**Chapter 25. Preprocessing and Feature Engineering**

Any data scientist worth her salt knows that one of the biggest challenges (and time sinks) in advanced analytics is preprocessing. It’s not that it’s particularly complicated programming, but rather that it requires deep knowledge of the data you are working with and an understanding of what your model needs in order to successfully leverage this data. This chapter covers the details of how you can use Spark to perform preprocessing and feature engineering. We’ll walk through the core requirements you’ll need to meet in order to train an MLlib model in terms of how your data is structured. We will then discuss the different tools Spark makes available for performing this kind of work.

**Formatting Models According to Your Use Case**

To preprocess data for Spark’s different advanced analytics tools, you must consider your end objective. The following list walks through the requirements for input data structure for each advanced analytics task in MLlib:

* In the case of most classification and regression algorithms, you want to get your data into a column of type Double to represent the label and a column of type Vector (either dense or sparse) to represent the features.
* In the case of recommendation, you want to get your data into a column of users, a column of items (say movies or books), and a column of ratings.
* In the case of unsupervised learning, a column of type Vector (either dense or sparse) is needed to represent the features.
* In the case of graph analytics, you will want a DataFrame of vertices and a DataFrame of edges.

The best way to get your data in these formats is through transformers. Transformers are functions that accept a DataFrame as an argument and return a new DataFrame as a response. This chapter will focus on what transformers are relevant for particular use cases rather than attempting to enumerate every possible transformer.

**NOTE**

Spark provides a number of transformers as part of the org.apache.spark.ml.feature package. The corresponding package in Python is pyspark.ml.feature. New transformers are constantly popping up in Spark MLlib and therefore it is impossible to include a definitive list in this book. The most up-to-date information can be found on the Spark [documentation site](http://spark.apache.org/docs/latest/ml-features.html).

Before we proceed, we’re going to read in several different sample datasets, each of which has different properties we will manipulate in this chapter:

*// 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")

*# in Python*

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")

fakeIntDF = spark.read.parquet("/data/simple-ml-integers")

simpleDF = spark.read.json("/data/simple-ml")

scaleDF = spark.read.parquet("/data/simple-ml-scaling")

In addition to this realistic sales data, we’re going to use several simple synthetic datasets as well. FakeIntDF, simpleDF, and scaleDF all have very few rows. This will give you the ability to focus on the exact data manipulation we are performing instead of the various inconsistencies of any particular dataset. Because we’re going to be accessing the sales data a number of times, we’re going to cache it so we can read it efficiently from memory as opposed to reading it from disk every time we need it. Let’s also check out the first several rows of data in order to better understand what’s in the dataset:

sales.cache()

sales.show()

+---------+---------+--------------------+--------+-------------------+---------

|InvoiceNo|StockCode| Description|Quantity| InvoiceDate|UnitPr...

+---------+---------+--------------------+--------+-------------------+---------

| 580538| 23084| RABBIT NIGHT LIGHT| 48|2011-12-05 08:38:00| 1...

...

| 580539| 22375|AIRLINE BAG VINTA...| 4|2011-12-05 08:39:00| 4...

+---------+---------+--------------------+--------+-------------------+---------

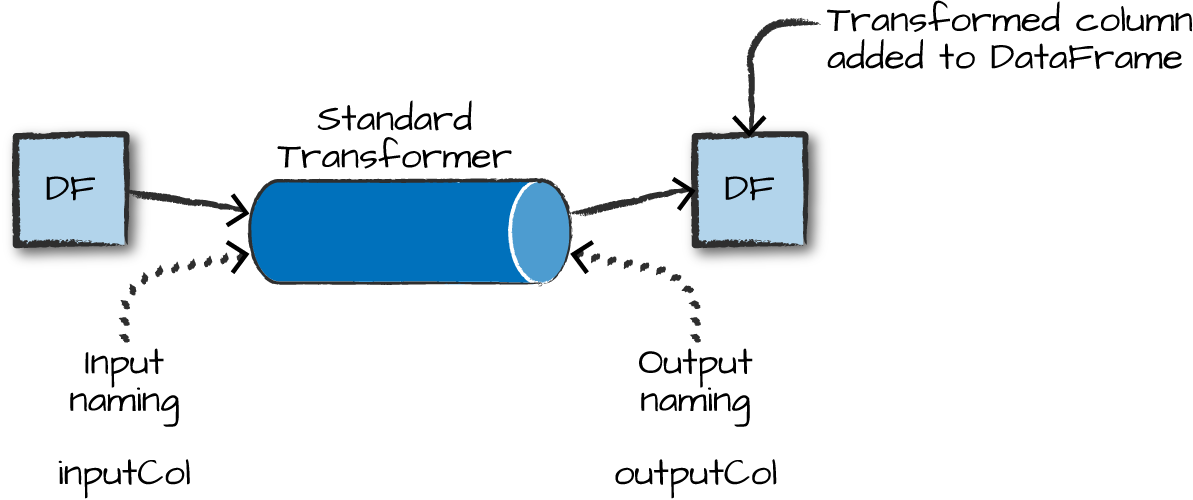
**NOTE**

It is important to note that we filtered out null values here. MLlib does not always play nicely with null values at this point in time. This is a frequent cause for problems and errors and a great first step when you are debugging. Improvements are also made with every Spark release to improve algorithm handling of null values.

**Transformers**

We discussed transformers in the previous chapter, but it’s worth reviewing them again here. Transformers are functions that convert raw data in some way. This might be 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. Transformers are primarily used in preprocessing or feature generation.

Spark’s transformer only includes a transform method. This is because it will not change based on the input data. Figure 25-1 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.



