**Chapter 29. Unsupervised Learning**

This chapter will cover the details of Spark’s available tools for unsupervised learning, focusing specifically on clustering. Unsupervised learning is, generally speaking, used less often than supervised learning because it’s usually harder to apply and measure success (from an end-result perspective). These challenges can become exacerbated at scale. For instance, clustering in high-dimensional space can create odd clusters simply because of the properties of high-dimensional spaces, something referred to as *the curse of dimensionality*. The curse of dimensionality describes the fact that as a feature space expands in dimensionality, it becomes increasingly sparse. This means that the data needed to fill this space for statistically meaningful results increases rapidly with any increase in dimensionality. Additionally, with high dimensions comes more noise in the data. This, in turn, may cause your model to hone in on noise instead of the true factors causing a particular result or grouping. Therefore in the model scalability table, we include computational limits, as well as a set of statistical recommendations. These are heuristics and should be helpful guides, not requirements.

At its core, *unsupervised learning* is trying to discover patterns or derive a concise representation of the underlying structure of a given dataset.

**Use Cases**

Here are some potential use cases. At its core, these patterns might reveal topics, anomalies, or groupings in our data that may not have been obvious beforehand:

Finding anomalies in data

If the majority of values in a dataset cluster into a larger group with several small groups on the outside, those groups might warrant further investigation.

Topic modeling

By looking at large bodies of text, it is possible to find topics that exist across those different documents.

**Model Scalability**

Just like with our other models, it’s important to mention the basic model scalability requirements along with statistical recommendations.

*Table 29-1. Clustering model scalability reference*

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Statistical recommendation** | **Computation limits** | **Training examples** |
| *k*-means | 50 to 100 maximum | Features x clusters < 10 million | No limit |
| Bisecting *k*-means | 50 to 100 maximum | Features x clusters < 10 million | No limit |
| GMM | 50 to 100 maximum | Features x clusters < 10 million | No limit |
| LDA | An interpretable number | 1,000s of topics | No limit |

Let’s get started by loading some example numerical data:

*// in Scala*

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

**val** va **=** **new** **VectorAssembler**()

.setInputCols(**Array**("Quantity", "UnitPrice"))

.setOutputCol("features")

**val** sales **=** va.transform(spark.read.format("csv")

.option("header", "true")

.option("inferSchema", "true")

.load("/data/retail-data/by-day/\*.csv")

.limit(50)

.coalesce(1)

.where("Description IS NOT NULL"))

sales.cache()

*# in Python*

**from** **pyspark.ml.feature** **import** VectorAssembler

va = VectorAssembler()\

.setInputCols(["Quantity", "UnitPrice"])\

.setOutputCol("features")

sales = va.transform(spark.read.format("csv")

.option("header", "true")

.option("inferSchema", "true")

.load("/data/retail-data/by-day/\*.csv")

.limit(50)

.coalesce(1)

.where("Description IS NOT NULL"))

sales.cache()

**k-means**

𝘬*-means* is one of the most popular clustering algorithms. In this algorithm, a user-specified number of clusters (𝘬) are randomly assigned to different points in the dataset. The unassigned points are then “assigned” to a cluster based on their proximity (measured in Euclidean distance) to the previously assigned point. Once this assignment happens, the center of this cluster (called the *centroid*) is computed, and the process repeats. All points are assigned to a particular centroid, and a new centroid is computed. We repeat this process for a finite number of iterations or until convergence (i.e., when our centroid locations stop changing). This does not, however, mean that our clusters are always sensical. For instance, a given “logical” cluster of data might be split right down the middle simply because of the starting points of two distinct clusters. Thus, it is often a good idea to perform multiple runs of 𝘬-means starting with different initializations.

Choosing the right value for 𝘬 is an extremely important aspect of using this algorithm successfully, as well as a hard task. There’s no real prescription for the number of clusters you need, so you’ll likely have to experiment with different values and consider what you would like the end result to be.

For more information on 𝘬-means, see [ISL 10.3](http://www-bcf.usc.edu/~gareth/ISL/) and [ESL 14.3](http://statweb.stanford.edu/~tibs/ElemStatLearn/).

