

MODULE 1 MLIB DAT ATYPES

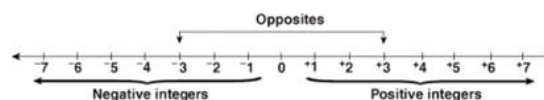
Lesson Objectives

- Local Vectors supported by Spark MLlib:
 - What they are
 - Types supported by Mllib –
 - Dense and Sparse
- Labeled point:
 - What is a labeled point?
 - Role in classifying machine learning data

Spark MLlib Data Types



Local Vector



Double	8 bytes	Range: -1.7e308 to +1.7e308	15 Digits of Precision
--------	---------	-----------------------------	------------------------

Get indices for the value 3:

5, 1, 3, 6, 3, 2, 7, 2, 3, 1, 3 Dense

0, 1, 2, 3

1. 0. 0. 0. 0. 0. 3.

size : 7

sparse : { indices : 0 6
values : 1. 3.

Dense Vector

dense

/dens/ ⓘ

adjective

1. closely compacted in substance.
"dense volcanic rock"

synonyms: thick, heavy, opaque, soupy, murky, smoggy, More

Dense : 2. 4. 1. 0. 3. 5. 2. 0.

 SciPy
NumPy

 python™
List

Sparse Vector

sparse

/spärs/ ⓘ

adjective

thinly dispersed or scattered.
"areas of sparse population"

synonyms: scant, scanty, scattered, scarce, infrequent, few and far between; More


Indices: 2, 6

Values: 1, 7

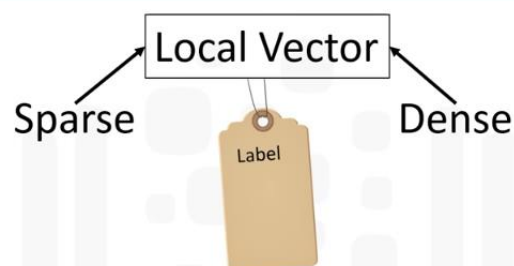
Sparse : 0. 0. 1. 0. 0. 0. 7. 0.

 Spark
MLlib

SparseVector

 SciPy
csc_matrix
(single column)

Labeled Points



Supervised Machine Learning

Sparse Data

LIBSVM = Library for Support Vector Machines



Hello and welcome.

My name is Deborah.

In this lesson we will look at what a local vector is, and the two types that are supported in Spark MLlib.

These are Dense and Sparse type local vectors.

We will explore the labeled point: what it is, and what its role is in classifying machine learning data.

Spark MLlib supports local vectors and matrices that are stored on a single machine, as well as distributed matrices that are backed by one or more resilient distributed datasets, or RDD's.

Let's take a closer look at Local Vectors to help define what they are.

A Local Vector contains integer-type and double-type values, as well as 0-based indices.

As we noted earlier, Spark MLlib supports both dense and sparse local vectors.

A Dense Vector is backed by a double array representing its entry values.

By definition, dense means closely compacted.

Therefore, a dense vector is one that contains many values, or values that are not zero's.

NumPy arrays and Python Lists are recognized by MLlib as dense vectors making them very easy to work with.

Generally, NumPy arrays are recommended over lists for efficiency.

By contrast, sparse vectors are backed by two parallel arrays.

These arrays represent the vector's indices and its values.

By definition, sparse means thinly dispersed or scattered.

Therefore, a majority of the values in a sparse vector will be zero's.

MLlib recognizes MLlib's SparseVector type vectors and SciPy's CSC underscore matrix types as sparse vectors.

Now, let's look at Labeled Points.

By their name, we can almost guess what they are.

A Labeled Point is a local vector that has an associated label or response.

They can be either sparse or dense vectors.

Labeled Points are used in supervised machine learning algorithms.

Labeled Points can be used for binary classification, where a negative classification is given a value of zero, and a positive classification receives a value of one.

For example, suppose that we wish to classify a group of objects as either (not a person), or (a person).

If an object in our group is a person, that object would receive a classification of one. Anything else would be classified as zero.

Labeled Points can also be used for multiclass classifications.

Here, labels start from zero and increment by one until all of the classification values are met.

For example, suppose we want to classify a group of objects based on their type of mammal.

We can classify a person as zero, a dog as one, a cat as two, a monkey as three, and so on, until all of the classifications that we want are represented.

Now let's take a closer look at Sparse Data.

Remember that by definition sparse means thinly dispersed or scattered.

For example, training data is commonly sparse.

Spark MLlib supports the reading of training data examples stored in LIBSVM format.

LIBSVM stands for Library for Support Vector Machines, and is the default text format.

A form of sparse data is a sparse matrix, which may look like this.

Here, each line represents a labeled sparse feature vector.

The indices are one-based and ordered in ascending order.

Once sparse data is loaded, the feature indices are converted to a zero-based index.

After completing this lesson, you should be able to explain what a local vector is, and the two types that are supported in MLlib.

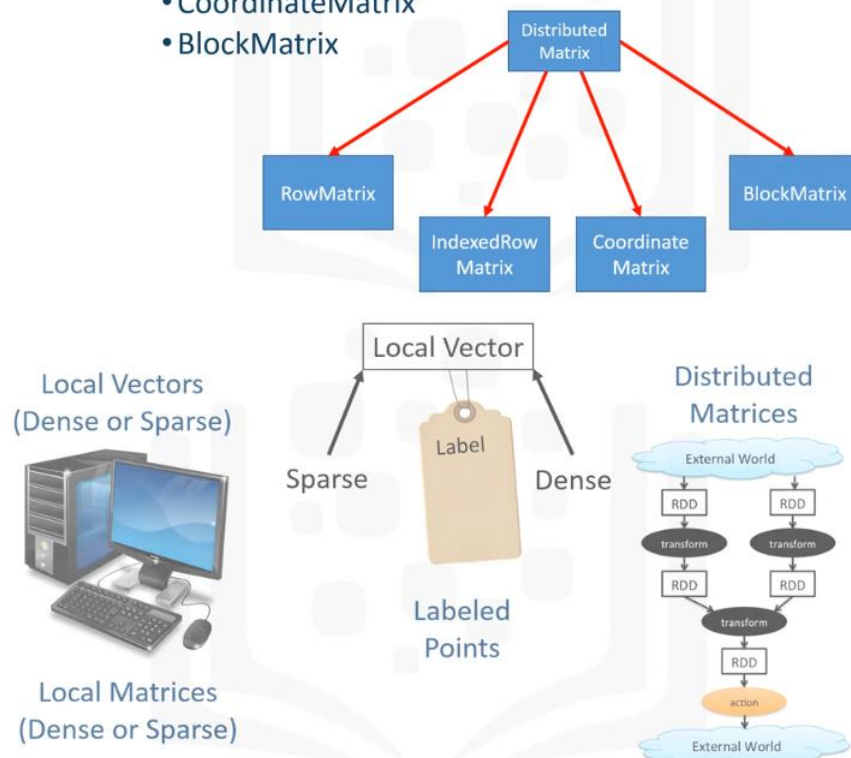
You should understand what labeled points are and their role in classifying machine learning data.

Thank you!

Spark MLlib Distributed Matrices

- Distributed Matrices:

- RowMatrix
- IndexedRowMatrix
- CoordinateMatrix
- BlockMatrix



Review: Row and Column Pointers

9	8	0
0	4	2
1	0	1

Row-Major: [9, 8, 0|0, 4, 2|1, 0, 1]

Row Index: [0, 0|1, 1|2, 2]

0 0 1 2 3 4 5 6

Row Index Pointers: [0, 2, 4, 6]

Column-Major: [9, 0, 1|8, 4, 0|0, 2, 1]

Column Index: [0, 0|1, 1|2, 2]

0 0 1 2 3 4 5 6

Column Index Pointers: [0, 2, 4, 6]

Hello and welcome.

My name is Deborah.

In this lesson we will examine the different

data types that are supported by Spark MLlib. First we will briefly review Local Vectors, Labeled Points, and Local Matrices. These were covered in detail in other lessons.

Then we will explore the different types of distributed matrices that are supported by Spark MLlib, focusing on the RowMatrix, the IndexedRowMatrix, and the CoordinateMatrix.

The BlockMatrix is covered in another lesson. Recall that Spark MLlib supports local vectors, labeled points, and local matrices that are generally stored on a single machine. However,

Spark MLlib also supports distributed matrices that are backed by one or more resilient distributed

datasets, or RDD's. Remember that the local matrix may be either dense or sparse, and that Spark MLlib supports Column-major ordering and Row-major ordering.

This brings us to our current topic: Distributed Matrices, of which Spark MLlib supports 4 types.

These are RowMatrix, IndexedRowMatrix, CoordinateMatrix, and BlockMatrix.

The BlockMatrix is covered in another lesson. Distributed matrices are stored in distributive form in one or more resilient distributed datasets, or RDD's.

Note that the RDD's must be deterministic since the matrix size is cached. Working with non-deterministic RDD's may result in errors. Distributed matrices feature Long-type row indices, Long-type column indices, and Double-type values.

It's also important to choose the correct format to store large, distributed matrices, since converting them to different formats can result in inefficiency and added expense.