*Figure 25-1. A Spark transformer*

The Tokenizer is an example of a transformer. It tokenizes a string, splitting on a given character, and has nothing to learn from our data; it simply applies a function. We’ll discuss the tokenizer in more depth later in this chapter, but here’s a small code snippet showing how a tokenizer is built to accept the input column, how it transforms the data, and then the output from that transformation:

*// 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] |

...

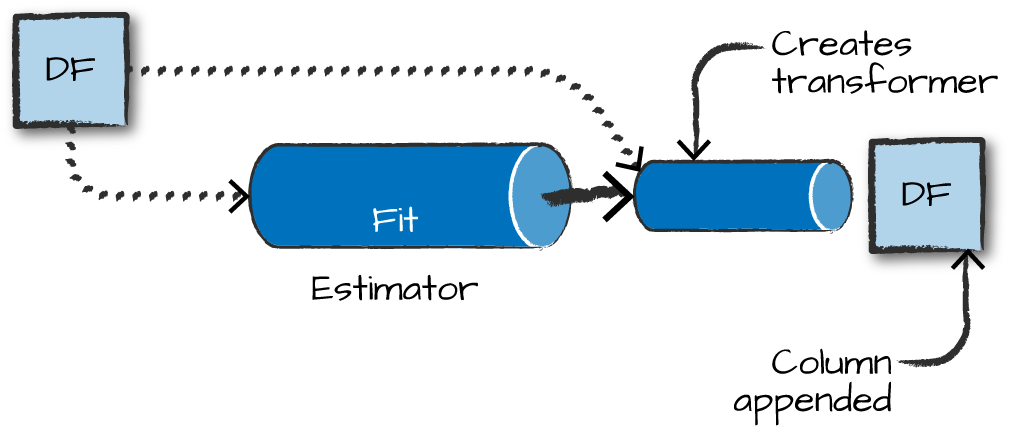
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world, champion] |

|AIRLINE BAG VINTAGE JET SET BROWN |[airline, bag, vintage, jet, set, brown] |

+-----------------------------------+------------------------------------------+

**Estimators for Preprocessing**

Another tool for preprocessing are estimators. An *estimator* is necessary when a transformation you would like to perform must be initialized with data or information about the input column (often derived by doing a pass over the input column itself). For example, if you wanted to scale the values in our column to have mean zero and unit variance, you would need to perform a pass over the entire data in order to calculate the values you would use to normalize the data to mean zero and unit variance. In effect, an estimator can be a transformer configured according to your particular input data. In simplest terms, you can either blindly apply a transformation (a “regular” transformer type) or perform a transformation based on your data (an estimator type). Figure 25-2 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).



*Figure 25-2. A Spark estimator*

An example of this type 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. Here’s a sample code snippet showing the entire process, as well as the output:

*// 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] |

+---+--------------+------------------------------------------------------------+

We will use both estimators and transformers throughout and cover more about these particular estimators (and add examples in Python) later on in this chapter.

**Transformer Properties**

All transformers require you to specify, at a minimum, the inputCol and the outputCol, which represent the column name of the input and output, respectively. You set these with setInputCol and setOutputCol. There are some defaults (you can find these in the documentation), but it is a best practice to manually specify them yourself for clarity. In addition to input and output columns, all transformers have different parameters that you can tune (whenever we mention a parameter in this chapter you must set it with a set() method). In Python, we also have another method to set these values with keyword arguments to the object’s constructor. We exclude these from the examples in the next chapter for consistency. Estimators require you to fit the transformer to your particular dataset and then call transform on the resulting object.

**NOTE**

Spark MLlib stores metadata about the columns it uses in each DataFrame as an attribute on the column itself. This allows it to properly store (and annotate) that a column of Doubles may actually represent a series of categorical variables instead of continuous values. However, metadata won’t show up when you print the schema or the DataFrame.

**High-Level Transformers**

High-level transformers, such as the RFormula we saw in the previous chapter, allow you to concisely specify a number of transformations in one. These operate at a “high level”, and 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. While this is not always possible, it’s a good objective.

**RFormula**

The RFormula is the easiest transfomer 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 something called *one-hot encoding*. In brief, one-hot encoding converts a set of values into a set of binary columns specifying whether or not the data point has each particular value (we’ll discuss one-hot encoding in more depth later in the chapter). With the RFormula, numeric columns will be cast to Double but will *not* be one-hot encoded. If the label column is of type String, it will be first transformed to Double with StringIndexer.

**WARNING**

Automatic casting of numeric columns to Double without one-hot encoding has some important implications. If you have numerically valued categorical variables, they will *only* be cast to Double, implicitly specifying an order. It is important to ensure the input types correspond to the expected conversion. If you have categorical variables that really have no order relation, they should be cast to String. You can also manually index columns (see “Working with Categorical Features”).

The RFormula allows you to specify your transformations in declarative syntax. It is simple to use once you understand the syntax. Currently, RFormula supports a limited subset of the R operators that in practice work quite well for simple transformations. The basic operators are:

~

Separate target and terms

+

Concatenate terms; “+ 0” means removing the intercept (this means the *y*-intercept of the line that we will fit will be 0)

-

Remove a term; “- 1” means removing intercept (this means the *y*-intercept of the line that we will fit will be 0)

:

Interaction (multiplication for numeric values, or binarized categorical values)

.

All columns except the target/dependent variable

RFormula also uses default columns of label and features to label, you guessed it, the label and the set of features that it outputs (for supervised machine learning). The models covered later on in this chapter by default require those column names, making it easy to pass the resulting transformed DataFrame into a model for training. If this doesn’t make sense yet, don’t worry—it’ll become clear once we actually start using models in later chapters.