**Model Hyperparameters**

These are configurations that we specify to determine the basic structure of the model:

𝗄

This is the number of clusters that you would like to end up with.

**Training Parameters**

initMode

The initialization mode is the algorithm that determines the starting locations of the centroids. The supported options are random and 𝘬-means|| (the default). The latter is a parallelized variant of the [𝘬-means||](http://theory.stanford.edu/~sergei/papers/kMeansPP-soda.pdf) method. While the details are not within the scope of this book, the thinking behind the latter method is that rather than simply choosing random initialization locations, the algorithm chooses cluster centers that are already well spread out to generate a better clustering.

initSteps

The number of steps for 𝘬-means|| initialization mode. Must be greater than 0. (The default value is 2.)

maxIter

Total number of iterations over the data before stopping. Changing this probably won’t change your results a ton, so don’t make this the first parameter you look to adjust. The default is 20.

tol

Specifies a threshold by which changes in centroids show that we optimized our model enough, and can stop iterating early, before maxIter iterations. The default value is 0.0001.

This algorithm is generally robust to these parameters, and the main trade-off is that running more initialization steps and iterations may lead to a better clustering at the expense of longer training time:

**Example**

*// in Scala*

**import** **org.apache.spark.ml.clustering.KMeans**

**val** km **=** **new** **KMeans**().setK(5)

println(km.explainParams())

**val** kmModel **=** km.fit(sales)

*# in Python*

**from** **pyspark.ml.clustering** **import** KMeans

km = KMeans().setK(5)

**print** km.explainParams()

kmModel = km.fit(sales)

**k-means Metrics Summary**

𝘬-means includes a summary class that we can use to evaluate our model. This class provides some common measures for 𝘬-means success (whether these apply to your problem set is another question). The 𝘬-means summary includes information about the clusters created, as well as their relative sizes (number of examples).

We can also compute the *within set sum of squared errors*, which can help measure how close our values are from each cluster centroid, using computeCost. The implicit goal in 𝘬-means is that we want to minimize the within set sum of squared error, subject to the given number 𝘬 of clusters:

*// in Scala*

**val** summary **=** kmModel.summary

summary.clusterSizes *// number of points*

kmModel.computeCost(sales)

println("Cluster Centers: ")

kmModel.clusterCenters.foreach(println)

*# in Python*

summary = kmModel.summary

**print** summary.clusterSizes *# number of points*

kmModel.computeCost(sales)

centers = kmModel.clusterCenters()

**print**("Cluster Centers: ")

**for** center **in** centers:

**print**(center)

**Bisecting k-means**

Bisecting 𝘬-means is a variant of 𝘬-means. The core difference is that instead of clustering points by starting “bottom-up” and assigning a bunch of different groups in the data, this is a top-down clustering method. This means that it will start by creating a single group and then splitting that group into smaller groups in order to end up with the 𝘬 number of clusters specified by the user. This is usually a faster method than 𝘬-means and will yield different results.

**Model Hyperparameters**

These are configurations that we specify to determine the basic structure of the model:

𝗄

This is the number of clusters that you would like to end up with.

**Training Parameters**

minDivisibleClusterSize

The minimum number of points (if greater than or equal to 1.0) or the minimum proportion of points (if less than 1.0) of a divisible cluster. The default is 1.0, meaning that there must be at least one point in each cluster.

maxIter

Total number of iterations over the data before stopping. Changing this probably won’t change your results a ton, so don’t make this the first parameter you look to adjust. The default is 20.

Most of the parameters in this model should be tuned in order to find the best result. There’s no rule that applies to all datasets.

**Example**

*// in Scala*

**import** **org.apache.spark.ml.clustering.BisectingKMeans**

**val** bkm **=** **new** **BisectingKMeans**().setK(5).setMaxIter(5)

println(bkm.explainParams())

**val** bkmModel **=** bkm.fit(sales)

*# in Python*

**from** **pyspark.ml.clustering** **import** BisectingKMeans

bkm = BisectingKMeans().setK(5).setMaxIter(5)

bkmModel = bkm.fit(sales)

**Bisecting k-means Summary**

Bisecting 𝘬-means includes a summary class that we can use to evaluate our model, that is largely the same as the 𝘬-means summary. This includes information about the clusters created, as well as their relative sizes (number of examples):

