CHAPTER 8

# Using Graph Algorithms to Enhance Machine Learning

We've covered several algorithms that learn and update state at each iteration, such as Label Propagation; however, up until this point, we've emphasized graph algorithms for general analytics. Because there's increasing application of graphs in machine learning (ML), we'll now look at how graph algorithms can be used to enhance ML workflows.

In this chapter, we focus on the most practical way to start improving ML predictions using graph algorithms: connected feature extraction and its use in predicting relationships. First, we'll cover some basic ML concepts and the importance of contextual data for better predictions. Then there's a quick survey of ways graph features are applied, including uses for spammer fraud, detection, and link prediction.

We'll demonstrate how to create a machine learning pipeline and then train and evaluate a model for link prediction, integrating Neo4j and Spark in our workflow. Our example will be based on the Citation Network Dataset, which contains authors, papers, author relationships, and citation relationships. We'll use several models to predict whether research authors are likely to collaborate in the future, and show how graph algorithms improve the results.

## Machine Learning and the Importance of Context

Machine learning is not artificial intelligence (AI), but a method for achieving AI. ML uses algorithms to train software through specific examples and progressive improvements based on expected outcome—without explicit programming of how to accomplish these better results. Training involves providing a lot of data to a model and enabling it to learn how to process and incorporate that information.

In this sense, learning means that algorithms iterate, continually making changes to get closer to an objective goal, such as reducing classification errors in comparison to the training data. ML is also dynamic, with the ability to modify and optimize itself when presented with more data. This can take place in pre-usage training on many batches or as online learning during usage.

Recent successes in ML predictions, accessibility of large datasets, and parallel compute power have made ML more practical for those developing probabilistic models for AI applications. As machine learning becomes more widespread, it's important to remember its fundamental goal: making choices similarly to the way humans do. If we forget that goal, we may end up with just another version of highly targeted, rules-based software.

In order to increase machine learning accuracy while also making solutions more broadly applicable, we need to incorporate a lot of contextual information—just as people should use context for better decisions. Humans use their surrounding context, not just direct data points, to figure out what's essential in a situation, estimate missing information, and determine how to apply lessons to new situations. Context helps us improve predictions.

## Graphs, Context, and Accuracy

Without peripheral and related information, solutions that attempt to predict behavior or make recommendations for varying circumstances require more exhaustive training and prescriptive rules. This is partly why AI is good at specific, well-defined tasks, but struggles with ambiguity. Graph-enhanced ML can help fill in that missing contextual information that is so important for better decisions.

We know from graph theory and from real life that relationships are often the strongest predictors of behavior. For example, if one person votes, there's an increased likelihood that their friends, family, and even coworkers will vote. Figure 8-1 illustrates a ripple effect based on reported voting and Facebook friends from the 2012 research paper "A 61-Million-Person Experiment in Social Influence and Political Mobilization", by R. Bond et al.

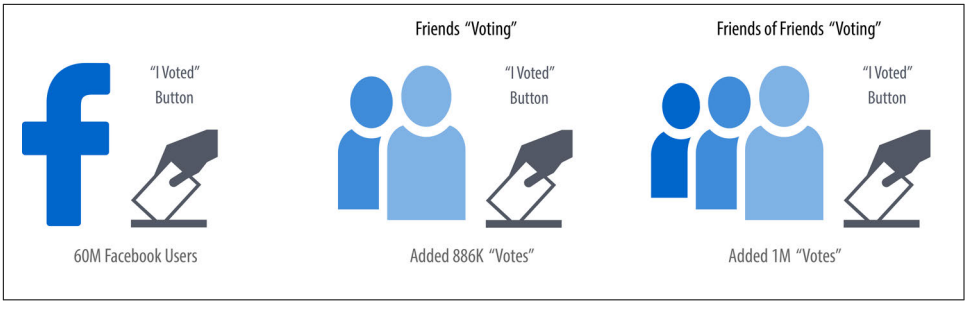


*Figure 8-1. People are influenced to vote by their social networks. In this example, friends two hops away had more total impact than direct relationships.*

The authors found that friends reporting voting influenced an additional 1.4% of users to also claim they'd voted and, interestingly, friends of friends added another 1.7%. Small percentages can have a significant impact, and we can see in Figure 8-1 that people at two hops out had in total more impact than the direct friends alone. Voting and other examples of how our social networks impact us are covered in the book *Connected*, by Nicholas Christakis and James Fowler (Little, Brown and Company).

Adding graph features and context improves predictions, especially in situations where connections matter. For example, retail companies personalize product recommendations with not only historical data but also contextual data about customer similarities and online behavior. Amazon's Alexa uses several layers of contextual models that demonstrate improved accuracy. In 2018, Amazon also introduced "context carryover" to incorporate previous references in a conversation when answering new questions.

Unfortunately, many machine learning approaches today miss a lot of rich contextual information. This stems from ML's reliance on input data built from tuples, leaving out a lot of predictive relationships and network data. Furthermore, contextual information is not always readily available or is too difficult to access and process. Even finding connections that are four or more hops away can be a challenge at scale for traditional methods. Using graphs, we can more easily reach and incorporate connected data.

## Connected Feature Extraction and Selection

Feature extraction and selection helps us take raw data and create a suitable subset and format for training our machine learning models. It's a foundational step that,

when well executed, leads to ML that produces more consistently accurate predictions.

### Feature Extraction and Selection

*Feature extraction* is a way to distill large volumes of data and attributes down to a set of representative descriptive attributes. The process derives numerical values (features) for distinctive characteristics or patterns in input data so that we can differentiate categories in other data. It's used when data is difficult for a model to analyze directly—perhaps because of size, format, or the need for incidental comparisons.

*Feature selection* is the process of determining the subset of extracted features that are most important or influential to a target goal. It's used to surface predictive importance as well as for efficiency. For example, if we have 20 features and 13 of them together explain 92% of what we need to predict, we can eliminate 7 features in our model.

Putting together the right mix of features can increase accuracy because it fundamentally influences how our models learn. Because even modest improvements can make a significant difference, our focus in this chapter is on *connected features*. Connected features are features extracted from the structure of the data. These features can be derived from graph-local queries based on parts of the graph surrounding a node, or graph-global queries that use graph algorithms to identify predictive elements within data based on relationships for connected feature extraction.

And it's not only important to get the right combination of features, but also to eliminate unnecessary features to reduce the likelihood that our models will be hypertargeted. This keeps us from creating models that only work well on our training data (known as *overfitting*) and significantly expands applicability. We can also use graph algorithms to evaluate those features and determine which ones are most influential to our model for connected feature selection. For example, we can map features to nodes in a graph, create relationships based on similar features, and then compute the centrality of features. Feature relationships can be defined by the ability to preserve cluster densities of data points. This method is described using datasets with high dimension and low sample size in "Unsupervised Graph-Based Feature Selection Via Subspace and PageRank Centrality", by K. Henniab, N. Mezghani, and C. Gouin-Vallerand.

### Graph Embedding

*Graph embedding* is the representation of the nodes and relationships in a graph as *feature vectors*. These are merely collections of features that have dimensional mappings, such as the $(x,y,z)$ coordinates shown in Figure 8-2.