Let’s use RFormula in an example. In this case, we want to use all available variables (the .) and then specify an interaction between value1 and color and value2 and color as additional features to generate:

*// in Scala*

**import** **org.apache.spark.ml.feature.RFormula**

**val** supervised **=** **new** **RFormula**()

.setFormula("lab ~ . + color:value1 + color:value2")

supervised.fit(simpleDF).transform(simpleDF).show()

*# in Python*

**from** **pyspark.ml.feature** **import** RFormula

supervised = RFormula(formula="lab ~ . + color:value1 + color:value2")

supervised.fit(simpleDF).transform(simpleDF).show()

+-----+----+------+------------------+--------------------+-----+

|color| lab|value1| value2| features|label|

+-----+----+------+------------------+--------------------+-----+

|green|good| 1|14.386294994851129|(10,[1,2,3,5,8],[...| 1.0|

| blue| bad| 8|14.386294994851129|(10,[2,3,6,9],[8....| 0.0|

...

| red| bad| 1| 38.97187133755819|(10,[0,2,3,4,7],[...| 0.0|

| red| bad| 2|14.386294994851129|(10,[0,2,3,4,7],[...| 0.0|

+-----+----+------+------------------+--------------------+-----+

**SQL Transformers**

A SQLTransformer allows you to leverage Spark’s vast library of SQL-related manipulations just as you would a MLlib transformation. 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 *TH IS*. You might want to use SQLTransformer if you want to formally codify some DataFrame manipulation as a preprocessing step, or try different SQL expressions for features during hyperparameter tuning. Also note that the output of this transformation will be appended as a column to the output DataFrame.

You might want to use an SQLTransformer in order to represent all of your manipulations on the very rawest form of your data so you can version different variations of manipulations as transformers. This gives you the benefit of building and testing varying pipelines, all by simply swapping out transformers. The following is a basic example of using SQLTransformer:

*// 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()

*# 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()

Here’s a sample of the output:

-------------+--------+----------+

|sum(Quantity)|count(1)|CustomerID|

+-------------+--------+----------+

| 119| 62| 14452.0|

...

| 138| 18| 15776.0|

+-------------+--------+----------+

For extensive samples of these transformations, refer back to [Part II](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part02.html#part2).

**VectorAssembler**

The VectorAssembler is a tool you’ll use in nearly every single pipeline you generate. It helps concatenate all your features into one big vector you can then pass into an estimator. It’s used typically in the last step of a machine learning pipeline and takes as input a number of columns of Boolean, Double, or Vector. 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.

The output from the following code snippet will make it clear how this works:

*// in Scala*

**import** **org.apache.spark.ml.feature.VectorAssembler**

**val** va **=** **new** **VectorAssembler**().setInputCols(**Array**("int1", "int2", "int3"))

va.transform(fakeIntDF).show()

*# 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**

Continuous features are just values on the number line, from positive infinity to negative infinity. There are two common transformers for continuous features. First, you can convert continuous features into categorical features via a process called bucketing, or you can scale and normalize your features according to several different requirements. These transformers will *only* work on Double types, so make sure you’ve turned any other numerical values to Double:

*// in Scala*

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

*# in Python*

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

**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.”

To specify the bucket, set its borders. For example, setting splits to 5.0, 10.0, 250.0 on our contDF will actually fail because we don’t cover all possible input ranges. 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.

**WARNING**

The Bucketizer can be confusing because we specify bucket borders via the splits method, but these are not actually splits.

To cover all possible ranges, scala.Double.NegativeInfinity might be another split option, with scala.Double.PositiveInfinity to cover all possible ranges outside of the inner splits. In Python we specify this in the following way: float("inf"), float("-inf").

In order to handle null or NaN values, we must specify the handleInvalid parameter as a certain value. We can either keep those values (keep), error or null, or skip those rows. Here’s an example of using bucketing:

*// 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()

*# 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|

...

+----+---------------------------------------+

In addition to splitting based on hardcoded values, 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 by setting the relative error for the approximate quantiles calculation using setRelativeError. 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. The following is an example:

*// in Scala*

**import** **org.apache.spark.ml.feature.QuantileDiscretizer**

**val** bucketer **=** **new** **QuantileDiscretizer**().setNumBuckets(5).setInputCol("id")

**val** fittedBucketer **=** bucketer.fit(contDF)

fittedBucketer.transform(contDF).show()

*# 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|

...

+----+----------------------------------------+

**ADVANCED BUCKETING TECHNIQUES**

The techniques descriubed here are the most common ways of bucketing data, but there are a number of other ways that exist in Spark today. All of these processes are the same from a data flow perspective: start with continuous data and place them in buckets so that they become categorical. Differences arise depending on the algorithm used to compute these buckets. The simple examples we just looked at are easy to intepret and work with, but more advanced techniques such as locality sensitivity hashing (LSH) are also available in MLlib.

**Scaling and Normalization**

We saw how we can use bucketing to create groups out of continuous variables. Another common task is to scale and normalize continuous data. While not always necessary, doing so is usually a best practice. You might want to do this when your 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 you 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. That’s an example where you should scale your data.

An example of normalization might involve transforming the data so that each point’s value is a representation of its distance from the mean of that column. Using the same example from before, we might want to know how far a given individual’s height is from the mean height. Many algorithms assume that their input data is normalized.

As you might imagine, there are a multitude of algorithms we can apply to our data to scale or normalize it. Enumerating them all is unnecessary here because they are covered in many other texts and machine learning libraries. If you’re unfamiliar with the concept in detail, check out any of the books referenced in the previous chapter. Just keep in mind 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. In MLlib, this is always done on columns of type Vector. MLlib will look across all the rows in a given column (of type Vector) and then treat every dimension in those vectors as its own particular column. It will then apply the scaling or normalization function on each dimension separately.