We looked at Row Pointers and Column Pointers in a previous lesson. We mention them here

as a reminder of what they are and how they relate to matrices. Please review this topic if necessary. A RowMatrix is a Row-oriented distributed matrix without meaningful row indices. It is backed by an RDD of its rows, where each row is a local vector. Because of this, however, the number of columns are limited by the integer

range. Next is the IndexedRowMatrix. It is similar

to a RowMatrix. However the IndexedRowMatrix uses meaningful row indices. It is backed by an RDD of indexed rows, meaning that each row is represented by its long-type index and a local vector. An IndexedRowMatrix can be created in two ways. The first is by using an RDD of IndexedRow with syntax similar to this.

The second way is to simply use a tuple containing the long-type index and a vector containing

a row of elements. An IndexedRowMatrix can be converted into

a RowMatrix by dropping its indices. This can be done by using the `.toRowMatrix()` function.

Next is the Coordinate Matrix. This type of distributed matrix is backed by an RDD of its entries. What does that mean? Imagine a coordinate plane, with the x-axis labeled as *i* and the y-axis labeled as *j*. Let's say we have a data point here that is given a value of 5. To define that data point, we would say that its value of *i* is 3, its value of *j* is 2, and that the point itself has a value of 5.

Now we'll try one more. What are the components that you would use to define this point?

Got

your answer? You should have gotten a value of 1 for *i*, a *j* value of minus 4, and the value of the point equal to 7. Now this is a simple concept, but how does it relate to entries in an RDD? Essentially, a group of entries like these form the

CoordinateMatrix,

in which every entry is a tuple. Each tuple contains 3 elements: the *i* or row index, which is a Long-type, the *j* or column index, which is also a Long-type, and the entry value, which is a Double-type. While the entry can be made using a tuple, it can also be made by using a `MatrixEntry`. A `CoordinateMatrix` should be used if both dimensions of the matrix, or *m* and *n*, are very large and the matrix is very sparse.

Finally, a `CoordinateMatrix` can be converted into a `RowMatrix` by using `.toRowMatrix()`, an `IndexedRowMatrix` by using `.toIndexedRowMatrix()`, or a `BlockMatrix` by using `.toBlockMatrix()`.

We will cover the `BlockMatrix` in a separate lesson.

After completing this lesson, the viewer should be able to recall Local Vectors, Labeled Points, and Local Matrices. You should then be able to explain what a distributed matrix is, and the characteristics of the first three types. Thank You!

- Review Data Types:

- Local Vectors (Dense and Sparse)
- Labeled Points
- Local Matrices

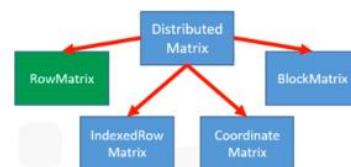
- Explore Distributed Matrices

- RowMatrix
- IndexedRowMatrix
- CoordinateMatrix

Covered in Other Lessons

Review: RowMatrix

- Without meaningful row indices
- Backed by RDD of its rows
- Each row is a local vector
- Column limited by integer range (due to local vector for rows)

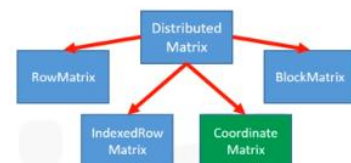
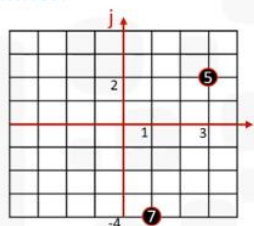


Review: CoordinateMatrix

A distributed matrix backed by an RDD of its entries.

5
 $i = 3$
 $j = 2$
Value = 5

7
 $i = 1$
 $j = -4$
Value = 7



Can be converted into:

RowMatrix w/ `.toRowMatrix()`
IndexedRowMatrix w/ `.toIndexedRowMatrix()`
BlockMatrix w/ `.toBlockMatrix()`

Each entry is a tuple (i : Long, j : Long, value: Double), where i is the row index, j is the column index, and values is the entry value.

MatrixEntry (Long-typed i , Long-typed j , Double-typed value)

Should be used if both of the dimensions (m, n) are very large and the matrix is very sparse.

BlockMatrix

Distributed Matrix backed by RDD of MatrixBlocks.

A MatrixBlock is a tuple with two inputs:

$((\text{blockRowIndex}, \text{blockColIndex}), \text{Matrix})$

Block's Integer Indices

Values at Index

Created from an RDD of sub-matrix blocks, made by using the BlockMatrix function:

`BlockMatrix(Blocks, rowsPerBlock, colsPerBlock)`

Supports methods like add and multiply with other BlockMatrix

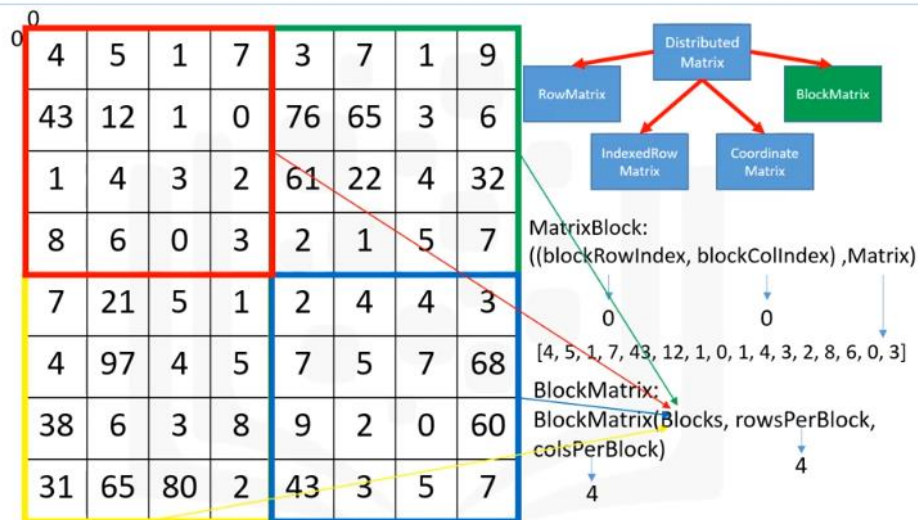
`Matrix1.add(Matrix2)` `Matrix1.subtract(Matrix2)` `Matrix1.multiply(Matrix2)`

`Validate` function checks for proper BlockMatrix set-up:

`Matrix1.validate()`



BlockMatrix



[Courseware](#) , current location

[Course Info](#)

[Discussion](#)

[Wiki](#)

[Resources](#)

[Progress](#)

[About this course](#)

[Module 1 - Spark MLib Data Types](#)

[Learning Objectives](#)

Spark MLlib Data Types (4:02)

Spark MLlib Data Types - Local Matrices (3:04)

Spark MLlib Data Types - Row IndexedRow and Coordinate Distributed Matrices (1:38)

Spark MLlib Data Types - The BlockMatrix (5:40) current section

Lab

Graded Review Questions

Review Questions This content is graded

Module 2 - Review of Algorithms

Module 3 - Spark MLlib Decision Trees and Random Forests

Module 4 - Spark MLlib Clustering

Final Exam

Completion Certificate

Course Survey and Feedback

Spark MLlib Data Types - The BlockMatrix (5:40)

Skip to a navigable version of this video's transcript.

5:27 / 5:40

Maximum Volume.

Skip to end of transcript.

Hello and welcome.

My name is Deborah.

In this lesson we will further examine the different data types that are supported by Spark MLlib.

First we will briefly review Local Vectors, Labeled Points, and Local Matrices.

These were covered in detail in other lessons.

Also, we will review three Distributed Matrices that were covered elsewhere.

All of this review provides a good foundation as we complete our look at distributed matrices

with the most complex of them all - the BlockMatrix.

Throughout this lesson, the viewer is encouraged to look back at other lessons if any of the review material is not clear.

So let's get started!

Recall that Spark MLlib supports local vectors, labeled points, and local matrices that are generally stored on a single machine.

However, Spark MLlib also supports distributed matrices that are backed by one or more resilient

distributed datasets, or RDD's.

Remember that the local matrix may be either dense or sparse, and that Spark MLlib supports

Column-major ordering and Row-major ordering.

This brings us to our current topic: Distributed Matrices and the four types that Spark MLlib supports.

These are RowMatrix, IndexedRowMatrix, CoordinateMatrix, and BlockMatrix.

The BlockMatrix is covered here.

We looked at Row Pointers and Column Pointers in a previous lesson. We mention them here as a reminder of what they are and how they relate to matrices. Here we see the main characteristics of a RowMatrix, one of the four key distributed matrices.

The IndexedRowMatrix is the second distributed matrix covered.

It is similar to a RowMatrix.

However the IndexedRowMatrix uses meaningful row indices.

Next is the Coordinate Matrix.

Remember that this type of distributed matrix consists of a collection of tuples, with each tuple containing the Row index, the Column index, and the entry value.

We have finally made our way to the BlockMatrix!

A BlockMatrix is a distributed matrix that is backed by an RDD of MatrixBlocks, or sub-matrices.

Think of a MatrixBlock as a sub-section of our BlockMatrix.

It is defined by a tuple with two inputs.

The first input is another tuple containing the coordinates that define where our MatrixBlock starts.

In other words, its row and column indices.

The second input is the sub-matrix itself and the values that it contains.

This matrix can be either dense or sparse.

To create our BlockMatrix from an RDD of sub-matrix blocks, we use the BlockMatrix function.

Each MatrixBlock contributes three inputs into our BlockMatrix function.

First is the Blocks input, which represents its indices and matrix values.

Next are the rowsPerBlock input, and the colsPerBlock input.

These two inputs define the number of rows and columns, respectively, contained in each Block.