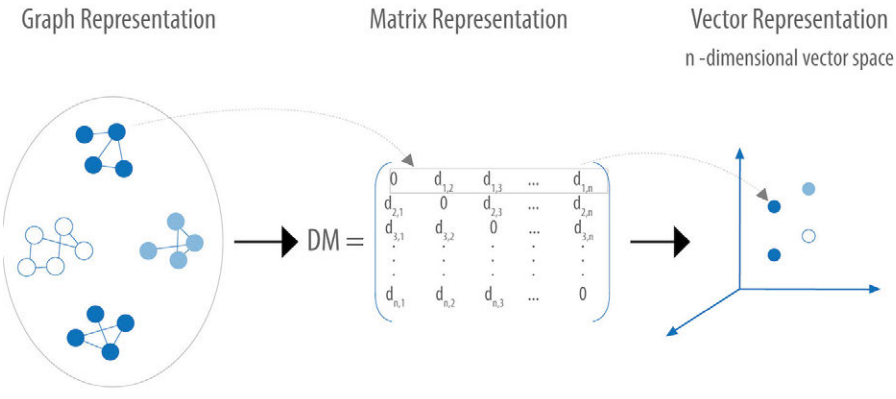


*Figure 8-2. Graph embedding maps graph data into feature vectors that can be visualized as multidimensional coordinates.*

Graph embedding uses graph data slightly differently than in connected feature extraction. It enables us to represent entire graphs, or subsets of graph data, in a numerical format ready for machine learning tasks. This is especially useful for unsupervised learning, where the data is not categorized because it pulls in more contextual information through relationships. Graph embedding is also useful for data exploration, computing similarity between entities, and reducing dimensionality to aid in statistical analysis.

This is a quickly evolving space with several options, including node2vec, struc2vec, GraphSAGE, DeepWalk, and DeepGL.

Now let's look at some of the types of connected features and how they are used.

## Graphy Features

*Graphy features* include any number of connection-related metrics about our graph, such as the number of relationships going into or out of nodes, a count of potential triangles, and neighbors in common. In our example, we'll start with these measures because they are simple to gather and a good test of early hypotheses.

In addition, when we know precisely what we're looking for, we can use feature engineering. For instance, if we want to know how many people have a fraudulent account at up to four hops out. This approach uses graph traversal to very efficiently find deep paths of relationships, looking at things such as labels, attributes, counts, and inferred relationships.

We can also easily automate these processes and deliver those predictive graphy features into our existing pipeline. For example, we could abstract a count of fraudster

relationships and add that number as a node attribute to be used for other machine learning tasks.

## Graph Algorithm Features

We can also use graph algorithms to find features where we know the general structure we're looking for but not the exact pattern. As an illustration, let's say we know certain types of community groupings are indicative of fraud; perhaps there's a prototypical density or hierarchy of relationships. In this case, we don't want a rigid feature of an exact organization but rather a flexible and globally relevant structure. We'll use community detection algorithms to extract connected features in our example, but centrality algorithms, like PageRank, are also frequently applied.

Furthermore, approaches that combine several types of connected features seem to outperform sticking to one single method. For example, we could combine connected features to predict fraud with indicators based on communities found via the Louvain algorithm, influential nodes using PageRank, and the measure of known fraudsters at three hops out.

A combined approach is demonstrated in Figure 8-3, where the authors combine graph algorithms like PageRank and Coloring with graphy measure such as in-degree and out-degree. This diagram is taken from the paper "Collective Spammer Detection in Evolving Multi-Relational Social Networks", by S. Fakhraei et al.

The Graph Structure section illustrates connected feature extraction using several graph algorithms. Interestingly, the authors found extracting connected features from multiple types of relationships even more predictive than simply adding more features. The Report Subgraph section shows how graph features are converted into features that the ML model can use. By combining multiple methods in a graph-enhanced ML workflow, the authors were able to improve prior detection methods and classify 70% of spammers that had previously required manual labeling, with 90% accuracy.

Even once we have extracted connected features, we can improve our training by using graph algorithms like PageRank to prioritize the features with the most influence. This enables us to adequately represent our data while eliminating noisy variables that could degrade results or slow processing. With this type of information, we can also identify features with high co-occurrence for further model tuning via feature reduction. This method is outlined in the research paper "Using PageRank in Feature Selection", by D. Ienco, R. Meo, and M. Botta.

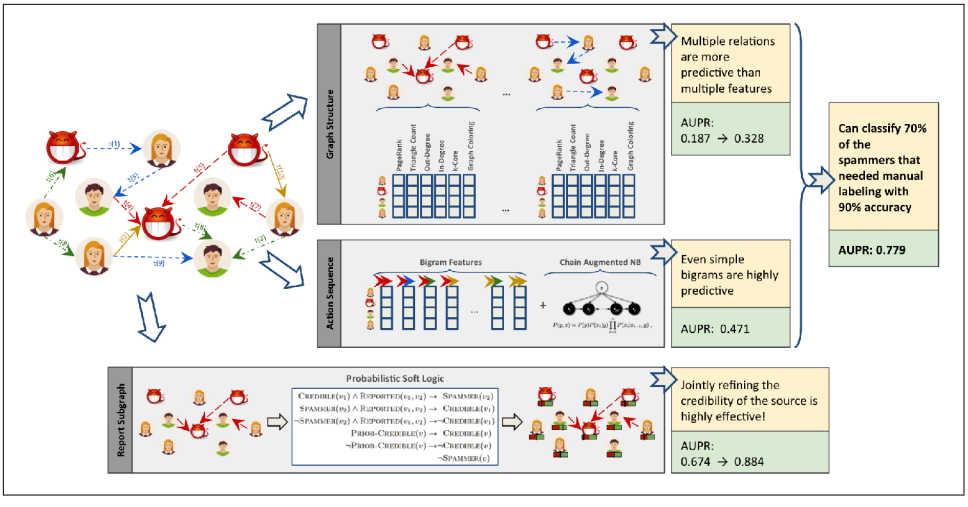


*Figure 8-3. Connected feature extraction can be combined with other predictive methods to improve results. AUPR refers to the area under the precision-recall curve, with higher numbers preferred.*

We've discussed how connected features are applied to scenarios involving fraud and spammer detection. In these situations, activities are often hidden in multiple layers of obfuscation and network relationships. Traditional feature extraction and selection methods may be unable to detect that behavior without the contextual information that graphs bring.

Another area where connected features enhance machine learning (and the focus of the rest of this chapter) is *link prediction*. Link prediction is a way to estimate how likely a relationship is to form in the future, or whether it should already be in our graph but is missing due to incomplete data. Since networks are dynamic and can grow fairly quickly, being able to predict links that will soon be added has broad applicability, from product recommendations to drug retargeting and even inferring criminal relationships.

Connected features from graphs are often used to improve link prediction using basic graphy features as well as features extracted from centrality and community algorithms. Link prediction based on node proximity or similarity is also standard; in the paper "The Link Prediction Problem for Social Networks" D. Liben-Nowell and J. Kleinberg suggest that the network structure alone may contain enough latent information to detect node proximity and outperform more direct measures.

Now that we've looked at ways connected features can enhance machine learning, let's dive into our link prediction example and see how we can apply graph algorithms to improve our predictions.

# Graphs and Machine Learning in Practice: Link Prediction

The rest of the chapter will demonstrate a hands-on example, based on the Citation Network Dataset, a research dataset extracted from DBLP, ACM, and MAG. The dataset is described in the paper "ArnetMiner: Extraction and Mining of Academic Social Networks", by J. Tang et al. The latest version contains 3,079,007 papers, 1,766,547 authors, 9,437,718 author relationships, and 25,166,994 citation relationships.

We'll be working with a subset focused on articles that appeared in the following publications:

- *Lecture Notes in Computer Science*
- *Communications of the ACM*
- *International Conference on Software Engineering*
- *Advances in Computing and Communications*

Our resulting dataset contains 51,956 papers, 80,299 authors, 140,575 author relationships, and 28,706 citation relationships. We'll create a coauthors graph based on authors who have collaborated on papers and then predict future collaborations between pairs of authors. We're only interested in collaborations between authors who haven't collaborated before—we're not concerned with multiple collaborations between pairs of authors.

In the remainder of the chapter, we'll set up the required tools and import the data into Neo4j. Then we'll cover how to properly balance data and split samples into Spark DataFrames for training and testing. After that, we explain our hypothesis and methods for link prediction before creating a machine learning pipeline in Spark. Finally, we'll walk through training and evaluating various prediction models, starting with basic graphy features and adding more graph algorithm features extracted using Neo4j.