A simple example might be the following vectors in a column:

1,2

3,4

When we apply our scaling (but not normalization) function, the “3” and the “1” will be adjusted according to those two values while the “2” and the “4” will be adjusted according to one another. This is commonly referred to as component-wise comparisons.

**StandardScaler**

The StandardScaler standardizes a set of features to have zero mean and a standard deviation of 1. The flag withStd will scale the data to unit standard deviation while the flag withMean (false by default) will center the data prior to scaling it.

**WARNING**

Centering can be very expensive on sparse vectors because it generally turns them into dense vectors, so be careful before centering your data.

Here’s an example of using a StandardScaler:

*// 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()

The output is shown below:

+---+--------------+------------------------------------------------------------+

|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] |

+---+--------------+------------------------------------------------------------+

**MINMAXSCALER**

The MinMaxScaler will scale the values in a vector (component wise) to the proportional values on a scale from a given min value to a max value. If you specify the minimum value to be 0 and the maximum value to be 1, then all the values will fall in between 0 and 1:

*// 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]|

+---+--------------+-----------------------------------------+

**MAXABSSCALER**

The max absolute scaler (MaxAbsScaler) scales the data by dividing each value by the maximum absolute value in this feature. All values therefore end up between −1 and 1. This transformer does not shift or center the data at all in the process:

*// 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] |

+---+--------------+----------------------------------------------------------+

**ELEMENTWISEPRODUCT**

The ElementwiseProduct allows us to scale each value in a vector by an arbitrary value. For example, given the vector below and the row “1, 0.1, -1” the output will be “10, 1.5, -20.” Naturally the dimensions of the scaling vector must match the dimensions of the vector inside the relevant column:

*// 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()

*# 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]|

+---+--------------+-----------------------------------------------+

**NORMALIZER**

The normalizer allows us to scale multidimensional vectors using one of several power norms, set through the parameter “p”. For example, we can use the Manhattan norm (or Manhattan distance) with p = 1, Euclidean norm with p = 2, and so on. The Manhattan distance is a measure of distance where you can only travel from point to point along the straight lines of an axis (like the streets in Manhattan).

Here’s an example of using the 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**

The most common task for categorical features is indexing. Indexing converts a categorical variable in a column to a numerical one that you can plug into machine learning algorithms. While this is conceptually simple, there are some catches that are important to keep in mind so that Spark can do this in a stable and repeatable manner.

In general, we recommend re-indexing every categorical variable when pre-processing just for consistency’s sake. This can be helpful in maintaining your models over the long run as your encoding practices may change over time.

**StringIndexer**

The simplest way to index is via the StringIndexer, which maps strings to different numerical IDs. Spark’s StringIndexer also creates metadata attached to the DataFrame that specify what inputs correspond to what outputs. This allows us later to get inputs back from their respective index values:

*// 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()

*# 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|

+-----+----+------+------------------+--------+

We can also apply StringIndexer to columns that are not strings, in which case, they will be converted to strings before being indexed:

*// in Scala*

**val** valIndexer **=** **new** **StringIndexer**()

.setInputCol("value1")

.setOutputCol("valueInd")

valIndexer.fit(simpleDF).transform(simpleDF).show()

*# in Python*

valIndexer = StringIndexer().setInputCol("value1").setOutputCol("valueInd")

valIndexer.fit(simpleDF).transform(simpleDF).show()

+-----+----+------+------------------+--------+

|color| lab|value1| value2|valueInd|

+-----+----+------+------------------+--------+

|green|good| 1|14.386294994851129| 1.0|

...

| red| bad| 2|14.386294994851129| 0.0|

+-----+----+------+------------------+--------+

Keep in mind that the StringIndexer is an estimator that must be fit on the input data. This means it must see all inputs to select a mapping of inputs to IDs. If you train a StringIndexer on inputs “a,” “b,” and “c” and then go to use it against input “d,” it will throw an error by default. Another option is to skip the entire row if the input value was not a value seen during training. Going along with the previous example, an input value of “d” would cause that row to be skipped entirely. We can set this option before or after training the indexer or pipeline. More options may be added to this feature in the future but as of Spark 2.2, you can only skip or throw an error on invalid inputs.

valIndexer.setHandleInvalid("skip")

valIndexer.fit(simpleDF).setHandleInvalid("skip")

**Converting Indexed Values Back to Text**

When inspecting your machine learning results, you’re likely going to want to map back to the original values. Since MLlib classification models make predictions using the indexed values, this conversion is useful for converting model predictions (indices) back to the original categories. We can do this with IndexToString. You’ll notice that we do not have to input our value to the String key; Spark’s MLlib maintains this metadata for you. You can optionally specify the outputs.

*// 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 your dataset. This tool will automatically find categorical features inside of your input vectors and convert them to categorical features with zero-based category indices. For example, in the following DataFrame, the first column in our Vector is a categorical variable with two different categories while the rest of the variables are continuous. By setting maxCategories to 2 in our VectorIndexer, we are instructing Spark to take any column in our vector with two or less distinct values and convert it to a categorical variable. This can be helpful when you know how many unique values there are in your largest category because you can specify this and it will automatically index the values accordingly. Conversely, Spark changes the data based on this parameter, so if you have continuous variables that don’t appear particularly continuous (lots of repeated values) these can be unintentionally converted to categorical variables if there are too few unique values.