Therefore, these two values determine the size of the matrix.

A BlockMatrix supports some mathematical operations such as add, subtract, and multiply.

These operations can be called by using add, subtract, and multiply functions.

Finally, the BlockMatrix has a validate function which checks whether the BlockMatrix has been

set up correctly.

It can be called using the validate function.

The BlockMatrix is the most complex of the four distributed matrices that we've encountered.

Let's look at a visual representation of what we just discussed.

This should help to clarify the structure of a BlockMatrix.

This grid represents a data matrix that we wish to treat as a BlockMatrix.

We remember that a BlockMatrix is comprised of a collection of Blocks, as well as information

on the size of those Blocks.

Further, a MatrixBlock is comprised of a row and column index, as well as a set of matrix values.

In our example, we will fill our grid-matrix with some values.

Let's also assume that the top left corner of our matrix has an index of (zero, zero).

The first step in setting up a BlockMatrix is to define our MatrixBlocks.

We define the red block as having a blockRowIndex of zero and a blockColIndex of zero.

Its matrix values are listed in row-major order, and therefore look like this.

We would repeat this process to create the green, yellow and blue MatrixBlocks.

For example, the blue MatrixBlock would have index values of 4 and 4.

We have now represented the entire large matrix in MatrixBlocks.

These can be parallelized with the SparkContext and used as the first parameter in the BlockMatrix function.

To complete the definition of our BlockMatrix, we must define the size of each sub-matrix.

In our example, we define rowsPerBlock as four and colsPerBlock as four.

Our BlockMatrix is now fully defined!

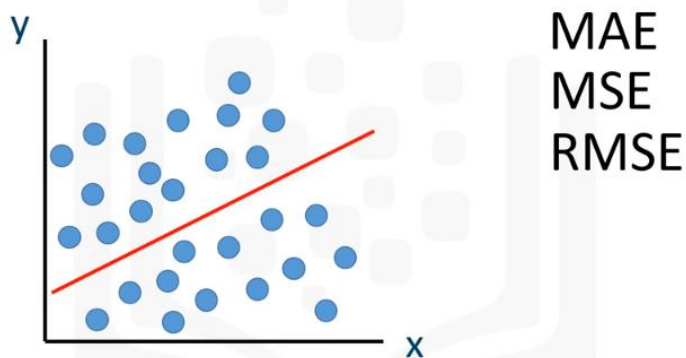
After completing this lesson, the viewer should be able to recall Local Vectors, Labeled Points, and Local Matrices.

You should then be able to explain what a distributed matrix is, and recall the characteristics of each of the four basic types, with the BlockMatrix explained in detail.

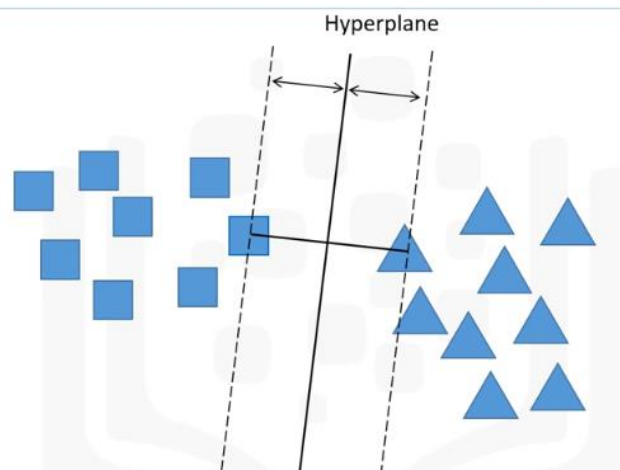
Thank You!

MODULE 2 REVIEWS OF ALGORITHMS

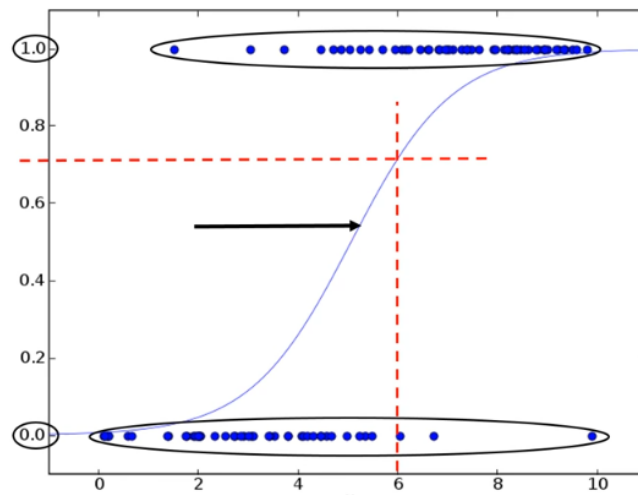
Linear Regression



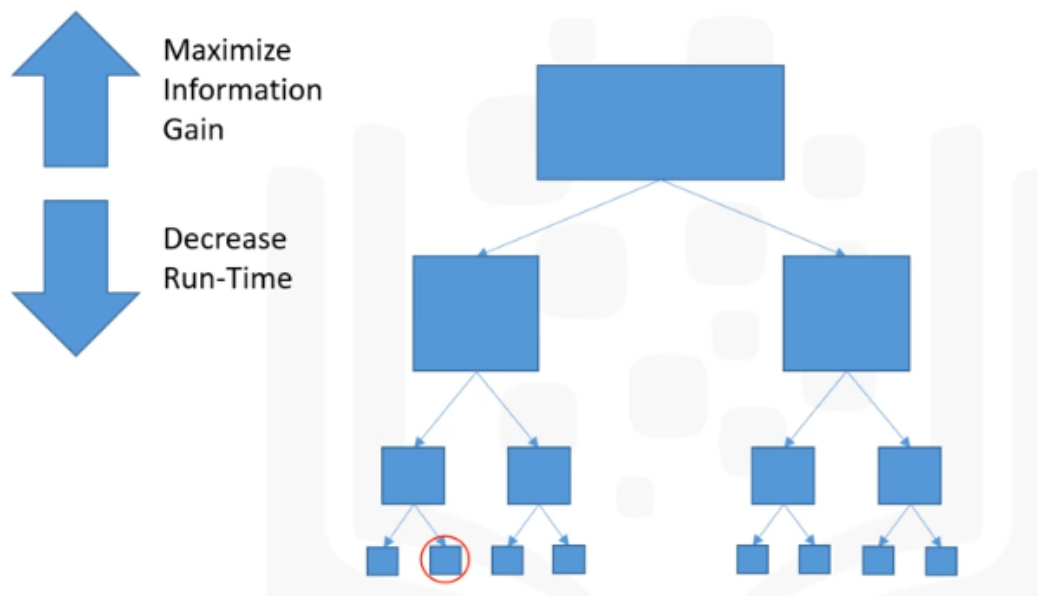
Support Vector Machines (SVM)