## Tools and Data

Let's get started by setting up our tools and data. Then we'll explore our dataset and create a machine learning pipeline.

Before we do anything else, let's set up the libraries used in this chapter:

*py2neo*
: A Neo4j Python library that integrates well with the Python data science ecosystem

*pandas*
: A high-performance library for data wrangling outside of a database with easy-to-use data structures and data analysis tools

*Spark MLlib*
: Spark's machine learning library

We use MLlib as an example of a machine learning library. The approach shown in this chapter could be used in combination with other ML libraries, such as scikit-learn.

All the code shown will be run within the pyspark REPL. We can launch the REPL by running the following command:

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
  --driver-memory 2g \
  --executor-memory 6g \
  --packages julioasotodv:spark-tree-plotting:0.2
```

This is similar to the command we used to launch the REPL in Chapter 3, but instead of GraphFrames, we're loading the `spark-tree-plotting` package. At the time of writing the latest released version of Spark is *spark-2.4.0-bin-hadoop2.7*, but as that may have changed by the time you read this, be sure to change the `SPARK_VERSION` environment variable appropriately.

Once we've launched that we'll import the following libraries that we'll be using:

```
from py2neo import Graph
import pandas as pd
from numpy.random import randint

from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.sql.types import *
from pyspark.sql import functions as F

from sklearn.metrics import roc_curve, auc
from collections import Counter

from cycler import cycler
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

And now let's create a connection to our Neo4j database:

```
graph = Graph("bolt://localhost:7687", auth=("neo4j", "neo"))
```

## Importing the Data into Neo4j

Now we're ready to load the data into Neo4j and create a balanced split for our training and testing. We need to download the ZIP file of Version 10 of the dataset, unzip it, and place the contents in our *import* folder. We should have the following files:

- *dblp-ref-0.json*
- *dblp-ref-1.json*
- *dblp-ref-2.json*
- *dblp-ref-3.json*

Once we have those files in the *import* folder, we need to add the following property to our Neo4j settings file so that we can process them using the APOC library:

```
apoc.import.file.enabled=true
apoc.import.file.use_neo4j_config=true
```

First we'll create constraints to ensure we don't create duplicate articles or authors:

```
CREATE CONSTRAINT ON (article:Article)
ASSERT article.index IS UNIQUE;

CREATE CONSTRAINT ON (author:Author)
ASSERT author.name IS UNIQUE;
```

Now we can run the following query to import the data from the JSON files:

```
CALL apoc.periodic.iterate(
  'UNWIND ["dblp-ref-0.json","dblp-ref-1.json",
           "dblp-ref-2.json","dblp-ref-3.json"] AS file
   CALL apoc.load.json("file:///" + file)
   YIELD value
   WHERE value.venue IN ["Lecture Notes in Computer Science",
                         "Communications of The ACM",
                         "international conference on software engineering",
                         "advances in computing and communications"]
   return value',
  'MERGE (a:Article {index:value.id})
   ON CREATE SET a += apoc.map.clean(value,["id","authors","references"],[0])
   WITH a,value.authors as authors
   UNWIND authors as author
   MERGE (b:Author{name:author})
   MERGE (b)<-[:AUTHOR]-(a)'
, {batchSize: 10000, iterateList: true});
```

This results in the graph schema seen in Figure 8-4.

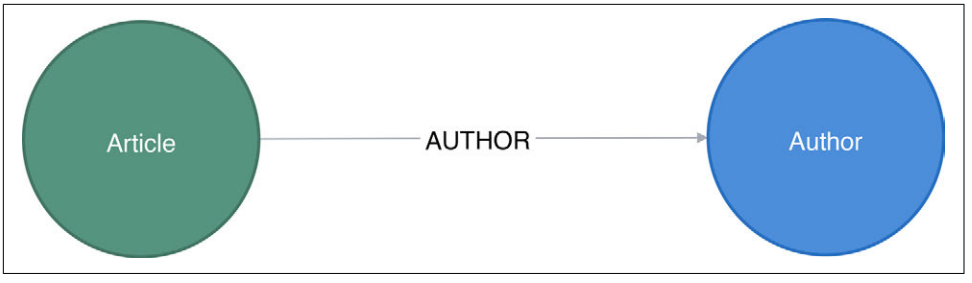


*Figure 8-4. The citation graph*

This is a simple graph that connects articles and authors, so we'll add more information we can infer from relationships to help with predictions.

## The Coauthorship Graph

We want to predict future collaborations between authors, so we'll start by creating a coauthorship graph. The following Neo4j Cypher query will create a `CO_AUTHOR` relationship between every pair of authors that have collaborated on a paper:

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
MERGE (a1)-[coauthor:CO_AUTHOR {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

The `year` property that we set on the `CO_AUTHOR` relationship in the query is the earliest year when those two authors collaborated. We're only interested in the first time that a pair of authors have collaborated—subsequent collaborations aren't relevant.

Figure 8-5 is in an example of part of the graph that gets created. We can already see some interesting community structures.

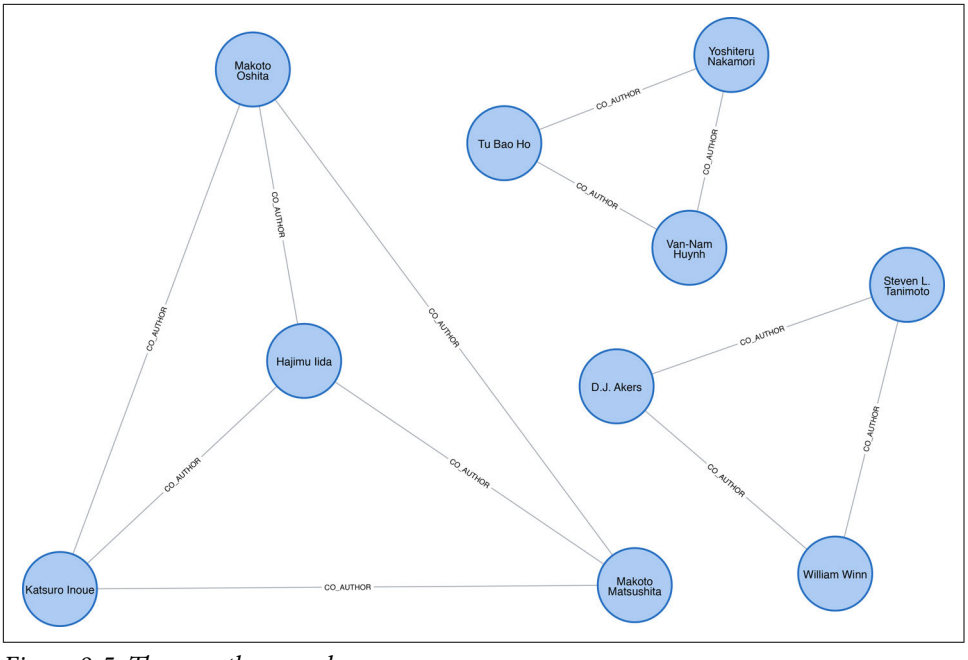


*Figure 8-5. The coauthor graph*

Each circle in this diagram represents one author and the lines between them are `CO_AUTHOR` relationships, so we have four authors that have all collaborated with each other on the left, and then on the right two examples of three authors who have collaborated. Now that we have our data loaded and a basic graph, let's create the two datasets we'll need for training and testing.

## Creating Balanced Training and Testing Datasets

With link prediction problems we want to try and predict the future creation of links. This dataset works well for that because we have dates on the articles that we can use to split our data. We need to work out which year we'll use to define our training/test split. We'll train our model on everything before that year and then test it on the links created after that date.