*// in Scala*

**val** summary **=** bkmModel.summary

summary.clusterSizes *// number of points*

kmModel.computeCost(sales)

println("Cluster Centers: ")

kmModel.clusterCenters.foreach(println)

*# in Python*

summary = bkmModel.summary

**print** summary.clusterSizes *# number of points*

kmModel.computeCost(sales)

centers = kmModel.clusterCenters()

**print**("Cluster Centers: ")

**for** center **in** centers:

**print**(center)

**Gaussian Mixture Models**

*Gaussian mixture models* (GMM) are another popular clustering algorithm that makes different assumptions than bisecting 𝘬-means or 𝘬-means do. Those algorithms try to group data by reducing the sum of squared distances from the center of the cluster. Gaussian mixture models, on the other hand, assume that each cluster produces data based upon random draws from a Gaussian distribution. This means that clusters of data should be less likely to have data at the edge of the cluster (reflected in the Guassian distribution) and much higher probability of having data in the center. Each Gaussian cluster can be of arbitrary size with its own mean and standard deviation (and hence a possibly different, ellipsoid shape). There are still 𝘬 user-specified clusters that will be created during training.

A simplified way of thinking about Gaussian mixture models is that they’re like a soft version of 𝘬-means. 𝘬-means creates very rigid clusters—each point is only within one cluster. GMMs allow for a more nuanced cluster associated with probabilities, instead of rigid boundaries.

For more information, see [ESL](http://statweb.stanford.edu/~tibs/ElemStatLearn/) 14.3.

**Model Hyperparameters**

These are configurations that we specify to determine the basic structure of the model:

𝗄

This is the number of clusters that you would like to end up with.

**Training Parameters**

maxIter

Total number of iterations over the data before stopping. Changing this probably won’t change your results a ton, so don’t make this the first parameter you look to adjust. The default is 100.

tol

This value simply helps us specify a threshold by which changes in parameters show that we optimized our weights enough. A smaller value can lead to higher accuracy at the cost of performing more iterations (although never more than maxIter). The default value is 0.01.

As with our 𝘬-means model, these training parameters are less likely to have an impact than the number of clusters, 𝘬.

**Example**

*// in Scala*

**import** **org.apache.spark.ml.clustering.GaussianMixture**

**val** gmm **=** **new** **GaussianMixture**().setK(5)

println(gmm.explainParams())

**val** model **=** gmm.fit(sales)

*# in Python*

**from** **pyspark.ml.clustering** **import** GaussianMixture

gmm = GaussianMixture().setK(5)

**print** gmm.explainParams()

model = gmm.fit(sales)

**Gaussian Mixture Model Summary**

Like our other clustering algorithms, Gaussian mixture models include a summary class to help with model evaluation. This includes information about the clusters created, like the weights, the means, and the covariance of the Gaussian mixture, which can help us learn more about the underlying structure inside of our data:

*// in Scala*

**val** summary **=** model.summary

model.weights

model.gaussiansDF.show()

summary.cluster.show()

summary.clusterSizes

summary.probability.show()

*# in Python*

summary = model.summary

**print** model.weights

model.gaussiansDF.show()

summary.cluster.show()

summary.clusterSizes

summary.probability.show()

**Latent Dirichlet Allocation**

*Latent Dirichlet Allocation* (LDA) is a hierarchical clustering model typically used to perform topic modelling on text documents. LDA tries to extract high-level topics from a series of documents and keywords associated with those topics. It then interprets each document as having a variable number of contributions from multiple input topics. There are two implementations that you can use: online LDA and expectation maximization. In general, online LDA will work better when there are more examples, and the expectation maximization optimizer will work better when there is a larger input vocabulary. This method is also capable of scaling to hundreds or thousands of topics.

To input our text data into LDA, we’re going to have to convert it into a numeric format. You can use the CountVectorizer to achieve this.

**Model Hyperparameters**

These are configurations that we specify to determine the basic structure of the model:

𝗄

The total number of topics to infer from the data. The default is 10 and must be a positive number.

docConcentration

Concentration parameter (commonly named “alpha”) for the prior placed on documents’ distributions over topics (“theta”). This is the parameter to a Dirichlet distribution, where larger values mean more smoothing (more regularization).