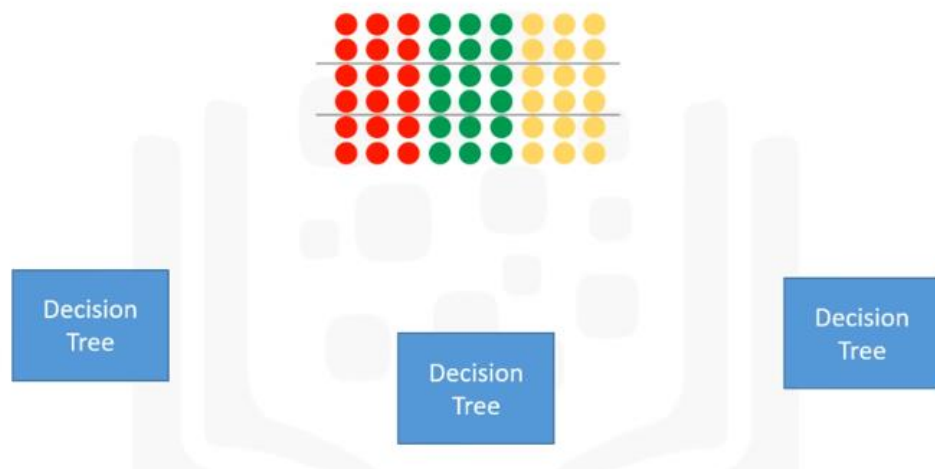
Logistic Regression



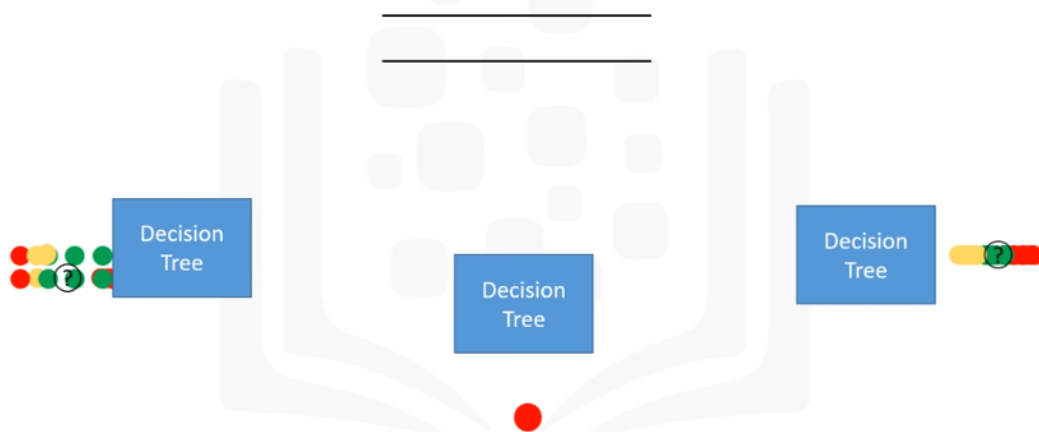
Decision Trees



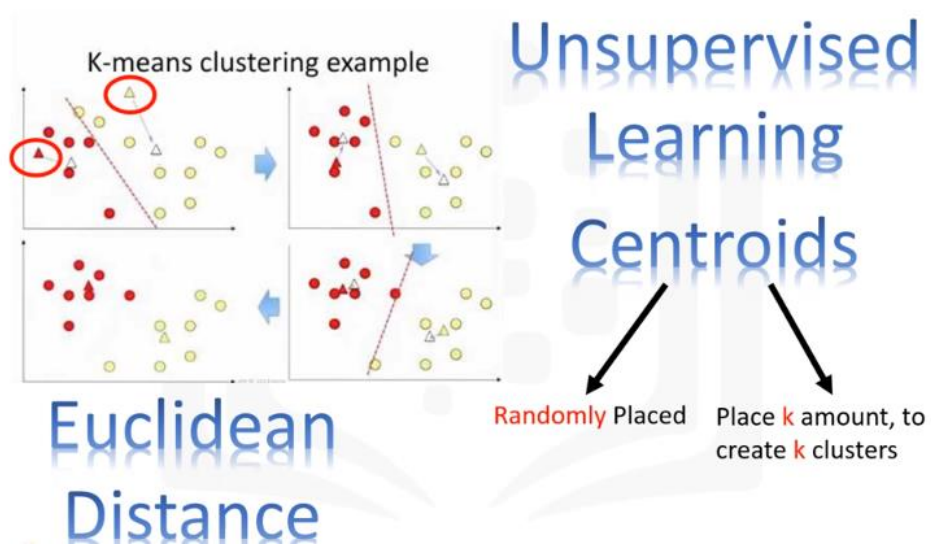
Random Forests



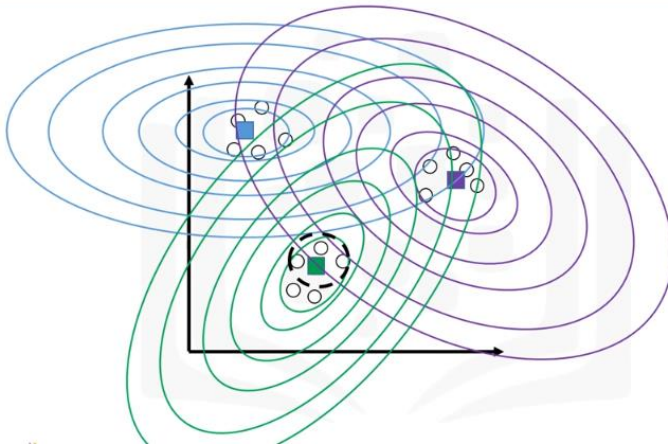
Random Forests



K-Means Clustering



Gaussian Mixture Clustering



Hello and welcome.

My name is Deborah.

In this lesson, we will review the algorithms available with Spark MLlib.

These are: Linear Regression,

Support Vector Machines, or SVM for short, Logistic Regression,

Decision Trees, Random Forests,

K-Means Clustering, and Gaussian Mixture Clustering.

Afterwards, you should determine which algorithms require more review before proceeding with

the next lessons in this course.

We will start our algorithm review with Linear Regression.

Linear Regression assumes that there is a linear relationship between the input of a function and its output.

It is used for making predictions based on previous data.

In general, a Regression algorithm will output a response that is ordered and continuous, where the output may consist of one or more continuous variables.

There are different evaluation methods that can be used for Linear Regression.

These are Mean Absolute Error or MAE, Mean Square Error or MSE, and Root Mean Squared Error or RMSE.

Next let's examine Support Vector Machine algorithms, or SVM's.

These can be used for classifying new data based on training data.

Consider two different classifications of data: triangles and squares.

A Support Vector Machine will find the shortest distance between two points in the two classifications of data.

The algorithm then constructs a vector from the first point to the second point.

A separating line, called a hyperplane, constructed to bisect the vector perpendicularly.

The best hyperplane is one that yields the largest margins between the hyperplane and the two nearest data points.

Logistic Regression algorithms are primarily used for binary response predictions.

A logistic regression tool yields a relationship curve similar to the one shown here.

This curve represents the probability that the independent variable, shown along the X-axis, will be classified as a 0 or 1 along the Y-axis, or the dependent variable axis.

For example, based on historical data, a team that practices for six days before the big game has a 70 percent probability of winning that game.

Looking at this plot, we see a clear separation between the data points.

This characteristic tells us that this data is suitable for classification using a Logistic Regression algorithm.

Decision Trees use greedy algorithms to perform recursive binary partitioning of the feature space.

The final prediction, or output, from the tree is equivalent to the label of the final leaf node reached during the decision process.

The decision made at each split represents the optimal choice at that split, while attempting to achieve an optimal global output.

However, the goal is to maximize information gain with minimal run-time.

This is what makes Decision Trees greedy.

Random Forests can be used for both classification and regression analyses.

These algorithms utilize multiple Decision Trees to predict a response.

They accomplish this by splitting the dataset into a number of partitions, and then training multiple decision trees in parallel.

Once all decision trees are trained, a common data point is passed into each tree, and a response is delivered from each.

These responses are then averaged.

By averaging in this manner, the reliability of the overall model increases, which is especially valuable when dealing with large datasets.

We also realize a decrease in variance over a single tree model, and may see a decrease in the overall training time.

K-means Clustering is an example of an unsupervised learning algorithm.

Here, entities called k-centroids are randomly placed within the dataset.

Each data point is then grouped to a certain k-centroid to form k-clusters.

Grouping is based on distance between data points and the k-centroids.

Once all data points have been classified under a k-centroid, the centroids move to new locations based on the average distance between it and the data points in its cluster.

Data points may then be re-grouped to different k-centroids, and the process continues until clustering is optimized, and the centroids no longer move.

The Gaussian Mixture Clustering model assumes that all data points are generated from a mixture of a finite number of Gaussian distributions.

We can think of Gaussian Mixture as a generalized, more complex form of K-means clustering.

From a set of unlabeled data, Gaussian Mixture Clustering uses centroids to produce Gaussian distributions.

These distributions indicate the probability that a data point belongs to a certain cluster.

In this example, we see that some data points have some probability of being purple, but they are more likely to be green.

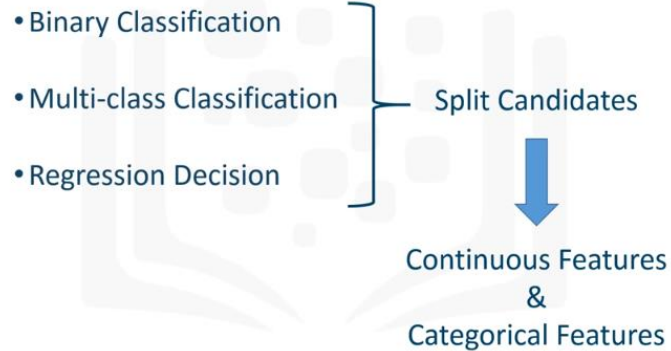
So the points may be classified as we see represented in color here.

After completing this review lesson, you should be familiar with the algorithms available in Spark MLlib, and you should identify which of those algorithms you need to review further before proceeding.

Thank you!

Decision Trees & Split Candidates

Spark Mllib Supports:



Split Candidates

Continuous Features

Small Datasets

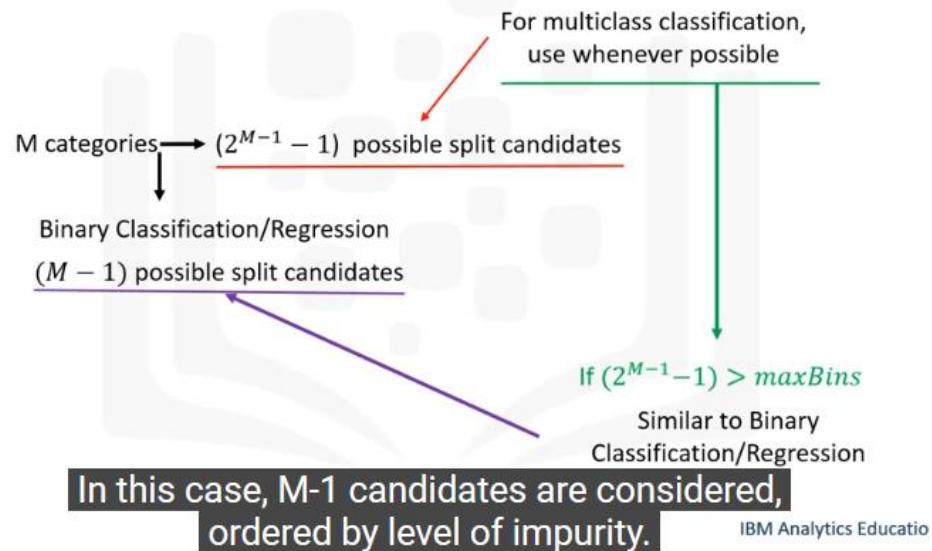
- Splits occur on unique values for the feature
- Sort feature values, then use the ordered unique values as split candidates
- Faster tree calculations

Large Datasets

- Create sets of split candidates through quantile calculations on sample of the data
- Splits create "bins". Maximum number of bins is specified with the maxBins parameter
- Maximum number of bins cannot exceed the number of instances

Split Candidates

Categorical Features



BIG DATA UNIVERSITY

Stopping Rule

Recursion stops if any of the following conditions are met.