Let's start by finding out when the articles were published. We can write the following query to get a count of the number of articles, grouped by year:

```
query = """
MATCH (article:Article)
RETURN article.year AS year, count(*) AS count
ORDER BY year
"""

by_year = graph.run(query).to_data_frame()
```

Let's visualize this as a bar chart, with the following code:

```
plt.style.use('fivethirtyeight')
ax = by_year.plot(kind='bar', x='year', y='count', legend=None, figsize=(15,8))
ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

We can see the chart generated by executing this code in Figure 8-6.

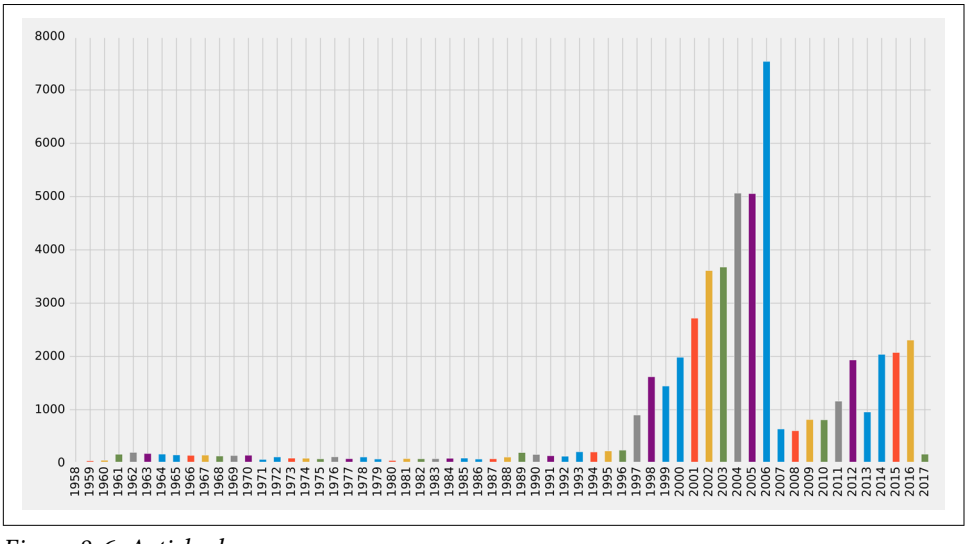


*Figure 8-6. Articles by year*

Very few articles were published before 1997, and then there were a lot published between 2001 and 2006, before a dip and then a gradual climb since 2011 (excluding 2013). It looks like 2006 could be a good year to split our data for training our model and making predictions. Let's check how many papers were published before that year and how many during and after. We can write the following query to compute this:

```
MATCH (article:Article)
RETURN article.year < 2006 AS training, count(*) AS count
```

The result of this is as follows, where *true* means a paper was published before 2006:

| training | count |
|---|---|
| false | 21059 |
| true | 30897 |

Not bad! 60% of the papers were published before 2006 and 40% during or after 2006. This is a fairly balanced split of data for our training and testing.

So now that we have a good split of papers, let's use the same 2006 split for coauthorship. We'll create a `CO_AUTHOR_EARLY` relationship between pairs of authors whose first collaboration was *before 2006*:

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year < 2006
MERGE (a1)-[coauthor:CO_AUTHOR_EARLY {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

And then we'll create a `CO_AUTHOR_LATE` relationship between pairs of authors whose first collaboration was *during or after 2006*:

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year >= 2006
MERGE (a1)-[coauthor:CO_AUTHOR_LATE {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

Before we build our training and test sets, let's check how many pairs of nodes we have that have links between them. The following query will find the number of `CO_AUTHOR_EARLY` pairs:

```
MATCH ()-[:CO_AUTHOR_EARLY]->()
RETURN count(*) AS count
```

Running that query will return the result shown here:

| count |
| --- |
| 81096 |

And this query will find the number of `CO_AUTHOR_LATE` pairs:

```
MATCH ()-[:CO_AUTHOR_LATE]->()
RETURN count(*) AS count
```

Running that query returns this result:

| count |
| --- |
| 74128 |

Now we're ready to build our training and test datasets.

### Balancing and splitting data

The pairs of nodes with `CO_AUTHOR_EARLY` and `CO_AUTHOR_LATE` relationships between them will act as our positive examples, but we'll also need to create some negative examples. Most real-world networks are sparse, with concentrations of relationships, and this graph is no different. The number of examples where two nodes do not have a relationship is much larger than the number that do have a relationship.

If we query our `CO_AUTHOR_EARLY` data, we'll find there are 45,018 authors with that type of relationship but only 81,096 relationships between authors. That might not sound imbalanced, but it is: the potential maximum number of relationships that our graph could have is (45018 * 45017) / 2 = 1,013,287,653, which means there are a lot of negative examples (no links). If we used all the negative examples to train our model, we'd have a severe class imbalance problem. A model could achieve extremely high accuracy by predicting that every pair of nodes doesn't have a relationship.

In their paper "New Perspectives and Methods in Link Prediction", R. Lichtenwalter, J. Lussier, and N. Chawla describe several methods to address this challenge. One of these approaches is to build negative examples by finding nodes within our neighborhood that we aren't currently connected to.

We will build our negative examples by finding pairs of nodes that are a mix of between two and three hops away from each other, excluding those pairs that already have a relationship. We'll then downsample those pairs of nodes so that we have an equal number of positive and negative examples.

We have 314,248 pairs of nodes that don't have a relationship between each other at a distance of two hops. If we increase the distance to three hops, we have 967,677 pairs of nodes.

The following function will be used to downsample the negative examples:

```
def down_sample(df):
    copy = df.copy()
    zero = Counter(copy.label.values)[0]
    un = Counter(copy.label.values)[1]
    n = zero - un
    copy = copy.drop(copy[copy.label == 0].sample(n=n, random_state=1).index)
    return copy.sample(frac=1)
```

This function works out the difference between the number of positive and negative examples, and then samples the negative examples so that there are equal numbers. We can then run the following code to build a training set with balanced positive and negative examples:

```
train_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_EARLY]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

train_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_EARLY]-()
MATCH (author)-[:CO_AUTHOR_EARLY*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR_EARLY]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

train_missing_links = train_missing_links.drop_duplicates()
training_df = train_missing_links.append(train_existing_links, ignore_index=True)
training_df['label'] = training_df['label'].astype('category')
training_df = down_sample(training_df)
training_data = spark.createDataFrame(training_df)
```

We've now coerced the `label` column to be a category, where `1` indicates that there is a link between a pair of nodes, and `0` indicates that there is not a link. We can look at the data in our DataFrame by running the following code:

```
training_data.show(n=5)
```

| node1 | node2 | label |
|---|---|---|
| 10019 | 28091 | 1 |
| 10170 | 51476 | 1 |
| 10259 | 17140 | 0 |
| 10259 | 26047 | 1 |
| 10293 | 71349 | 1 |

The results show us a list of node pairs and whether they have a coauthor relationship; for example, nodes `10019` and `28091` have a `1` label, indicating a collaboration.

Now let's execute the following code to check the summary of contents for the DataFrame:

```
training_data.groupby("label").count().show()
```

Here's the result:

| label | count |
|---|---|
| 0 | 81096 |
| 1 | 81096 |

We've created our training set with the same number of positive and negative samples. Now we need to do the same for the test set. The following code will build a test set with balanced positive and negative examples:

```
test_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_LATE]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

test_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_LATE]-()
MATCH (author)-[:CO_AUTHOR*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

test_missing_links = test_missing_links.drop_duplicates()
test_df = test_missing_links.append(test_existing_links, ignore_index=True)
test_df['label'] = test_df['label'].astype('category')
test_df = down_sample(test_df)
test_data = spark.createDataFrame(test_df)
```

We can execute the following code to check the contents of the DataFrame:

```
test_data.groupby("label").count().show()
```

Which gives the following result:

| label | count |
|---|---|
| 0 | 74128 |
| 1 | 74128 |

Now that we have balanced training and test datasets, let's look at our methods for predicting links.

## How We Predict Missing Links

We need to start with some basic assumptions about what elements in our data might predict whether two authors will become coauthors at a later date. Our hypothesis would vary by domain and problem, but in this case, we believe the most predictive features will be related to communities. We'll begin with the assumption that the following elements increase the probability that authors become coauthors:

- More coauthors in common
- Potential triadic relationships between authors
- Authors with more relationships

- Authors in the same community
- Authors in the same, tighter community

We'll build graph features based on our assumptions and use those to train a binary classifier. *Binary classification* is a type of ML with the task of predicting which of two predefined groups an element belongs to based on a rule. We're using the classifier for the task of predicting whether a pair of authors will have a link or not, based on a classification rule. For our examples, a value of `1` means there is a link (coauthorship), and a value of `0` means there isn't a link (no coauthorship).

We'll implement our binary classifier as a random forest in Spark. A *random forest* is an ensemble learning method for classification, regression, and other tasks, as illustrated in Figure 8-7.

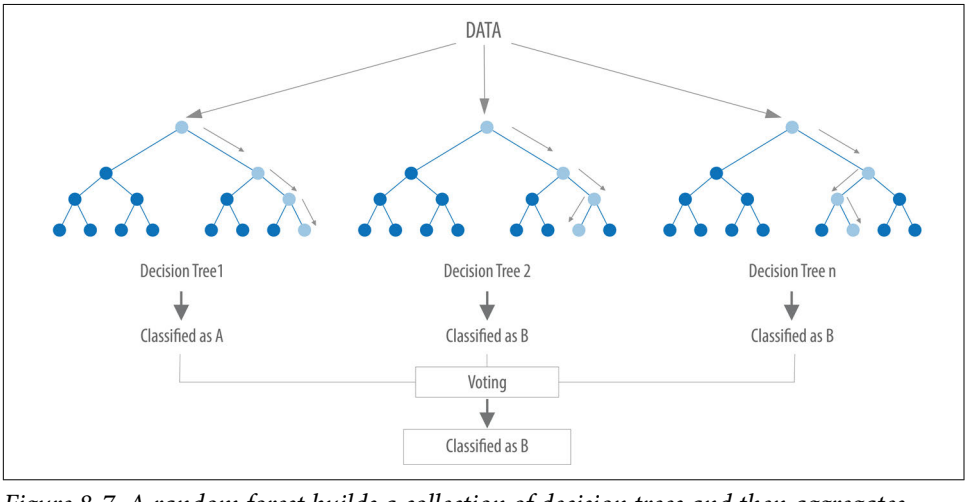


*Figure 8-7. A random forest builds a collection of decision trees and then aggregates results for a majority vote (for classification) or an average value (for regression).*

Our random forest classifier will take the results from the multiple decision trees we train and then use voting to predict a classification—in our example, whether there is a link (coauthorship) or not.

Now let's create our workflow.

## Creating a Machine Learning Pipeline

We'll create our machine learning pipeline based on a random forest classifier in Spark. This method is well suited as our dataset will be comprised of a mix of strong and weak features. While the weak features will sometimes be helpful, the random forest method will ensure we don't create a model that only fits our training data.

To create our ML pipeline, we'll pass in a list of features as the `fields` variable—these are the features that our classifier will use. The classifier expects to receive those features as a single column called `features`, so we use the `VectorAssembler` to transform the data into the required format.

The following code creates a machine learning pipeline and sets up our parameters using MLlib:

```
    def create_pipeline(fields):
        assembler = VectorAssembler(inputCols=fields, outputCol="features")
        rf = RandomForestClassifier(labelCol="label", featuresCol="features",
                                    numTrees=30, maxDepth=10)
        return Pipeline(stages=[assembler, rf])
```

The `RandomForestClassifier` uses these parameters:

`labelCol`
: The name of the field containing the variable we want to predict; i.e., whether a pair of nodes have a link

`featuresCol`
: The name of the field containing the variables that will be used to predict whether a pair of nodes have a link

`numTrees`
: The number of decision trees that form the random forest

`maxDepth`
: The maximum depth of the decision trees

We chose the number of decision trees and their depth based on experimentation. We can think about hyperparameters like the settings of an algorithm that can be adjusted to optimize performance. The best hyperparameters are often difficult to determine ahead of time, and tuning a model usually requires some trial and error.

We've covered the basics and set up our pipeline, so let's dive into creating our model and evaluating how well it performs.

## Predicting Links: Basic Graph Features

We'll start by creating a simple model that tries to predict whether two authors will have a future collaboration based on features extracted from common authors, preferential attachment, and the total union of neighbors:

*Common authors*
: Finds the number of potential triangles between two authors. This captures the idea that two authors who have coauthors in common may be introduced and collaborate in the future.

*Preferential attachment*
: Produces a score for each pair of authors by multiplying the number of coauthors each has. The intuition is that authors are more likely to collaborate with someone who already coauthors a lot of papers.

*Total union of neighbors*
: Finds the total number of coauthors that each author has, minus the duplicates.

In Neo4j, we can compute these values using Cypher queries. The following function will compute these measures for the training set:

```
def apply_graphy_training_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           size([(p1)-[:CO_AUTHOR_EARLY]-(a)-
                     [:CO_AUTHOR_EARLY]-(p2) | a]) AS commonAuthors,
           size((p1)-[:CO_AUTHOR_EARLY]-()) * size((p2)-
                    [:CO_AUTHOR_EARLY]-()) AS prefAttachment,
           size(apoc.coll.toSet(
             [(p1)-[:CO_AUTHOR_EARLY]-(a) | id(a)] +
                   [(p2)-[:CO_AUTHOR_EARLY]-(a) | id(a)]
           )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                                for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                                {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

And the following function will compute them for the test set:

```
def apply_graphy_test_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           size([(p1)-[:CO_AUTHOR]-(a)-[:CO_AUTHOR]-(p2) | a]) AS commonAuthors,
           size((p1)-[:CO_AUTHOR]-()) * size((p2)-[:CO_AUTHOR]-())
                                    AS prefAttachment,
           size(apoc.coll.toSet(
             [(p1)-[:CO_AUTHOR]-(a) | id(a)] + [(p2)-[:CO_AUTHOR]-(a) | id(a)]
           )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                      for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
```

```
                        {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

Both of these functions take in a DataFrame that contains pairs of nodes in the columns `node1` and `node2`. We then build an array of maps containing these pairs and compute each of the measures for each pair of nodes.

The `UNWIND` clause is particularly useful in this chapter for taking a large collection of node pairs and returning all their features in one query.

We can apply these functions in Spark to our training and test DataFrames with the following code:

```
training_data = apply_graphy_training_features(training_data)
test_data = apply_graphy_test_features(test_data)
```

Let's explore the data in our training set. The following code will plot a histogram of the frequency of `commonAuthors`:

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    common_authors = filtered.toPandas()["commonAuthors"]
    histogram = common_authors.value_counts().sort_index()
    histogram /= float(histogram.sum())
    histogram.plot(kind="bar", x='Common Authors', color="darkblue",
                   ax=axs[index], title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Common Authors")

plt.tight_layout()
plt.show()
```

We can see the chart generated in Figure 8-8.

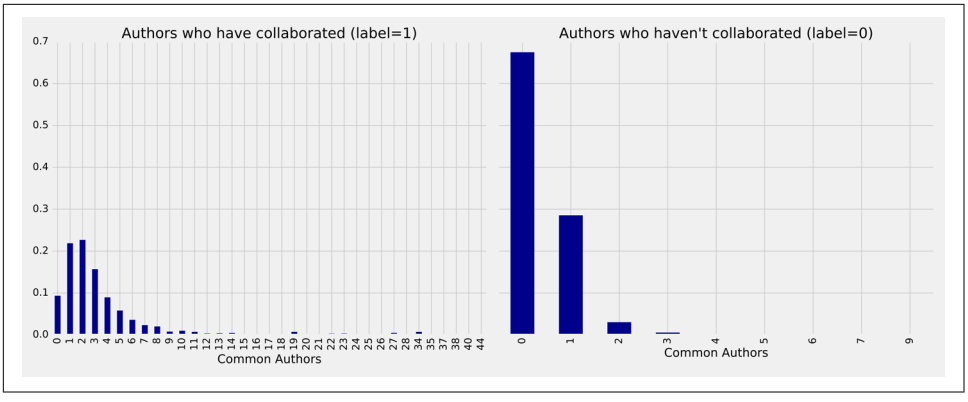


*Figure 8-8. Frequency of commonAuthors*

On the left we see the frequency of `commonAuthors` when authors have collaborated, and on the right we see the frequency of `commonAuthors` when they haven't. For those who haven't collaborated (right side) the maximum number of common authors is 9, but 95% of the values are 1 or 0. It's not surprising that of the people who have not collaborated on a paper, most also do not have many other coauthors in common. For those who have collaborated (left side), 70% have less than five coauthors in common, with a spike between one and two other coauthors.