*// in Scala*

**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

*# in Python*

**from** **pyspark.ml.feature** **import** VectorIndexer

**from** **pyspark.ml.linalg** **import** Vectors

idxIn = spark.createDataFrame([

(Vectors.dense(1, 2, 3),1),

(Vectors.dense(2, 5, 6),2),

(Vectors.dense(1, 8, 9),3)

]).toDF("features", "label")

indxr = 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 categorical variables is only half of the story. One-hot encoding is an extremely common data transformation performed after indexing categorical variables. This is because 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, you will notice that 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, which makes no sense in the case of the current categories. To avoid this, we use OneHotEncoder, which will convert each distinct value to a Boolean flag (1 or 0) as a component in a vector. When we encode the color value, then we can see these are no longer ordered, making them easier for downstream models (e.g., a linear model) to process:

*// in Scala*

**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()

*# in Python*

**from** **pyspark.ml.feature** **import** OneHotEncoder, StringIndexer

lblIndxr = StringIndexer().setInputCol("color").setOutputCol("colorInd")

colorLab = lblIndxr.fit(simpleDF).transform(simpleDF.select("color"))

ohe = 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 is always tricky input because it often requires lots of manipulation to map to a format that a machine learning model will be able to use effectively. There are generally two kinds of texts you’ll see: free-form text and string categorical variables. This section primarily focuses on free-form text because we already discussed categorical variables.

**Tokenizing Text**

Tokenization is the process of converting free-form text into a list of “tokens” or individual words. The easiest way to do this is by using the Tokenizer class. This transformer will take a string of words, separated by whitespace, and convert them into an array of words. For example, in our dataset we might want to convert the Description field into a list of tokens.

*// in Scala*

**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**)

*# in Python*

**from** **pyspark.ml.feature** **import** Tokenizer

tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")

tokenized = tkn.transform(sales.select("Description"))

tokenized.show(20, 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] |

+-----------------------------------+------------------------------------------+

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:

*// in Scala*

**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**)

*# in Python*

**from** **pyspark.ml.feature** **import** RegexTokenizer

rt = RegexTokenizer()\

.setInputCol("Description")\

.setOutputCol("DescOut")\

.setPattern(" ")\

.setToLowercase(True)

rt.transform(sales.select("Description")).show(20, 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] |

+-----------------------------------+------------------------------------------+

Another way of using the RegexTokenizer is to use it to output values matching the provided pattern instead of using it as a gap. We do this by setting the gaps parameter to false. Doing this with a space as a pattern returns all the spaces, which is not too useful, but if we made our pattern capture individual words, we could return those:

*// in Scala*

**import** **org.apache.spark.ml.feature.RegexTokenizer**

**val** rt **=** **new** **RegexTokenizer**()

.setInputCol("Description")

.setOutputCol("DescOut")

.setPattern(" ")

.setGaps(**false**)

.setToLowercase(**true**)

rt.transform(sales.select("Description")).show(**false**)

*# in Python*

**from** **pyspark.ml.feature** **import** RegexTokenizer

rt = RegexTokenizer()\

.setInputCol("Description")\

.setOutputCol("DescOut")\

.setPattern(" ")\

.setGaps(False)\

.setToLowercase(True)

rt.transform(sales.select("Description")).show(20, False)

+-----------------------------------+------------------+

|Description DescOut |

+-----------------------------------+------------------+

|RABBIT NIGHT LIGHT |[ , ] |

|DOUGHNUT LIP GLOSS |[ , , ] |

...

|AIRLINE BAG VINTAGE WORLD CHAMPION |[ , , , , ] |

|AIRLINE BAG VINTAGE JET SET BROWN |[ , , , , ] |

+-----------------------------------+------------------+

**Removing Common Words**

A common task after tokenization is to filter *stop words*, common words that are not relevant in many kinds of analysis and should thus be removed. Frequently occurring stop words in English include “the,” “and,” and “but.” Spark contains a list of default stop words you can see by calling the following method, which can be made case insensitive if necessary (as of Spark 2.2, supported languages for stopwords are “danish,” “dutch,” “english,” “finnish,” “french,” “german,” “hungarian,” “italian,” “norwegian,” “portuguese,” “russian,” “spanish,” “swedish,” and “turkish”):

*// in Scala*

**import** **org.apache.spark.ml.feature.StopWordsRemover**

**val** englishStopWords **=** **StopWordsRemover**.loadDefaultStopWords("english")

**val** stops **=** **new** **StopWordsRemover**()

.setStopWords(englishStopWords)

.setInputCol("DescOut")

stops.transform(tokenized).show()

*# in Python*

**from** **pyspark.ml.feature** **import** StopWordsRemover

englishStopWords = StopWordsRemover.loadDefaultStopWords("english")

stops = StopWordsRemover()\

.setStopWords(englishStopWords)\

.setInputCol("DescOut")

stops.transform(tokenized).show()

The following output shows how this works:

+--------------------+--------------------+------------------------------------+

| Description| DescOut|StopWordsRemover\_4ab18...6ed\_\_output|

+--------------------+--------------------+------------------------------------+

...

|SET OF 4 KNICK KN...|[set, of, 4, knic...| [set, 4, knick, k...|

...

+--------------------+--------------------+------------------------------------+

Notice how the word of is removed in the output column. That’s because it’s such a common word that it isn’t relevant to any downstream manipulation and simply adds noise to our dataset.

**Creating Word Combinations**

Tokenizing our strings and filtering stop words leaves us with a clean set of words to use as features. It is often of interest to look at combinations of words, usually by looking at colocated words. Word combinations are technically referred to as *n-grams*—that is, sequences of words of length *n*. An *n*-gram of length 1 is called a *unigrams*; those of length 2 are called *bigrams*, and those of length 3 are called *trigrams* (anything above those are just four-gram, five-gram, etc.), Order matters with *n*-gram creation, so converting a sentence with three words into bigram representation would result in two bigrams. The goal when creating *n*-grams is to better capture sentence structure and more information than can be gleaned by simply looking at all words individually. Let’s create some *n*-grams to illustrate this concept.