- 1. Node depth is equal to the maxDepth training parameter
- 2. No split candidate leads to information gain greater than minInfoGain
- 3. No split candidate produces child nodes which have at least minInstancesPerNode training instances.

Hello and welcome.

My name is Deborah.

Here we will discuss Decision Trees and Random Forests supported by Spark MLlib.

This lesson will cover the features and parameters that drive Split Candidate decisions and that

determine when a recursion ends.

These are classified as Continuous Features, Categorical Features and Stopping Rules

Spark MLlib supports binary classification trees, multi-class classification trees, and regression decision trees.

Trees utilize two categories of features to develop Split Candidates in the data.

These are Continuous Features and Categorical Features.

Let's talk about Continuous Features first.

The Split Candidates for continuous features are dependent on the size of the dataset.

There are two size classifications: Small Datasets and Large Datasets.

If the dataset is small the splits usually occur based on unique values for the feature in question.

However, there is also an implementation in which the feature values are first sorted,

and then ordered.

These ordered, unique values are used as split candidates.

This allows for faster tree calculations.

For large distributed datasets, however, the task of sorting feature values becomes too costly.

Therefore, another implementation is utilized that creates sets of split candidates.

This is achieved by performing quantile calculations on a sample of the data.

In other words, the distribution of data values are partitioned, or split, into equal size groups.

When these splits occur, they create "bins", and the maximum number of bins can be specified

with the maxBins parameter.

For categorical features, the possible split candidates depend on the number of categories.

If there are M categories, then there are two to the power M minus one, minus one possible split candidates.

When dealing with a multiclass classification, then the $(2^{(M-1)}-1)$ possible splits are used whenever possible.

Note that for a binary classification, the number of split candidates is reduced to M minus one, where the categorical feature values are ordered by the average label.

For multiclass classification, there is a chance that the number of possible split candidates exceeds the maxBin value.

If this is the case, then a similar method to the binary version is used.

In this case, M-1 candidates are considered, ordered by level of impurity.

When running decision trees, there are three conditions that can cause the recursion to stop.

In the first condition, the node depth is equal to the maxDepth training parameter.

In the second, there is no split candidate that provides an information gain that is greater than minInfoGain.

The third and final condition occurs when no split candidate produces child nodes with at least the number of training instances as minInstancesPerNode.

After completing this lesson, you should be able to describe the features and parameters that drive Split Candidate decisions and that determine when a recursion ends.

In particular, you should be familiar with the classifications Continuous Features, Categorical Features and Stopping Rules Thank You!

DecisionTree Parameters

Specifiable Parameters (without Tuning required)

- numClasses, categoricalFeaturesInfo

Tunable Parameters

- maxBins, impurity

Stopping Parameters (Tunable)

- maxDepth, minInstancesPerNode, minInfoGain

Specifiable Parameters (No Tuning Required)

numClasses:

- Number of classes for classification
- Value depends on the dataset



Specifiable Parameters (No Tuning Required)

categoricalFeaturesInfo:

- Which features are categorical
- Number of values each feature can take
- Map from feature indices to number of categories.

Map(7 -> 6)

{0, 1, 2, 3, 4, 5}

Tunable Parameters

→ **Important Reminder!** ←

Avoid Overfitting!

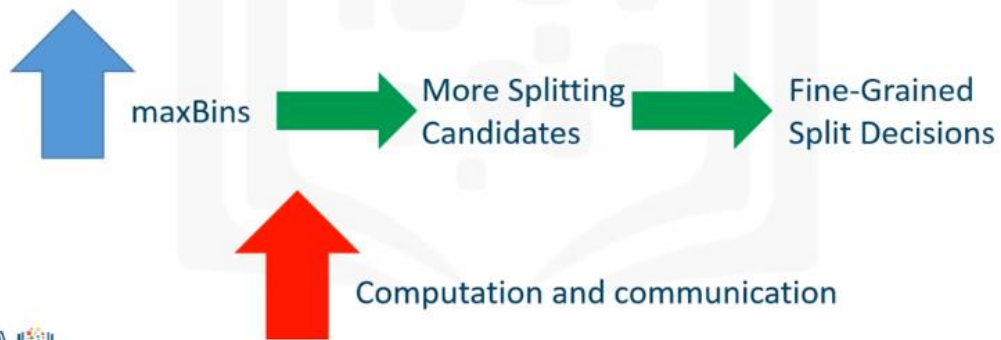
Use
Train/Test
Split!

Testing and
Training on
the same
dataset

Tunable Parameters

maxBins:

- Number of bins used
- Transforming continuous features into discrete features
- \geq max number of categories for any categorical feature



BIG DATA UNIVERSITY

Tunable Parameters

Impurity:

- Measure used to choose between candidate splits
- Used in the information gain calculation
- Must match type of DecisionTree.



BIG DATA UNIVERSITY

If a Classification Decision Tree is being used, then gini or entropy can be used.

Hello and welcome.

My name is Deborah.

Here we will discuss Decision Trees and Random Forests using Spark MLlib.

In this lesson, we will look at the different types of parameters required to create Decision Trees.

These include Specifiable Parameters that do not require tuning, and Tunable Parameters. Stopping Parameters are covered in another lesson.

We will examine how varying the value of parameters affect our tree models, either in a positive way or a negative way.

Spark MLlib supports three types of Decision Trees.

These are binary classification trees, multi-class classification trees, and regression decision trees.

Decision Trees utilize two categories of features to develop Split Candidates in the data.

These are Continuous Features and Categorical Features.

This lesson will focus on two of the three categories of input parameters used to create classification and regression Decision Tree models.

These are: Specifiable Parameters that do not require tuning, and Tunable Parameters.

Within each category are several class methods from Spark MLlib that we will explore in detail.

Stopping Parameters are covered in another lesson.

Let's first look at specifiable parameters that do not require tuning.

The first is numClasses, which is the number of classes in our model.

This value changes with the particular dataset.

For example, our dataset may contain information for several classes of dogs.

Note that there are 15 dog types in our dataset.

So for this dataset, numClasses is equal to 15.

Now consider the set of mangos shown here.

We can see that there are 30 types of mangos in this dataset.

In this case, we would have a numClasses of 30.

categoricalFeaturesInfo also does not require tuning.

It is used to specify which features are categorical, and the number of values that each feature can take.

This parameter provides a map from the feature indices to the number of categories.

In this example, our parameter tells us that feature 7 has 6 categories.

The values for the categories are 0-indexed, which means it has the values {0, 1, 2, 3, 4, 5}.

Next, let's discuss Tunable Parameters.

As a general rule, tuning these and other parameters should be done using test data that is different from the data used to train the model.

This is important to avoid overfitting.

Using Train/Test Split is much preferred to testing and training with the same dataset.

Now let's look at our first tunable parameter, maxBins.

This parameter defines the number of bins used when transforming continuous features into discrete features.

The value of maxBins must be at least the maximum number of categories for any categorical feature.

Increasing the value of maxBins allows for more possible split candidates.

Our model can therefore make finer-grained split decisions at the cost of increased computation

and communication.

The next tunable parameter is called impurity.

This parameter is a measure of the mix of data in a node and is used to choose between split candidates.

Impurity is used in the information gain calculation.

The impurity must match the type of Decision Tree model being used.

If a Classification Decision Tree is being used, then gini or entropy can be used.

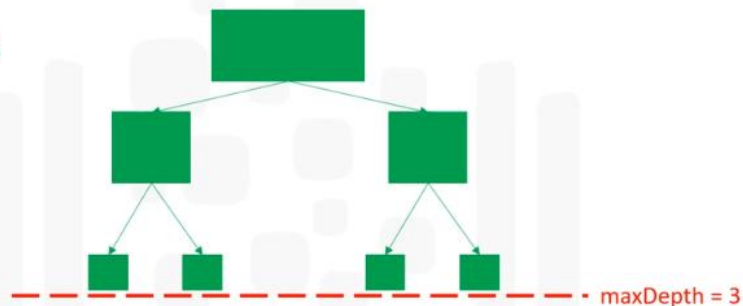
If a Regression Decision Tree is being used, then only variance is supported.

After completing this lesson, you should understand and be able to describe the different types of parameters required to create Decision Trees. These include Specifiable Parameters that do not require tuning, and Tunable Parameters. Stopping Parameters are covered in another lesson. You should be able to describe how varying the value of parameters affect our tree models, either in a positive way or a negative way. Thank You!

Stopping Parameters (Tunable Parameters)

`maxDepth` = Maximum depth of the tree

`maxDepth = 4`



- Larger Depth Yields:
- Chance of higher accuracy
 - Increased Training Cost
 - Risk of overfitting

to train and chance of overfitting.