Now we want to train a model to predict missing links. The following function does this:

```
def train_model(fields, training_data):
    pipeline = create_pipeline(fields)
    model = pipeline.fit(training_data)
    return model
```

We'll start by creating a basic model that only uses `commonAuthors`. We can create that model by running this code:

```
basic_model = train_model(["commonAuthors"], training_data)
```

With our model trained, let's check how it performs against some dummy data. The following code evaluates the code against different values for `commonAuthors`:

```
eval_df = spark.createDataFrame(
    [(0,), (1,), (2,), (10,), (100,)],
    ['commonAuthors'])

(basic_model.transform(eval_df)
 .select("commonAuthors", "probability", "prediction")
 .show(truncate=False))
```

Running that code will give the following result:

| commonAuthors | probability | prediction |
|---|---|---|
| 0 | [0.7540494940434322,0.24595050595656787] | 0.0 |
| 1 | [0.7540494940434322,0.24595050595656787] | 0.0 |
| 2 | [0.0536835525078107,0.9463164474921892] | 1.0 |
| 10 | [0.0536835525078107,0.9463164474921892] | 1.0 |

If we have a `commonAuthors` value of less than 2 there's a 75% probability that there won't be a relationship between the authors, so our model predicts 0. If we have a `commonAuthors` value of 2 or more there's a 94% probability that there will be a relationship between the authors, so our model predicts 1.

Let's now evaluate our model against the test set. Although there are several ways to evaluate how well a model performs, most are derived from a few baseline predictive metrics, as outlined in Table 8-1:

*Table 8-1. Predictive metrics*

| Measure | Formula | Description |
|---|---|---|
| Accuracy | $\frac{TruePositives + TrueNegatives}{TotalPredictions}$ | The fraction of predictions our model gets right, or the total number of correct predictions divided by the total number of predictions. Note that accuracy alone can be misleading, especially when our data is unbalanced. For example, if we have a dataset containing 95 cats and 5 dogs and our model predicts that every image is a cat we'll have a 95% accuracy score despite correctly identifying none of the dogs. |
| Precision | $\frac{TruePositives}{TruePositives + FalsePositives}$ | The proportion of *positive identifications* that are correct. A low precision score indicates more false positives. A model that produces no false positives has a precision of 1.0. |
| Recall (true positive rate) | $\frac{TruePositives}{TruePositives + FalseNegatives}$ | The proportion of *actual positives* that are identified correctly. A low recall score indicates more false negatives. A model that produces no false negatives has a recall of 1.0. |
| False positive rate | $\frac{FalsePositives}{FalsePositives + TrueNegatives}$ | The proportion of *incorrect positives* that are identified. A high score indicates more false positives. |
| Receiver operating characteristic (ROC) curve | X-Y chart | ROC curve is a plot of the Recall (true positive rate) against the False Positive rate at different classification thresholds. The area under the curve (AUC) measures the two-dimensional area underneath the ROC curve from an X-Y axis (0,0) to (1,1). |

We'll use accuracy, precision, recall, and ROC curves to evaluate our models. Accuracy is a coarse measure, so we'll focus on increasing our overall precision and recall measures. We'll use the ROC curves to compare how individual features change predictive rates.

Depending on our goals we may want to favor different measures. For example, we may want to eliminate all false negatives for disease indicators, but we wouldn't want to push predictions of everything into a positive result. There may be multiple thresholds we set for different models that pass some results through to secondary inspection on the likelihood of false results.

Lowering classification thresholds results in more overall positive results, thus increasing both false positives and true positives.

Let's use the following function to compute these predictive measures:

```
def evaluate_model(model, test_data):
    # Execute the model against the test set
    predictions = model.transform(test_data)

    # Compute true positive, false positive, false negative counts
    tp = predictions[(predictions.label == 1) &
                     (predictions.prediction == 1)].count()
    fp = predictions[(predictions.label == 0) &
                     (predictions.prediction == 1)].count()
    fn = predictions[(predictions.label == 1) &
                     (predictions.prediction == 0)].count()

    # Compute recall and precision manually
    recall = float(tp) / (tp + fn)
    precision = float(tp) / (tp + fp)

    # Compute accuracy using Spark MLLib's binary classification evaluator
    accuracy = BinaryClassificationEvaluator().evaluate(predictions)

    # Compute false positive rate and true positive rate using sklearn functions
    labels = [row["label"] for row in predictions.select("label").collect()]
    preds = [row["probability"][1] for row in predictions.select
               ("probability").collect()]
    fpr, tpr, threshold = roc_curve(labels, preds)
    roc_auc = auc(fpr, tpr)

    return { "fpr": fpr, "tpr": tpr, "roc_auc": roc_auc, "accuracy": accuracy,
             "recall": recall, "precision": precision }
```

We'll then write a function to display the results in an easier-to-consume format:

```
def display_results(results):
    results = {k: v for k, v in results.items() if k not in
                        ["fpr", "tpr", "roc_auc"]}
    return pd.DataFrame({"Measure": list(results.keys()),
                         "Score": list(results.values())})
```

We can call the function with this code and display the results:

```
basic_results = evaluate_model(basic_model, test_data)
display_results(basic_results)
```

The predictive measures for the common authors model are:

| measure | score |
|---|---|
| accuracy | 0.864457 |
| recall | 0.753278 |
| precision | 0.968670 |

This is not a bad start given that we're predicting future collaboration based only on the number of common authors in our pairs of authors. However, we get a bigger picture if we consider these measures in context with one another. For example, this model has a precision of 0.968670, which means it's very good at predicting that *links exist*. However, our recall is 0.753278, which means it's not good at predicting when *links do not exist*.

We can also plot the ROC curve (correlation of true positives and False positives) using the following functions:

```
def create_roc_plot():
    plt.style.use('classic')
    fig = plt.figure(figsize=(13, 8))
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.rc('axes', prop_cycle=(cycler('color',
                   ['r', 'g', 'b', 'c', 'm', 'y', 'k'])))
    plt.plot([0, 1], [0, 1], linestyle='--', label='Random score
                   (AUC = 0.50)')
    return plt, fig


def add_curve(plt, title, fpr, tpr, roc):
    plt.plot(fpr, tpr, label=f"{title} (AUC = {roc:0.2})")
```

We call it like this:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

We can see the ROC curve for our basic model in Figure 8-9.