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

* “Big Data”
* “Data Processing”
* “Processing Made”
* “Made Simple”

While the trigrams are:

* “Big Data Processing”
* “Data Processing Made”
* “Procesing Made Simple”

With *n*-grams, we can look at sequences of words that commonly co-occur and use them as inputs to a machine learning algorithm. These can create better features than simply looking at all of the words individually (say, tokenized on a space character):

*// in Scala*

**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**)

*# in Python*

**from** **pyspark.ml.feature** **import** NGram

unigram = NGram().setInputCol("DescOut").setN(1)

bigram = 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**

Once you have word features, it’s time to start counting instances of words and word combinations for use in our models. The simplest way is just to include binary counts of a word in a given document (in our case, a row). Essentially, we’re measuring whether or not each row contains a given word. This is a simple way to normalize for document sizes and occurrence counts and get numerical features that allow us to classify documents based on content. In addition, we can count words using a CountVectorizer, or reweigh them according to the prevalence of a given word in all the documents using a TF–IDF transformation (discussed next).

A CountVectorizer operates on our tokenized data and does two things:

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.

Conceptually this tranformer treats every row as a *document* and every word as a *term* and the total collection of all terms as the *vocabulary*. These are all tunable parameters, meaning we can set the minimum term frequency (minTF) for the term to be included in the vocabulary (effectively removing rare words from the vocabulary); minimum number of documents a term must appear in (minDF) before being included in the vocabulary (another way to remove rare words from the vocabulary); and finally, the total maximum vocabulary size (vocabSize). Lastly, by default the CountVectorizer will output the counts of a term in a document. To just return whether or not a word exists in a document, we can use setBinary(true). Here’s an example of using CountVectorizer:

*// 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**)

*# in Python*

**from** **pyspark.ml.feature** **import** CountVectorizer

cv = CountVectorizer()\

.setInputCol("DescOut")\

.setOutputCol("countVec")\

.setVocabSize(500)\

.setMinTF(1)\

.setMinDF(2)

fittedCV = cv.fit(tokenized)

fittedCV.transform(tokenized).show(False)

While the output looks a little complicated, it’s actually just a sparse vector that contains the total vocabulary size, the index of the word in the vocabulary, and then the counts of that particular word:

+---------------------------------+--------------------------------------------+

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**

Another way to approach the problem of converting text into a numerical representation is to use term frequency–inverse document frequency (TF–IDF). In simplest terms, *TF–IDF* measures how often a word occurs in each document, weighted according to how many documents that word occurs in. The result is that 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 because of its prevalence while a more specialized word like “streaming” would occur in fewer documents and thus would be weighted higher. In a way, TF–IDF helps find documents that share similar topics. Let’s take a look at an example—first, we’ll inspect some of the documents in our data containing the word “red”:

*// in Scala*

**val** tfIdfIn **=** tokenized

.where("array\_contains(DescOut, 'red')")

.select("DescOut")

.limit(10)

tfIdfIn.show(**false**)

*# 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] |

+---------------------------------------+

We can see some overlapping words in these documents, but these words provide at least a rough topic-like representation. Now let’s input that into TF–IDF. To do this, we’re going to hash each word and convert it to a numerical representation, and then weigh each word in the voculary according to the inverse document frequency. Hashing is a similar process as CountVectorizer, but is irreversible—that is, from our output index for a word, we cannot get our input word (multiple words might map to the same output index):

*// 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)

*# 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)

*// in Scala*

idf.fit(tf.transform(tfIdfIn)).transform(tf.transform(tfIdfIn)).show(**false**)

*# in Python*

idf.fit(tf.transform(tfIdfIn)).transform(tf.transform(tfIdfIn)).show(10, False)

While the output is too large to include here, notice that a certain value is assigned to “red” and that this value appears in every document. Also note that this term is weighted extremely low because it appears in every document. The output format is a sparse Vector we can subsequently input into a machine learning model in a form like this:

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

This vector is represented using three different values: the total vocabulary size, the hash of every word appearing in the document, and the weighting of each of those terms. This is similar to the CountVectorizer output.

**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, so we can then make generalizations about the words themselves. This model is easy to train and use, and has been shown to be useful in a number of natural language processing applications, including entity recognition, disambiguation, parsing, tagging, and machine translation.

Word2Vec is notable for capturing relationships between words based on their semantics. For example, if v~king, v~queen, v~man, and v~women represent the vectors for those four words, then we will often get a representation where v~king − v~man + v~woman ~= v~queen. To do this, Word2Vec uses a technique called “skip-grams” to convert a sentence of words into a vector representation (optionally of a specific size). It does this by building a vocabulary, and then for every sentence, it removes a token and trains the model to predict the missing token in the "*n*-gram” representation. Word2Vec works best with continuous, free-form text in the form of tokens.

Here’s a simple example from the documentation:

*// in Scala*

**import** **org.apache.spark.ml.feature.Word2Vec**

**import** **org.apache.spark.ml.linalg.Vector**

**import** **org.apache.spark.sql.Row**

*// Input data: Each row is a bag of words from a sentence or document.*

**val** documentDF **=** spark.createDataFrame(**Seq**(

"Hi I heard about Spark".split(" "),

"I wish Java could use case classes".split(" "),

"Logistic regression models are neat".split(" ")

).map(**Tuple1**.apply)).toDF("text")

*// Learn a mapping from words to Vectors.*

**val** word2Vec **=** **new** **Word2Vec**()

.setInputCol("text")

.setOutputCol("result")

.setVectorSize(3)

.setMinCount(0)

**val** model **=** word2Vec.fit(documentDF)

**val** result **=** model.transform(documentDF)

result.collect().foreach { **case** **Row**(text**:** **Seq**[**\_**], features**:** **Vector**) **=>**

println(s"Text: [${text.mkString(", ")}] => \nVector: $features\n")