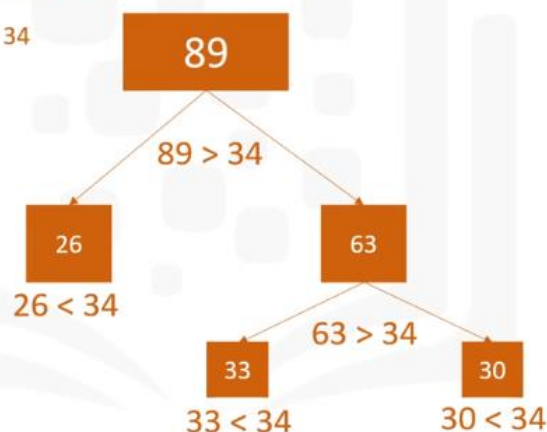
IBM Analytics Education

Stopping Parameters (Tunable Parameters)

`minInstancesPerNode`:

- Number of instances required for node to split further
- Commonly used in RandomForest

`minInstancesPerNode = 34`



so no further splits can occur for this tree.

Analytics Education

Stopping Parameters (Tunable Parameters)

minInfoGain:

- Node must provide at least this much information gain in order to split
- Information Gain: Amount of variability decrease resulting from a node split.



Variability decreases as a result of each node split.

IBM Analytics Educat

Hello and welcome.

My name is Deborah.

Here we will discuss Decision Trees and Random Forests using Spark MLlib.

In this lesson, we will look at the different types of parameters required to create Decision Trees.

Specifiable Parameters that do not require tuning, and Tunable parameters are covered in another lesson.

Here we will focus on Stopping Parameters that are tunable.

We will also examine how varying the value of parameters affect our tree models, either in a positive way or a negative way.

Spark MLlib supports three types of Decision Trees.

These are binary classification trees, multi-class classification trees, and regression decision trees.

Decision Trees utilize two categories of features to develop Split Candidates in the data.

These are Continuous Features and Categorical Features.

This lesson will focus on the one of the three categories of input parameters used to create classification and regression DecisionTree models.

This is the Stopping Parameter category.

These parameters are tunable.

Within each of the categories are several class methods from Spark MLlib that we will explore in detail.

Next, let's discuss Stopping Parameters.

These are tunable parameters.

As a general rule, tuning these and other parameters should be done using test data that is different from the data used to train the model.

This is important to avoid overfitting.

Using Train/Test Split is much preferred to testing and training with the same dataset.

The first stopping parameter we will explore is the simplest one and is called maxDepth.

As the name implies, this parameter is the maximum depth that the tree is allowed to reach.

As the tree recursively splits nodes, it will stop splitting if it reaches this defined depth.

Here is a tree with binary splits.

We have set `maxDepth` to four for this tree.

If instead we had set `maxDepth` equal to three then we would cut off the tree one layer higher, and the model would not build that last level of leaf nodes.

Note that a larger depth creates a better chance of higher accuracy, but increases cost to train and chance of overfitting.

Our next stopping parameter is `minInstancesPerNode`.

This parameter defines how many instances any node must contain before it is allowed to split further.

This is a commonly used parameter in Random Forest models.

For example, let's say that `minInstancesPerNode` is set to 34 and we have a node containing 89 instances.

Since that node has more instances than required by the `minInstancesPerNode` parameter, it can split.

Now we have one node with 26 instances and another with 63.

The node with 26 instances does not surpass `minInstancesPerNode`, so it cannot split.

However, the other node contains 63 instances.

Since this is greater than our parameter, this node can split.

Now the two nodes produced by that split both have instances less than `minInstancesPerNode`, so no further splits can occur for this tree.

The last stopping parameter is `minInfoGain`, and it defines how much information gain is necessary to allow a node to split.

Remember that information gain is the amount of variability at a node.

Variability decreases as a result of each node split.

Our previous example provides a good model for how `minInfoGain` performs.

We can imagine simply replacing number of instances with information gain.

Not enough gain?

The node can't split.

After completing this lesson, you should understand and be able to describe the different types

of parameters required to create Decision Trees.

Specifically, you should understand Stopping Parameters that are tunable.

Specifiable Parameters that do not require tuning and Tunable Parameters are covered in another lesson.

You should be able to describe how varying the value of parameters affect our tree models, either in a positive way or a negative way.

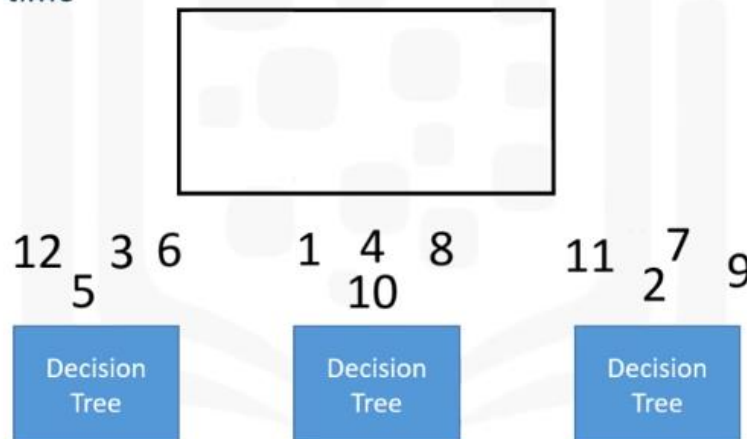
Thank You!

Parameter Comparison

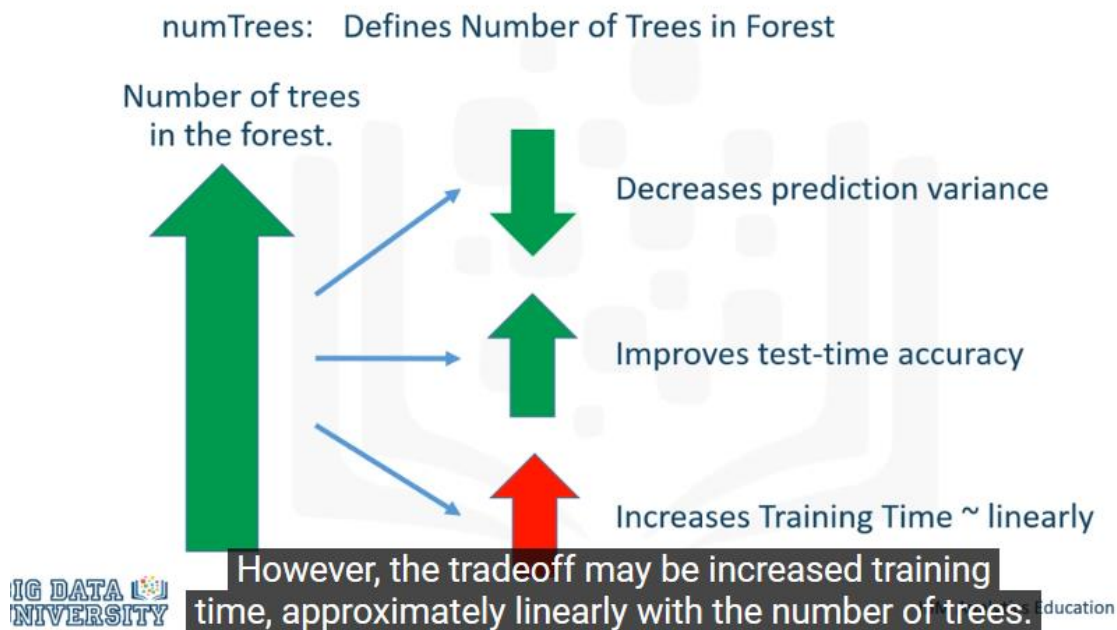
	Decision Tree Parameters	Random Forest Parameters
Specifiable Parameters (without Tuning required)	<ul style="list-style-type: none"> •numClasses •categoricalFeaturesInfo 	<ul style="list-style-type: none"> •numClasses •categoricalFeaturesInfo •seed
Tunable Parameters	<ul style="list-style-type: none"> •maxBins •Impurity 	<ul style="list-style-type: none"> •maxBins •impurity •numTrees •featureSubsetStrategy
Stopping Parameters (Tunable)	<ul style="list-style-type: none"> •maxDepth •minInstancesPerNode •minInfoGain 	<ul style="list-style-type: none"> •maxDepth

Specifiable Parameter (without Tuning)

- seed:
- Random seed for bootstrapping, choosing feature subsets.
 - Set as None (default): Generates seed based on system time



Tunable Parameters



Tunable Parameters

featureSubsetStrategy:

- Number of features used as candidates for splitting at each tree node
- Specified as a function of the total number of features
- Decreasing speeds up training, but too low can affect performance
- Supports: "auto", "all", "sqrt", "log2", "onethird"

auto: Choose automatically for task:

If numTrees == 1, set to "all."

If numTrees > 1 (forest), set to "sqrt" for classification

If numTrees > 1 (forest), set to "onethird" for regression."

all: use all features

sqrt: use $\sqrt{\# \text{ of features}}$

log2: use $\log_2(\# \text{ of features})$

Stopping Parameters (Tunable Parameters)

maxDepth: Maximum depth of each tree in the forest

- Increasing maxDepth = more powerful and expressive model
- May increase training time
- May become more prone to overfitting.
- Averaging multiple trees yields variance reduction, allowing deeper trees



is realized when the trees are averaged to get the result.

IBM Analytics Educa

Hello and welcome.

My name is Deborah.

Here we will discuss Decision Trees and Random Forests using Spark MLlib.

We will look at the different categories of parameters required to create Decision Trees and Random Forests.

These include Specifiable Parameters that do not require tuning, Tunable Parameters, and Stopping Parameters that are also tunable.

We will examine how varying the value of parameters affect our tree models, either in a positive

way or a negative way.

Finally, we will compare the parameters that are utilized in Decision Trees to those used in Random Forests.