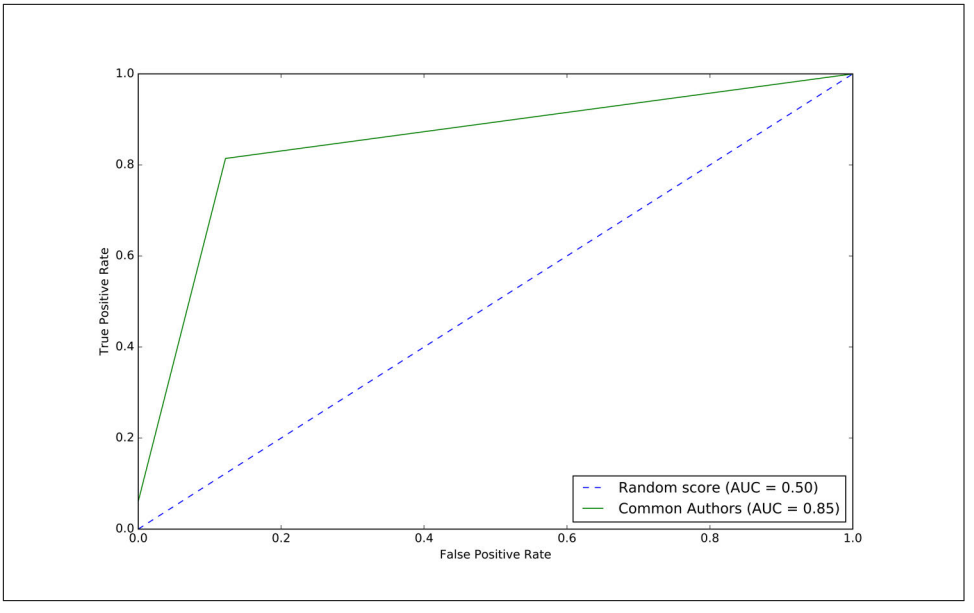


*Figure 8-9. The ROC curve for basic model*

The common authors model gives us a 0.86 area under the curve (AUC) score. Although this gives us one overall predictive measure, we need the chart (or other measures) to evaluate whether this fits our goal. In Figure 8-9 we see that as we get close to an 80% true positive rate (recall) our false positive rate reaches about 20%. That could be problematic in scenarios like fraud detection where false positives are expensive to chase.

Now let's use the other graphy features to see if we can improve our predictions. Before we train our model, let's see how the data is distributed. We can run the following code to show descriptive statistics for each of our graphy features:

```
(training_data.filter(training_data["label"]==1)
 .describe()
 .select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
 .show())

(training_data.filter(training_data["label"]==0)
 .describe()
 .select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
 .show())
```

We can see the results of running those bits of code in the following tables:

| summary | commonAuthors | prefAttachment | totalNeighbors |
|---|---|---|---|
| count | 81096 | 81096 | 81096 |
| mean | 3.5959233501035808 | 69.93537289138798 | 10.082408503502021 |

| summary | commonAuthors | prefAttachment | totalNeighbors |
|---|---|---|---|
| stddev | 4.715942231635516 | 171.47092255919472 | 8.44109970920685 |
| min | 0 | 1 | 2 |
| max | 44 | 3150 | 90 |

| summary | commonAuthors | prefAttachment | totalNeighbors |
|---|---|---|---|
| count | 81096 | 81096 | 81096 |
| mean | 0.37666469369635985 | 48.18137762651672 | 12.97586810693499 |
| stddev | 0.6194576095461857 | 94.92635344980489 | 10.082991078685803 |
| min | 0 | 1 | 1 |
| max | 9 | 1849 | 89 |

Features with larger differences between links (coauthorship) and no link (no coauthorship) should be more predictive because the divide is greater. The average value for `prefAttachment` is higher for authors who have collaborated versus those who haven't. That difference is even more substantial for `commonAuthors`. We notice that there isn't much difference in the values for `totalNeighbors`, which probably means this feature won't be very predictive. Also interesting is the large standard deviation as well as the minimum and maximum values for preferential attachment. This is what we might expect for small-world networks with concentrated hubs (superconnectors).

Now let's train a new model, adding preferential attachment and total union of neighbors, by running the following code:

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors"]
graphy_model = train_model(fields, training_data)
```

And now let's evaluate the model and display the results:

```
graphy_results = evaluate_model(graphy_model, test_data)
display_results(graphy_results)
```

The predictive measures for the graphy model are:

| measure | score |
|---|---|
| accuracy | 0.978351 |
| recall | 0.924226 |
| precision | 0.943795 |

Our accuracy and recall have increased substantially, but the precision has dropped a bit and we're still misclassifying about 8% of the links. Let's plot the ROC curve and compare our basic and graphy models by running the following code:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"],
                                basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
                                 graphy_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

We can see the output in Figure 8-10.

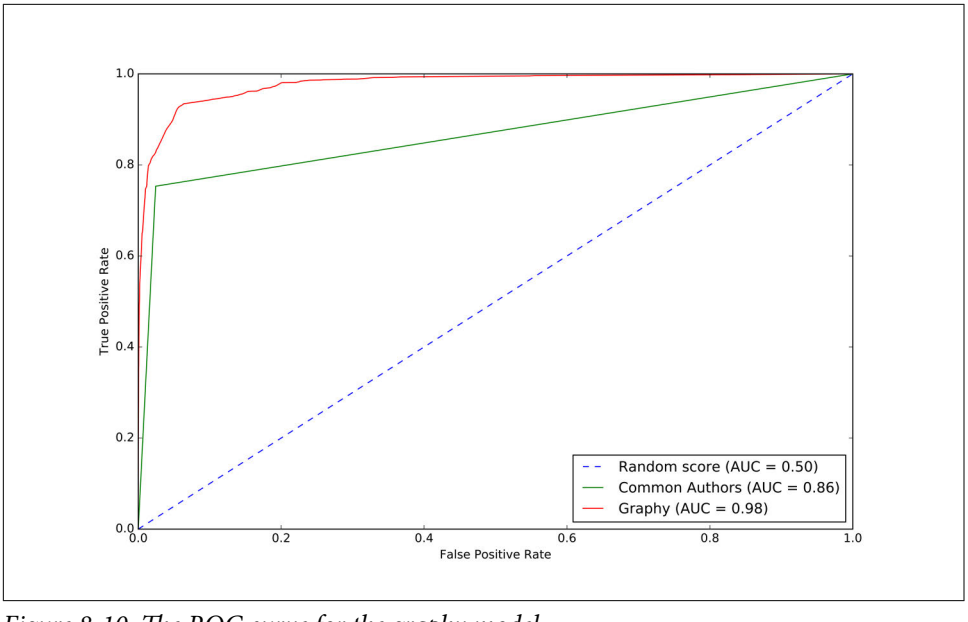


*Figure 8-10. The ROC curve for the graphy model*

Overall it looks like we're headed in the right direction and it's helpful to visualize comparisons to get a feel for how different models impact our results.

Now that we have more than one feature, we want to evaluate which features are making the most difference. We'll use *feature importance* to rank the impact of different features to our model's prediction. This enables us to evaluate the influence on results that different algorithms and statistics have.

To compute feature importance, the random forest algorithm in Spark averages the reduction in impurity across all trees in the forest. The *impurity* is the frequency at which randomly assigned labels are incorrect.

Feature rankings are in comparison to the group of features we're evaluating, always normalized to 1. If we rank one feature, its feature importance is 1.0 as it has 100% of the influence on the model.

The following function creates a chart showing the most influential features:

```
def plot_feature_importance(fields, feature_importances):
    df = pd.DataFrame({"Feature": fields, "Importance": feature_importances})
    df = df.sort_values("Importance", ascending=False)
    ax = df.plot(kind='bar', x='Feature', y='Importance', legend=None)
    ax.xaxis.set_label_text("")
    plt.tight_layout()
    plt.show()
```

And we call it like this:

```
rf_model = graphy_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

The results of running that function can be seen in Figure 8-11.