If not set by the user, then docConcentration is set automatically. If set to singleton vector [alpha], then alpha is replicated to a vector of length k in fitting. Otherwise, the docConcentration vector must be length 𝘬.

topicConcentration

The concentration parameter (commonly named “beta” or “eta”) for the prior placed on a topic’s distributions over terms. This is the parameter to a symmetric Dirichlet distribution. If not set by the user, then topicConcentration is set automatically.

**Training Parameters**

These are configurations that specify how we perform training:

maxIter

Total number of iterations over the data before stopping. Changing this probably won’t change your results a ton, so don’t make this the first parameter you look to adjust. The default is 20.

optimizer

This determines whether to use EM or online training optimization to determine the LDA model. The default is online.

learningDecay

Learning rate, set as an exponential decay rate. This should be between (0.5, 1.0] to guarantee asymptotic convergence. The default is 0.51 and only applies to the online optimizer.

learningOffset

A (positive) learning parameter that downweights early iterations. Larger values make early iterations count less. The default is 1,024.0 and only applies to the online optimizer.

optimizeDocConcentration

Indicates whether the docConcentration (Dirichlet parameter for document-topic distribution) will be optimized during training. The default is true but only applies to the online optimizer.

subsamplingRate

The fraction of the corpus to be sampled and used in each iteration of mini-batch gradient descent, in range (0, 1]. The default is 0.5 and only applies to the online optimizer.

seed

This model also supports specifying a random seed for reproducibility.

checkpointInterval

This is the same checkpoint feature that we saw in [Chapter 26](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch26.html#s6c3---classification).

**Prediction Parameters**

topicDistributionCol

The column that will hold the output of the topic mixture distribution for each document.

**Example**

*// in Scala*

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

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

**val** tokenized **=** tkn.transform(sales.drop("features"))

**val** cv **=** **new** **CountVectorizer**()

.setInputCol("DescOut")

.setOutputCol("features")

.setVocabSize(500)

.setMinTF(0)

.setMinDF(0)

.setBinary(**true**)

**val** cvFitted **=** cv.fit(tokenized)

**val** prepped **=** cvFitted.transform(tokenized)

*# in Python*

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

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

tokenized = tkn.transform(sales.drop("features"))

cv = CountVectorizer()\

.setInputCol("DescOut")\

.setOutputCol("features")\

.setVocabSize(500)\

.setMinTF(0)\

.setMinDF(0)\

.setBinary(True)

cvFitted = cv.fit(tokenized)

prepped = cvFitted.transform(tokenized)

*// in Scala*

**import** **org.apache.spark.ml.clustering.LDA**

**val** lda **=** **new** **LDA**().setK(10).setMaxIter(5)

println(lda.explainParams())

**val** model **=** lda.fit(prepped)

*# in Python*

**from** **pyspark.ml.clustering** **import** LDA

lda = LDA().setK(10).setMaxIter(5)

**print** lda.explainParams()

model = lda.fit(prepped)

After we train the model, you will see some of the top topics. This will return the term indices, and we’ll have to look these up using the CountVectorizerModel that we trained in order to find out the true words. For instance, when we trained on the data our top 3 topics were hot, home, and brown after looking them up in our vocabulary:

*// in Scala*

model.describeTopics(3).show()

cvFitted.vocabulary

*# in Python*

model.describeTopics(3).show()

cvFitted.vocabulary

These methods result in detailed information about the vocabulary used as well as the emphasis on particular terms. These can be helpful for better understanding the underlying topics. Due to space constraints, we can’t show this output. Using similar APIs, we can get some more technical measures like the log likelihood and perplexity. The goal of these tools is to help you optimize the number of topics, based on your data. When using perplexity in your success criteria, you should apply these metrics to a holdout set to reduce the overall perplexity of the model. Another option is to optimize to increase the log likelihood value on the holdout set. We can calculate each of these by passing a dataset into the following functions: model.logLikelihood and model.logPerplexity.

**Conclusion**

This chapter covered the most popular algorithms that Spark includes for unsupervised learning. The next chapter will bring us out of MLlib and talk about some of the advanced analytics ecosystem that has grown outside of Spark.