Spark MLlib supports three types of Random Forests.

These are binary classification forests, multi-class classification forests, and regression forests.

Random Forests utilize two categories of features to develop Split Candidates in their trees.

These are Continuous Features and Categorical Features.

Like those used for Decision Trees, the parameters used to define a Random Forest model fall

into the three categories shown here.

From this table, we can see that Decision Trees and Random Forests have a large number of parameters in common.

With one exception, maxDepth, this lesson will focus on parameters that are exclusive to Random Forests, highlighted here.

Please review the lessons on Decision Trees for explanations of the parameters that are common to Decision Trees and Random Forests.

The specifiable parameter that is unique to Random Forests is the seed.

This is a random seed used for bootstrapping and choosing feature subsets.

This parameter randomizes the data so that it can be split into a subsets and fed into

the individual Decision Trees in the forest.

The default value is None, which generates a seed based on system time.

While the value of seed can change, it is still not considered a tuning parameter since changing its value has no effect on how the tree classifies data.

For example, suppose we have a list of values ranging from 1 to 12, and three Decision Trees comprising our forest.

The seed parameter would randomize these values, and then feed the distinct subsets of the data into each Decision Trees.

The first tunable parameter is numTrees.

numTrees defines how many trees will be created in the forest.

If the number of trees increases, then we see decrease in the Random Forest's sensitivity to changes in the data, or prediction variance.

This results in an improvement in the model's test-time accuracy.

However, the tradeoff may be increased training time, approximately linearly with the number of trees.

The next tunable parameter is feature Subset Strategy.

This represents the number of features used as candidates for splitting at each tree node.

This number is specified as a function of the total number of features in the dataset.

Note that decreasing this value can decrease training time, but too low of a value can affect the performance and accuracy of the Random Forest model.

Spark MLlib supports the values shown here.

Let's look closer at each of these values: "Auto" chooses the function of featureSubsetStrategy

automatically, depending on the task for the model:

For example, If numTrees is defined as 1, then the number of splitting candidates is set to the total number of features.

If numTrees is greater than 1 forest, and this is a classification model, then the number of splitting candidates is set to the square root of the total number of features.

And if numTrees is greater than 1 forest, and this is a regression model, then the number of splitting candidates is set to one third of the total number of features.

Individual values are defined as follows: "All" uses the total number of features in the dataset as the number of split candidates.

"Square Root" uses the square root of the number of features.

"Log2" uses the log base 2 of the number of features.

And "Onethird" uses one third of the number of features.

Now let's talk about the stopping parameter for Random Forests: maxDepth.

We recall this parameter from studying Decision Trees.

Remember that increasing maxDepth can yield a more powerful and expressive model.

The tradeoffs may be increased training time, and the possibility of overfitting.

However, Random Forests are more forgiving of deeper trees in that a variance reduction is realized when the trees are averaged to get the result.

After finishing this lesson, you should have a good understanding of, and be able to describe the different types of parameters required to create Decision Trees and Random Forests.

You should also be able to describe how varying the value of parameters affect our tree models,

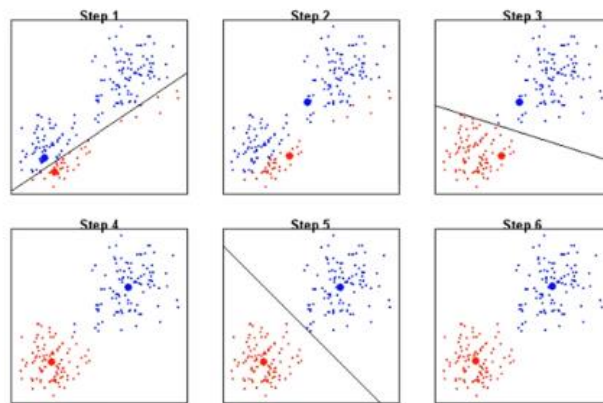
either in a positive way or a negative way.

Finally, you should be able to compare the parameters that are utilized in Decision Trees to those used in Random Forests.

Thank you!

MODULE 4 SPARK LIB CLUSTERING

K-Means Clustering



K-means++

BIG DATA UNIVERSITY

The Spark MLlib implementation of K-Means plus plus is a parallelized variant called

Analytics Educat

K-Means || Parameters

```
classmethod train(rdd, k, maxIterations=100, runs=1, initializationMode='k-means||',
                  initializationSteps=5, epsilon=0.0001, initialModel=None)
```

k: number of desired clusters

maxIterations: maximum number of iterations to run

runs: number of times to run, yielding best result out of the runs

initializationMode: specifies either random initialization or k-means || initialization

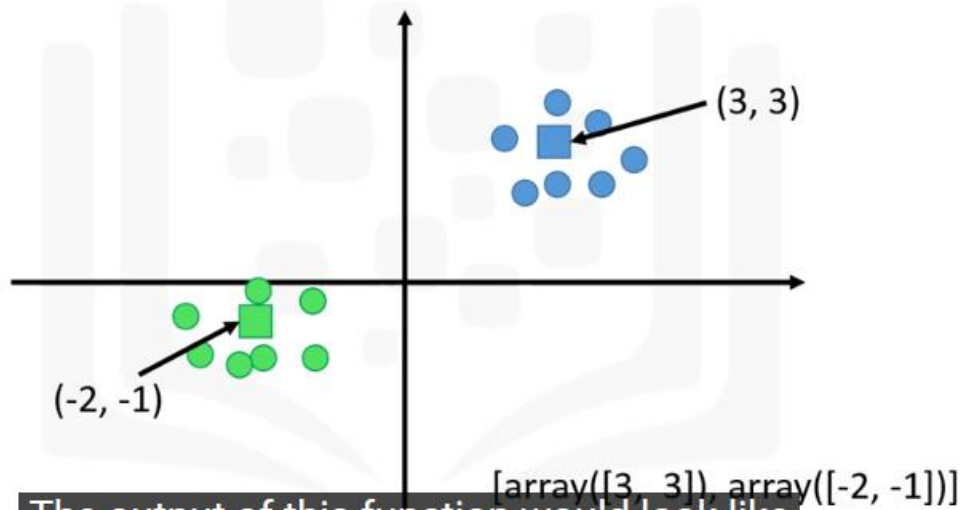
initializationSteps: determines number of steps in k-means || algorithm

epsilon: determines the distance threshold within which we consider k-means to have converged

initialModel: optional set of cluster centers used for initialization. If set to true, only one run is performed

clusterCenters

Returns cluster centers, represented as NumPy arrays.



K-Means || - Functions

computeCost(rdd)

Returns the K-means cost (sum of squared distances of points to their nearest center) for this model on the given data.

k

The total number of clusters.

load(sc, path)

Load the model to a given path.

save(sc, path)

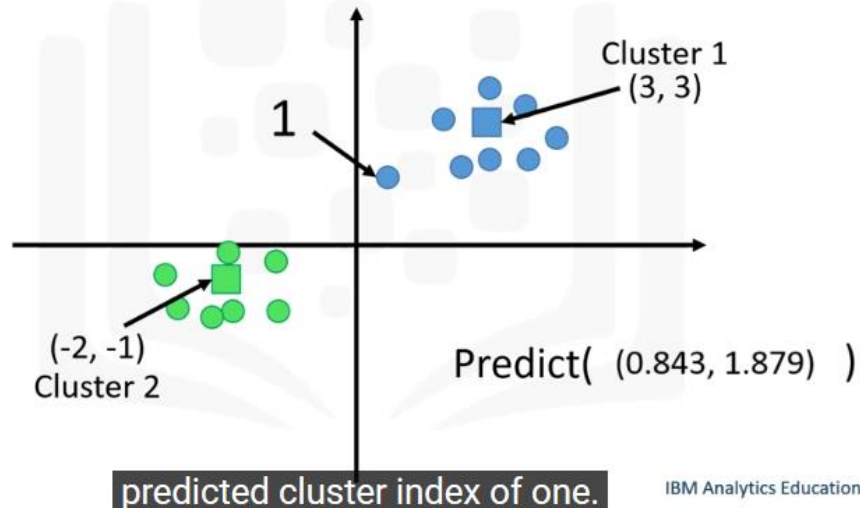
Save the model to a given path.

"Predict" is the last K-Means function we will study

K-Means | - Functions

predict(x)

Find the cluster that each of the points belongs to in this model.



BIG DATA UNIVERSITY

IBM Analytics Education

Hello and welcome.

My name is Deborah .

This lesson covers clustering algorithms supported by Spark MLlib.

We'll explore K-Means Clustering, looking first at a parallelized version implemented by Spark MLlib.

We'll examine some of the input parameters used by K-Means, and their effect on the output, including maxIterations and epsilon.

Finally, we'll look at the functions that are available for use with K-Means.

K-Means Clustering is one of the most commonly used clustering algorithms.

The Spark MLlib implementation of K-Means plus plus is a parallelized variant called K-Means parallel.

Here are some of the input parameters used to train K-Means-Parallel models.

Most of these parameters are tunable.

The first parameter is "k", which specifies the number of clusters that we wish to define.

"maxIterations" defines the maximum number of iterations that the algorithm runs before stopping.

Therefore this is also the number of times that centroids move.

The "runs" parameter determines the number of times the K-Means algorithm runs through its iterations.

It then identifies the best result from those runs.

The "initialization mode" parameter specifies whether the algorithm initiates as a "random" or as a K-Means-Parallel algorithm.