}

*# in Python*

**from** **pyspark.ml.feature** **import** Word2Vec

*# Input data: Each row is a bag of words from a sentence or document.*

documentDF = spark.createDataFrame([

("Hi I heard about Spark".split(" "), ),

("I wish Java could use case classes".split(" "), ),

("Logistic regression models are neat".split(" "), )

], ["text"])

*# Learn a mapping from words to Vectors.*

word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text",

outputCol="result")

model = word2Vec.fit(documentDF)

result = model.transform(documentDF)

**for** row **in** result.collect():

text, vector = row

**print**("Text: [%s] => **\n**Vector: %s**\n**" % (", ".join(text), str(vector)))

Text: [Hi, I, heard, about, Spark] =>

Vector: [-0.008142343163490296,0.02051363289356232,0.03255096450448036]

Text: [I, wish, Java, could, use, case, classes] =>

Vector: [0.043090314205203734,0.035048123182994974,0.023512658663094044]

Text: [Logistic, regression, models, are, neat] =>

Vector: [0.038572299480438235,-0.03250147425569594,-0.01552378609776497]

Spark’s Word2Vec implementation includes a variety of tuning parameters that can be found in [the documentation](http://bit.ly/2DRnljk).

**Feature Manipulation**

While nearly every transformer in ML manipulates the feature space in some way, the following algorithms and tools are automated means of either expanding the input feature vectors or reducing them to a lower number of dimensions.

**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. The power of PCA is that it can create a smaller set of more meaningful features to be input into your model, at the potential cost of interpretability.

You’d want to use PCA if you have a large input dataset and want to reduce the total number of features you have. This frequently comes up in text analysis where the entire feature space is massive and many of the features are largely irrelevant. Using PCA, we can find the most important combinations of features and only include those in our machine learning model. PCA takes a parameter 𝘬, specifying the number of output features to create. Generally, this should be much smaller than your input vectors’ dimension.

**NOTE**

Picking the right 𝘬 is nontrivial and there’s no prescription we can give. Check out the relevant chapters in [ESL](http://statweb.stanford.edu/~tibs/ElemStatLearn/) and [ISL](http://www-bcf.usc.edu/~gareth/ISL/) for more information.

Let’s train PCA with a 𝘬 of 2:

*// in Scala*

**import** **org.apache.spark.ml.feature.PCA**

**val** pca **=** **new** **PCA**().setInputCol("features").setK(2)

pca.fit(scaleDF).transform(scaleDF).show(**false**)

*# in Python*

**from** **pyspark.ml.feature** **import** PCA

pca = PCA().setInputCol("features").setK(2)

pca.fit(scaleDF).transform(scaleDF).show(20, False)

+---+--------------+------------------------------------------+

|id |features |pca\_7c5c4aa7674e\_\_output |

+---+--------------+------------------------------------------+

|0 |[1.0,0.1,-1.0]|[0.0713719499248418,-0.4526654888147822] |

...

|1 |[3.0,10.1,3.0]|[-10.872398139848944,0.030962697060150646]|

+---+--------------+------------------------------------------+

**Interaction**

In some cases, you might have domain knowledge about specific variables in your dataset. For example, you might know that a certain interaction between the two variables is an important variable to include in a downstream estimator. The feature transformer Interaction allows you to create an interaction between two variables manually. It just multiplies the two features together—something that a typical linear model would not do for every possible pair of features in your data. This transformer is currently only available directly in Scala but can be called from any language using the RFormula. We recommend users just use RFormula instead of manually creating interactions.

**Polynomial Expansion**

Polynomial expansion is used to generate interaction variables of all the input columns. With polynomial expansion, we specify to what degree we would like to see various interactions. For example, for a degree-2 polynomial, Spark takes every value in our 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). If we have three input features, we’ll get nine output features (3x3). If we use a third-degree polynomial, we’ll get 27 output features (3x3x3) and so on. This transformation is useful when you want to see interactions between particular features but aren’t necessarily sure about which interactions to consider.

**WARNING**

Polynomial expansion can greatly increase your feature space, leading to both high computational costs and overfitting. Use it with caution, especially for higher degrees.

Here’s an example of a second degree polynomial:

*// in Scala*

**import** **org.apache.spark.ml.feature.PolynomialExpansion**

**val** pe **=** **new** **PolynomialExpansion**().setInputCol("features").setDegree(2)

pe.transform(scaleDF).show(**false**)

*# in Python*

**from** **pyspark.ml.feature** **import** PolynomialExpansion

pe = PolynomialExpansion().setInputCol("features").setDegree(2)

pe.transform(scaleDF).show()

+---+--------------+-----------------------------------------------------------+

|id |features |poly\_9b2e603812cb\_\_output |

+---+--------------+-----------------------------------------------------------+

|0 |[1.0,0.1,-1.0]|[1.0,1.0,0.1,0.1,0.010000000000000002,-1.0,-1.0,-0.1,1.0] |

...

|1 |[3.0,10.1,3.0]|[3.0,9.0,10.1,30.299999999999997,102.00999999999999,3.0... |

+---+--------------+-----------------------------------------------------------+

**Feature Selection**

Often, you will have a large range of possible features and want to select a smaller subset to use for training. For example, many features might be correlated, or using too many features might lead to overfitting. This process is called feature selection. There are a number of ways to evaluate feature importance once you’ve trained a model but another option is to do some rough filtering beforehand. Spark has some simple options for doing that, such as ChiSqSelector.

**ChiSqSelector**

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. It’s often used with categorical data in order to reduce the number of features you will input into your model, as well as to reduce the dimensionality of text data (in the form of frequencies or counts). Since this method is based on the Chi-Square test, there are several different ways we can pick the “best” features. The methods are numTopFeatures, which is ordered by *p*-value; percentile, which takes a proportion of the input features (instead of just the top N features); and fpr, which sets a cut off *p*-value.

