF.select("color"))
val ohe = new OneHotEncoder().setInputCol("colorInd")
ohe.transform(colorLab).show()
```

```
+-----+--------+-----------------------------------------+
|color|colorInd|OneHotEncoder_46b5ad1ef147bb355612__output|
+-----+--------+-----------------------------------------+
|green|     1.0|                            (2,[1],[1.0])|
| blue|     2.0|                                (2,[],[])|
...
|  red|     0.0|                            (2,[0],[1.0])|
|  red|     0.0|                            (2,[0],[1.0])|
+-----+--------+-----------------------------------------+
```

Text Data Transformers

- Text Data Transformation is needed for Machine Learning model to use Text effectively.
- Two types of texts : Free-form text and string categorical variables.
- We now focus on free-form text.

Tokenizing Text

- Converts free form texts to “Tokens” or individual words.
- **Tokenizer** class is the easiest way to do this transformation.
- This class separates a string of words by whitespace and converts them into array of words.

```
import org.apache.spark.ml.feature.Tokenizer
val tkn = new Tokenizer().setInputCol("Description").setOutputCol("DescOut")
val tokenized = tkn.transform(sales.select("Description")) tokenized.show(false)
```

```
+---------------------------------+-----------------------------------------+
|Description                      DescOut                                   |
+---------------------------------+-----------------------------------------+
|RABBIT NIGHT LIGHT               |[rabbit, night, light]                   |
|DOUGHNUT LIP GLOSS               |[doughnut, lip, gloss]                   |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion]  |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set, brown]  |
+---------------------------------+-----------------------------------------+
```

Tokenizing Text(contd.)

- We can also create a Tokenizer that is not just based white space but a regular expression with the **RegexTokenizer**
- The format of the regular expression should conform to the Java Regular Expression (RegEx) syntax

```
import org.apache.spark.ml.feature.RegexTokenizer
 val rt = new RegexTokenizer().setInputCol("Description").setOutputCol("DescOut")
.setPattern(" ") // simplest expression
.setToLowercase(true)
rt.transform(sales.select("Description")).show(false)
```

```
+-----------------------------------+------------------------------------------+
|Description                        DescOut                                    |
+-----------------------------------+------------------------------------------+
|RABBIT NIGHT LIGHT                 |[rabbit, night, light]                    |
|DOUGHNUT LIP GLOSS                 |[doughnut, lip, gloss]                    |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion]  |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set, brown]  |
+-----------------------------------+------------------------------------------+
```

Removing Common Words

- A common task after tokenization is to filter **stop words.**
- Stop words are the common words that are not relevant in many kinds of analysis.
- Example: the, and, but
- Spark contains a list of default stop words

Creating Word Combinations

- After Tokenizing and filtering stop words we get clean set of words to use as feature.
- Word combinations are technically referred to as **n-grams** (sequences of words of length n)
- n-gram of length 1 is called unigrams, those of length 2 are called bigrams, and those of length 3 are called trigrams.
- Order matters in n-gram creation.

Creating Word Combinations

- The bigrams of “Big Data Processing Made Simple” are:
  - “Big Data”
  - “Data Processing”
  - “Processing Made”
  - “Made Simple”

- The trigrams are :
  - “Big Data Processing”
  - “Data Processing Made”
  - “Processing Made Simple”

Creating Word Combinations

```
import org.apache.spark.ml.feature.NGram
val unigram = new NGram().setInputCol("DescOut").setN(1)
val bigram = new NGram().setInputCol("DescOut").setN(2)
unigram.transform(tokenized.select("DescOut")).show(false)
bigram.transform(tokenized.select("DescOut")).show(false)
```

```
+----------------------------------------+-----------------------------------
DescOut                                  |ngram_104c4da6a01b__output       ...
+----------------------------------------+-----------------------------------
|[rabbit, night, light]                  |[rabbit, night, light]           ...
|[doughnut, lip, gloss]                  |[doughnut, lip, gloss]           ...
...
|[airline, bag, vintage, world, champion] |[airline, bag, vintage, world, cha...
|[airline, bag, vintage, jet, set, brown] |[airline, bag, vintage, jet, set, ...
+----------------------------------------+-----------------------------------
```

And the result for bigrams:

```
+----------------------------------------+-----------------------------------
DescOut                                  |ngram_6e68fb3a642a__output       ...
+----------------------------------------+-----------------------------------
|[rabbit, night, light]                  |[rabbit night, night light]      ...
|[doughnut, lip, gloss]                  |[doughnut lip, lip gloss]        ...
...
|[airline, bag, vintage, world, champion]  |[airline bag, bag vintage, vintag...
|[airline, bag, vintage, jet, set, brown]  |[airline bag, bag vintage, vintag...
+----------------------------------------+-----------------------------------
```

Converting words into numerical Representations

- Simplest way is just to include binary counts of a word in a given document (in our case, a row) by measuring whether or not each row contains a given word.
- In addition, we can **count** words using a **CountVectorizer**.
- A CountVectorizer does two things while operating on tokenized data:

1. During the fit process, it **finds** the set of words in all the **documents** and then **counts** the occurrences of those words in those documents.
2. It then **counts** the **occurrences** of a given word in **each row** of the DataFrame column during the transformation process, and outputs a vector with the terms that occur in that row.

Converting words into numerical Representations(cont.)

- CountVectorizer treats every row as a **Document** and every word as a **term**
- The total collection of all terms is **Vocabulary**

```
// in Scala
import org.apache.spark.ml.feature.CountVectorizer
val cv = new CountVectorizer()
  .setInputCol("DescOut")
  .setOutputCol("countVec")
  .setVocabSize(500)
  .setMinTF(1)
  .setMinDF(2)
val fittedCV = cv.fit(tokenized)
fittedCV.transform(tokenized).show(false)
```

```
+---------------------------------+----------------------------------------+
DescOut                           |countVec                                |
+---------------------------------+----------------------------------------+
|[rabbit, night, light]           |(500,[150,185,212],[1.0,1.0,1.0])       |
|[doughnut, lip, gloss]           |(500,[462,463,492],[1.0,1.0,1.0])       |
...
|[airline, bag, vintage, world,...|(500,[2,6,328],[1.0,1.0,1.0])           |
|[airline, bag, vintage, jet, s...|(500,[0,2,6,328,405],[1.0,1.0,1.0,1.0,1.0])|
+---------------------------------+----------------------------------------+
```

Term Frequency-inverse document frequency(TF-IDF)

- Another way to approach the problem of converting text into a numerical representation.
- *TF–IDF* measures how often a word occurs in each document, **weighted** according to how many documents that word occurs in.
- Words that occur in a few documents are given more weight than words that occur in many documents
- In practice, a word like “the” would be weighted very low.
- TF–IDF helps find documents that share similar topics.

TF-IDF(Cont.)

```scala
// in Scala
val tfIdfIn = tokenized
  .where("array_contains(DescOut, 'red')")
  .select("DescOut")
  .limit(10)
tfIdfIn.show(false)
```

```python
# in Python
tfIdfIn = tokenized\
  .where("array_contains(DescOut, 'red')")\
  .select("DescOut")\
  .limit(10)
tfIdfIn.show(10, False)
```

```
+-------------------------------------+
DescOut                               |
+-------------------------------------+
|[gingham, heart, , doorstop, red]    |
...
|[red, retrospot, oven, glove]        |
|[red, retrospot, plate]              |
+-------------------------------------+
```

TF-IDF(Cont.)

```scala
// in Scala
val tfIdfIn = tokenized
  .where("array_contains(DescOut, 'red')")
  .select("DescOut")
  .limit(10)
tfIdfIn.show(false)
```

```python
# in Python
tfIdfIn = tokenized\
  .where("array_contains(DescOut, 'red')")\
  .select("DescOut")\
  .limit(10)
tfIdfIn.show(10, False)
```

```
+------------------------------------+
DescOut                              |
+------------------------------------+
|[gingham, heart, , doorstop, red]   |
```

```
...
|[red, retrospot, oven, glove]       |
|[red, retrospot, plate]             |
+------------------------------------+
```

```scala
// in Scala
import org.apache.spark.ml.feature.{HashingTF, IDF}
val tf = new HashingTF()
  .setInputCol("DescOut")
  .setOutputCol("TFOut")
  .setNumFeatures(10000)
val idf = new IDF()
  .setInputCol("TFOut")
  .setOutputCol("IDFOut")
  .setMinDocFreq(2)
```

```python
# in Python
from pyspark.ml.feature import HashingTF, IDF
tf = HashingTF()\
  .setInputCol("DescOut")\
  .setOutputCol("TFOut")\
  .setNumFeatures(10000)
idf = IDF()\
  .setInputCol("TFOut")\
  .setOutputCol("IDFOut")\
  .setMinDocFreq(2)
```

TF-IDF(Cont.)