If "random" is selected, centroids are generated at random locations.

If "K-Means" is selected, the model chooses initial centroid locations to help avoid poor clustering.

initializationSteps determines the number of steps in the K-Means algorithm.

This is an advanced setting and the default value is typically used.

Epsilon is an important tuning parameter.

It represents the distance threshold that determines whether the K-Means algorithm has converged.

If the centroids move a distance less than epsilon, then the iterations stop.

Finally, the `initialModel` parameter allows the model to initialize cluster centers using a `KMeansModel` object instead of the random or `k-means-parallel` initialization models.

This is an optional parameter.

Looking closer at `maxIterations`, it seems similar to the `"runs"` parameter.

However, unlike `"runs"`, `maxIterations` is a stopping parameter.

Imagine a cluster of points with two centroids.

Setting `maxIterations` to a value of five will cause the centroids to move 5 times while clustering these data points, after which the algorithm stops.

If we set `maxIterations` to a value of 2, then only two clustering moves occur before stopping.

Like `maxIterations`, `"epsilon"` is an important stopping parameter.

Imagine another cluster of data points.

For simplicity, let's consider a single centroid.

We can imagine `epsilon` as a circle around the centroid.

If an iteration of the algorithm causes the centroid to move outside of this `epsilon` range, then the algorithm can continue.

However, if the move results in the centroid still within the range of the `epsilon` value, then algorithm stops.

Notice that if we decrease or increase the value of `epsilon`, the range for centroid moves changes accordingly.

Now let's look at what functions are available in K-Means models.

We'll start with `clusterCenters`.

This function will identify the locations of the cluster centers, represented as a list of NumPy arrays.

For example, here are two clusters of data.

Cluster centers are represented by squares.

The blue cluster center is located at position $x=3$ and $y=3$, while the green cluster center is located at negative 2 and negative 1.

The output of this function would look like this .

The next function is `computeCost(rdd)`, which returns the K-means cost, or the sum of the squared distances of each point to its nearest center.

The input (`rdd`) is a Resilient Distributed Dataset consisting of the points for which we wish to compute the cost.

Our next function is `"k"` , which simply tells us the total number of clusters.

Briefly, there are also loading and saving functions available to K-Means models.

`"Predict"` is the last K-Means function we will study.

`Predict` finds the cluster that each data point belongs to in the model.

The function examines a data point, or an RDD of multiple data points, determines which cluster that point belongs to, and then returns the predicted cluster index.

Let's say we have two clusters, and a random data point at this location.

We pass the coordinates of the data point into the `Predict` function and receive the predicted cluster index of one.

After completing this lesson, you should be able to explain:

Spark's parallelized version of K-means clusters,

The input parameters in K-means and how these parameters affect the output,

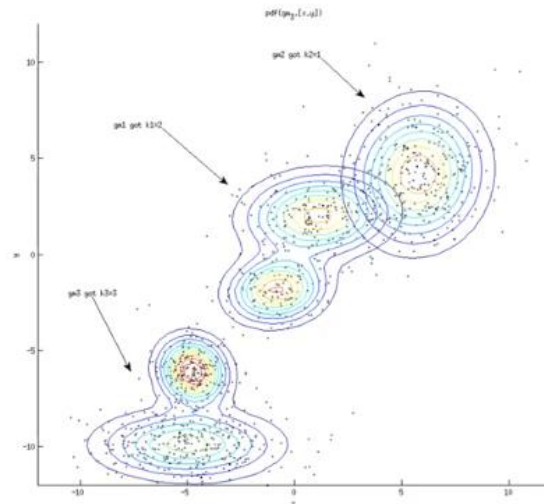
How `maxIterations` and Increasing the value of `maxBins` allows for more possible split candidates.

directly affect how a K-means algorithm stops,

And what functions are available with K-means.

Thank you!

Gaussian Mixture Clustering



BIG DATA UNIVERSITY

in which points are drawn from one of "k" Gaussian sub-distributions, each with its

Analytics Education

Gaussian Mixture - Parameters

```
classmethod train(rdd, k, convergenceTol=0.001, maxIterations=100,  
                  initialModel=None)
```

k: number of desired clusters

convergenceTol: max change in log-likelihood where convergence is achieved

maxIterations: max number of iterations to perform without reaching convergence

initialModel: optional starting point for GMM algorithm. If omitted, a random starting point is

BIG DATA UNIVERSITY

This is an optional parameter which can be used to bypass the random initialization of

Analytics Education

Gaussian Mixture - Functions

gaussians

- Returns Array of MultivariateGaussian
- `gaussians[i]` - Multivariate Gaussian Distribution for the "i" Gaussian

Mean

```
MultivariateGaussian(mu=DenseVector([-4.0109, -2.0483]),
sigma=DenseMatrix(2, 2, [1.222, -0.0085, -0.0085, 1.2463], 0))
```

IG DATA UNIVERSITY

Here, Mu represents the mean of the Gaussian, while sigma represents the standard deviation.

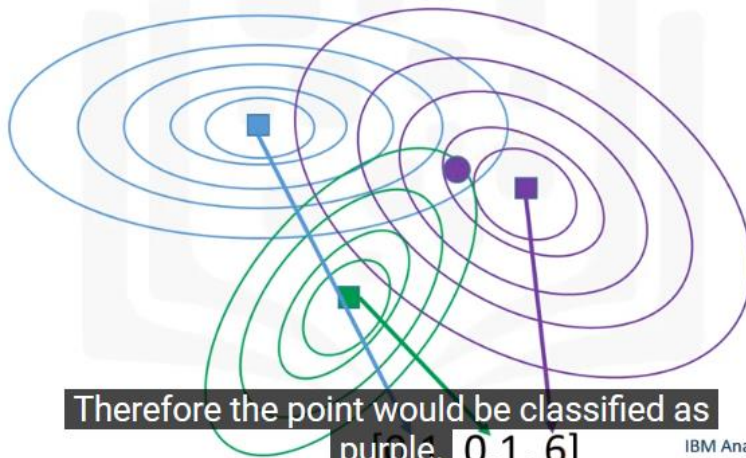
Analytics Education

6

Gaussian Mixture- Functions

predictSoft(x)

Returns the membership of point 'x' or each point in RDD 'x' to all mixture components



IG DATA UNIVERSITY

Therefore the point would be classified as purple. [0.1, 6]

IBM Analytics Education

weights

- Weights for each Gaussian distribution in the mixture
- `weights[i]` - The weight for Gaussian "i"
- `weights.sum == 1.`

[0.1543518, 0.8456482]

k

Number of gaussians in mixture.

load(sc, path)

Load the model to a given path.

save(sc, path)

Save the model to a given path.



One exception is that here the 'k' function returns the number of gaussians in the mixture.

Hello and welcome.

My name is Deborah.

This lesson covers clustering algorithms supported by Spark MLlib.

In particular, we will look at the Gaussian Mixture Clustering algorithm.

We'll explore the different parameters involved in creating these algorithms, as well as the functions associated with the Gaussian Mixture.

We remember from other lessons that a Gaussian Mixture represents a composite distribution

in which points are drawn from one of "k" Gaussian sub-distributions, each with its own probability.

Looking at this diagram, the curves that we see are Gaussian distributions.

In another lesson, we saw the parameters needed to form a K-Means cluster.

Here now is the training class of parameters for Gaussian Mixture Clustering.

The first parameter we consider is k.

K is the number of desired clusters, or how many clusters we wish to create.

Next is convergenceTol, which is the maximum change in log-likelihood for which we consider

convergence to have been achieved.

This parameter also acts as a stopping parameter that can be tuned.

In a way, convergenceTol is similar to epsilon in k-means clustering.

Now let's look at maxIterations.

This parameter represents the maximum number of iterations that can occur, even if convergence isn't reached.

This is the same parameter as the one we visited in a lesson on K-means clustering.

The final parameter is initialModel.

This is an optional parameter which can be used to bypass the random initialization of the data.

Now let's look at the functions available for Gaussian Mixture Clustering, starting with gaussians.

This function returns an array of MultivariateGaussian.

Here, gaussians[i] represents the multivariate gaussian, or normal, distribution for the

Gaussian identified as "i".

The output of a single Gaussian might look like this.

Here, μ represents the mean of the Gaussian, while σ represents the standard deviation.

The next function is `predictSoft`, which finds and returns the membership of a point 'x', or each point in an RDD, to all mixture components.

Suppose we have 3 gaussians as shown here.

Now suppose we introduce a data point that we wish to classify.

What is the probability that the point belongs to the blue, the green, or the purple cluster? `predictSoft` will provide an array of numbers that determines the membership of the point to each cluster.

In this example, we see that the highest membership value is in purple.

Therefore the point would be classified as purple.

Now let's look at the "weights"

function which, as the name implies, determines the weight of each Gaussian distribution in the mixture.

Note that `weights[i]` represents the weight for the 'i' Gaussian, and the sum of all Gaussian weight values, or `weights.sum`, must equal one.

The 'weights' function outputs an array in which each value in the array represents a Gaussian distribution weight.

The larger the value, the larger the distribution.

The last three functions are virtually the same as those that we explored in our lessons on K-Means Clustering.

One exception is that here the 'k' function returns the number of gaussians in the mixture.

This is analogous to the number of cluster centers that 'k' returned in K-means.

After completing this lesson you should understand and be able to explain the different parameters

involved in, and the functions associated with Gaussian Mixture Clusters.

Thank you!