We will demonstrate this with the output of the CountVectorizer created earlier in this chapter:

*// in Scala*

**import** **org.apache.spark.ml.feature.**{**ChiSqSelector**, **Tokenizer**}

**val** tkn **=** **new** **Tokenizer**().setInputCol("Description").setOutputCol("DescOut")

**val** tokenized **=** tkn

.transform(sales.select("Description", "CustomerId"))

.where("CustomerId IS NOT NULL")

**val** prechi **=** fittedCV.transform(tokenized)

**val** chisq **=** **new** **ChiSqSelector**()

.setFeaturesCol("countVec")

.setLabelCol("CustomerId")

.setNumTopFeatures(2)

chisq.fit(prechi).transform(prechi)

.drop("customerId", "Description", "DescOut").show()

*# in Python*

**from** **pyspark.ml.feature** **import** ChiSqSelector, Tokenizer

tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")

tokenized = tkn\

.transform(sales.select("Description", "CustomerId"))\

.where("CustomerId IS NOT NULL")

prechi = fittedCV.transform(tokenized)\

.where("CustomerId IS NOT NULL")

chisq = ChiSqSelector()\

.setFeaturesCol("countVec")\

.setLabelCol("CustomerId")\

.setNumTopFeatures(2)

chisq.fit(prechi).transform(prechi)\

.drop("customerId", "Description", "DescOut").show()

**Advanced Topics**

There are several advanced topics surrounding transformers and estimators. Here we touch on the two most common, persisting transformers as well as writing custom ones.

**Persisting Transformers**

Once you’ve used an estimator to configure a transformer, it can be helpful to write it to disk and simply load it when necessary (e.g., for use in another Spark session). We saw this in the previous chapter when we persisted an entire pipeline. To persist a transformer individually, we use the write method on the fitted transformer (or the standard transformer) and specify the location:

*// in Scala*

**val** fittedPCA **=** pca.fit(scaleDF)

fittedPCA.write.overwrite().save("/tmp/fittedPCA")

*# in Python*

fittedPCA = pca.fit(scaleDF)

fittedPCA.write().overwrite().save("/tmp/fittedPCA")

We can then load it back in:

*// in Scala*

**import** **org.apache.spark.ml.feature.PCAModel**

**val** loadedPCA **=** **PCAModel**.load("/tmp/fittedPCA")

loadedPCA.transform(scaleDF).show()

*# in Python*

**from** **pyspark.ml.feature** **import** PCAModel

loadedPCA = PCAModel.load("/tmp/fittedPCA")

loadedPCA.transform(scaleDF).show()

**Writing a Custom Transformer**

Writing a custom transformer can be valuable when you want to encode some of your own business logic in a form that you can fit into an ML Pipeline, pass on to hyperparameter search, and so on. In general you should try to use the built-in modules (e.g., SQLTransformer) as much as possible because they are optimized to run efficiently. But sometimes we do not have that luxury. Let’s create a simple tokenizer to demonstrate:

**import** **org.apache.spark.ml.UnaryTransformer**

**import** **org.apache.spark.ml.util.**{**DefaultParamsReadable**, **DefaultParamsWritable**,

**Identifiable**}

**import** **org.apache.spark.sql.types.**{**ArrayType**, **StringType**, **DataType**}

**import** **org.apache.spark.ml.param.**{**IntParam**, **ParamValidators**}

**class** **MyTokenizer**(**override** **val** uid**:** **String**)

**extends** **UnaryTransformer**[**String**, **Seq**[**String**],

**MyTokenizer**] **with** **DefaultParamsWritable** {

**def** **this**() **=** **this**(**Identifiable**.randomUID("myTokenizer"))

**val** maxWords**:** **IntParam** = **new** **IntParam**(**this**, "maxWords",

"The max number of words to return.",

**ParamValidators**.gtEq(0))

**def** setMaxWords(value**:** **Int**)**:** **this.type** = set(maxWords, value)

**def** getMaxWords**:** **Integer** = $(maxWords)

**override** **protected** **def** createTransformFunc**:** **String** => **Seq**[**String**] **=** (

inputString**:** **String**) **=>** {

inputString.split("\\s").take($(maxWords))

}

**override** **protected** **def** validateInputType(inputType**:** **DataType**)**:** **Unit** = {

require(

inputType == **StringType**, s"Bad input type: $inputType. Requires String.")

}

**override** **protected** **def** outputDataType**:** **DataType** = **new** **ArrayType**(**StringType**,

**true**)

}

*// this will allow you to read it back in by using this object.*

**object** **MyTokenizer** **extends** **DefaultParamsReadable**[**MyTokenizer**]

**val** myT **=** **new** **MyTokenizer**().setInputCol("someCol").setMaxWords(2)

myT.transform(**Seq**("hello world. This text won't show.").toDF("someCol")).show()

It is also possible to write a custom estimator where you must customize the transformation based on the actual input data. However, this isn’t as common as writing a standalone transformer and is therefore not included in this book. A good way to do this is to look at one of the simple estimators we saw before and modify the code to suit your use case. A good place to start might be the [StandardScaler](http://bit.ly/2FkPIn4).

**Conclusion**

This chapter gave a whirlwind tour of many of the most common preprocessing transformations Spark has available. There are several domain-specific ones we did not have enough room to cover (e.g., Discrete Cosine Transform), but you can find more information in the [documentation](http://bit.ly/2pE51jZ). This area of Spark is also constantly growing as the community develops new ones.

Another important aspect of this feature engineering toolkit is consistency. In the previous chapter we covered the pipeline concept, an essential tool to package and train end-to-end ML workflows. In the next chapter we will start going through the variety of machine learning tasks you may have and what algorithms are available for each one.