```scala
// in Scala
val tfIdfIn = tokenized
  .where("array_contains(DescOut, 'red')")
  .select("DescOut")
  .limit(10)
tfIdfIn.show(false)
```

```
+-------------------------------------+
DescOut                               |
+-------------------------------------+
|[gingham, heart, , doorstop, red]    |
...
|[red, retrospot, oven, glove]        |
|[red, retrospot, plate]              |
+-------------------------------------+
```

```python
# in Python
tfIdfIn = tokenized\
  .where("array_contains(DescOut, 'red')")\
  .select("DescOut")\
  .limit(10)
tfIdfIn.show(10, False)
```

```scala
// in Scala
import org.apache.spark.ml.feature.{HashingTF, IDF}
val tf = new HashingTF()
  .setInputCol("DescOut")
  .setOutputCol("TFOut")
  .setNumFeatures(10000)
val idf = new IDF()
  .setInputCol("TFOut")
  .setOutputCol("IDFOut")
  .setMinDocFreq(2)
```

```python
# in Python
from pyspark.ml.feature import HashingTF, IDF
tf = HashingTF()\
  .setInputCol("DescOut")\
  .setOutputCol("TFOut")\
  .setNumFeatures(10000)
idf = IDF()\
  .setInputCol("TFOut")\
  .setOutputCol("IDFOut")\
  .setMinDocFreq(2)
```

```scala
// in Scala
idf.fit(tf.transform(tfIdfIn)).transform(tf.transform(tfIdfIn)).show(false)
```

```python
# in Python
idf.fit(tf.transform(tfIdfIn)).transform(tf.transform(tfIdfIn)).show(10, False)
```

```
(10000,[2591,4291,4456],[1.0116009116784799,0.0,0.0])
```

Word2Vec

- Word2Vec is a deep learning–based tool for computing a vector representation of a set of words
- The goal is to have similar words close to one another in this vector space
- Word2Vec is notable for capturing **relationships** between words based on their semantics
- Word2Vec uses a technique called “skip-grams” to convert a sentence of words into a vector representation.

Word2Vec (Cont.)

- Steps for converting a sentence of words into a vector representation
    - Builds a vocabulary
    - Removes a token
    - Trains the model to predict the missing token in the “n-gram” representation.

Feature Manipulation

- PCA
- Interaction
- Polynomial Expansion

Principal Component Analysis (PCA)

- Principal Components Analysis (PCA) is a mathematical technique for finding the most important aspects of our data (the principal components).
- It changes the feature representation of our data by creating a new set of features (“aspects”).
- Each new feature is a combination of the original features.
- PCA is useful when we have a large input dataset is large and want to **reduce** the number of features.

Principal Component Analysis (PCA)

```
import org.apache.spark.ml.feature.PCA
val pca = new PCA().setInputCol("features").setK(2)
pca.fit(scaleDF).transform(scaleDF).show(false)
```

```
+---+-------------+-----------------------------------------+
|id |features     |pca_7c5c4aa7674e__output                 |
+---+-------------+-----------------------------------------+
|0  |[1.0,0.1,-1.0]|[0.0713719499248418,-0.4526654888147822]  |
...
|1  |[3.0,10.1,3.0]|[-10.872398139848944,0.03096269706015064]|
+---+-------------+-----------------------------------------+
```

Interaction

- The feature transformer **Interaction** allows to create an interaction between two variables manually.
- It multiplies the two features together—something that a typical linear model would not do for every possible pair of features.
- Currently only available directly in Scala but can be called from any language using the RFormula.

Polynomial Expansion

- Polynomial expansion is used to generate interaction variables of all the input columns.
- For a degree-2 polynomial, Spark takes every value in feature vector, multiplies it by every other value in the feature vector, and then stores the results as features
- For instance, if we have two input features, we'll get four output features if we use a second degree polynomial (2x2).

Feature Selection

- Feature selection is the process of selecting a smaller subset from large range of possible features.
- **ChiSqSelector** leverages a statistical test to identify features that are not independent from the label we are trying to predict, and drop the uncorrelated features.
- This method is based on Chi-Sqauare Test.
- Several ways of picking up best features : using **numTopFeatures** method, **percentile** and **fpr**

Advanced Topics

- **Persisting Transformers:** To persist a transformer we can write method on the fitted transformer and specify the disk location to write it.
- **Writing a custom transformer:** Custom transformers are valuable when we have to encode our own business logic to fit into the ML pipeline.