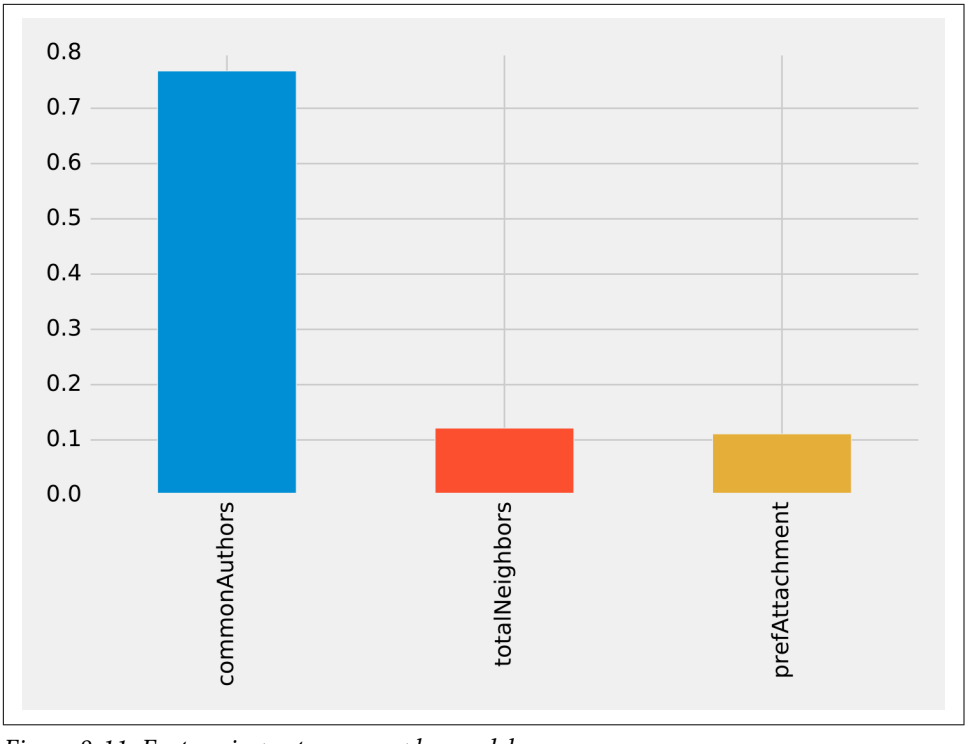


*Figure 8-11. Feature importance: graphy model*

Of the three features we've used so far, `commonAuthors` is the most important feature by a large margin.

To understand how our predictive models are created, we can visualize one of the decision trees in our random forest using the spark-tree-plotting library. The following code generates a GraphViz file:

```
from spark_tree_plotting import export_graphviz

dot_string = export_graphviz(rf_model.trees[0],
    featureNames=fields, categoryNames=[], classNames=["True", "False"],
    filled=True, roundedCorners=True, roundLeaves=True)

with open("/tmp/rf.dot", "w") as file:
    file.write(dot_string)
```

We can then generate a visual representation of that file by running the following command from the terminal:

```
dot -Tpdf /tmp/rf.dot -o /tmp/rf.pdf
```

The output of that command can be seen in Figure 8-12.

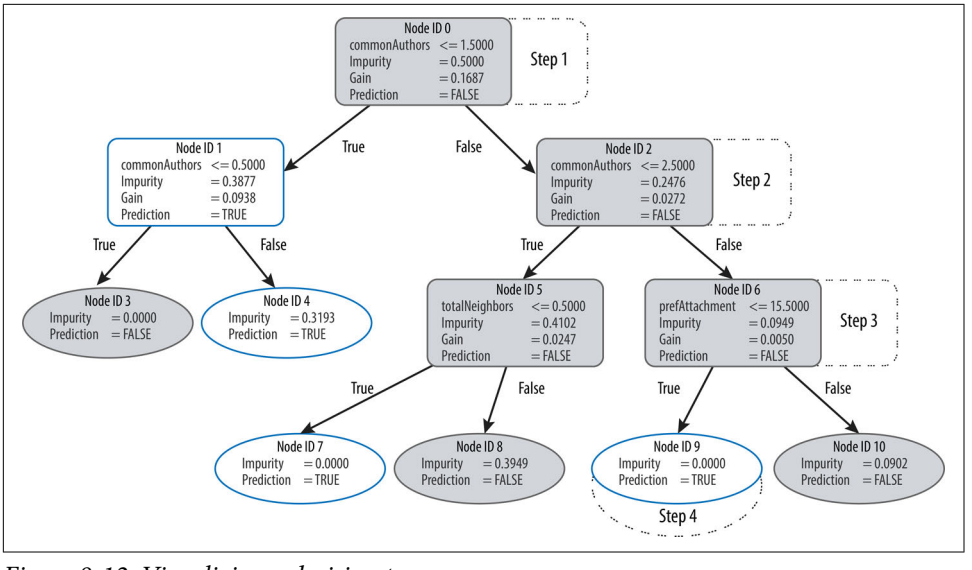


*Figure 8-12. Visualizing a decision tree*

Imagine that we're using this decision tree to predict whether a pair of nodes with the following features are linked:

| commonAuthors | prefAttachment | totalNeighbors |
|---|---|---|
| 10 | 12 | 5 |

Our random forest walks through several steps to create a prediction:

1. We start from `node 0`, where we have more than 1.5 `commonAuthors`, so we follow the `False` branch down to `node 2`.
2. We have more than 2.5 `commonAuthors` here, so we follow the `False` branch to `node 6`.
3. We have a score of less than 15.5 for `prefAttachment`, which takes us to `node 9`.
4. Node 9 is a leaf node in this decision tree, which means that we don't have to check any more conditions—the value of `Prediction` (i.e., `True`) on this node is the decision tree's prediction.
5. Finally, the random forest evaluates the item being predicted against a collection of these decision trees and makes its prediction based on the most popular outcome.

Now let's look at adding more graph features.

## Predicting Links: Triangles and the Clustering Coefficient

Recommendation solutions often base predictions on some form of triangle metric, so let's see if they further help with our example. We can compute the number of triangles that a node is a part of and its clustering coefficient by executing the following query:

```
CALL algo.triangleCount('Author', 'CO_AUTHOR_EARLY', { write:true,
  writeProperty:'trianglesTrain', clusteringCoefficientProperty:
                'coefficientTrain'});

CALL algo.triangleCount('Author', 'CO_AUTHOR', { write:true,
  writeProperty:'trianglesTest', clusteringCoefficientProperty:
                'coefficientTest'});
```

The following function will add these features to our DataFrames:

```
def apply_triangles_features(data, triangles_prop, coefficient_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           apoc.coll.min([p1[$trianglesProp], p2[$trianglesProp]])
                                              AS minTriangles,
           apoc.coll.max([p1[$trianglesProp], p2[$trianglesProp]])
                                              AS maxTriangles,
           apoc.coll.min([p1[$coefficientProp], p2[$coefficientProp]])
                                                AS minCoefficient,
           apoc.coll.max([p1[$coefficientProp], p2[$coefficientProp]])
                                                AS maxCoefficient
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]}
                           for row in data.collect()],
        "trianglesProp": triangles_prop,
        "coefficientProp": coefficient_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

Notice that we've used min and max prefixes for our triangle count and clustering coefficient algorithms. We need a way to prevent our model from learning based on the order authors in pairs are passed in from our undirected graph. To do this, we've split these features by the authors with minimum and maximum counts.

We can apply this function to our training and test DataFrames with the following code:

```
training_data = apply_triangles_features(training_data,
                                         "trianglesTrain", "coefficientTrain")
test_data = apply_triangles_features(test_data,
                                         "trianglesTest", "coefficientTest")
```

And run this code to show descriptive statistics for each of our triangle features:

```
(training_data.filter(training_data["label"]==1)
 .describe()
 .select("summary", "minTriangles", "maxTriangles",
                    "minCoefficient", "maxCoefficient")
 .show())

(training_data.filter(training_data["label"]==0)
 .describe()
 .select("summary", "minTriangles", "maxTriangles", "minCoefficient",
                                                    "maxCoefficient")
 .show())
```

We can see the results of running those bits of code in the following tables.

| summary | minTriangles | maxTriangles | minCoefficient | maxCoefficient |
|---|---|---|---|---|
| count | 81096 | 81096 | 81096 | 81096 |
| mean | 19.478260333431983 | 27.73590559337082 | 0.5703773654487051 | 0.8453786164620439 |
| stddev | 65.7615282768483 | 74.01896188921927 | 0.3614610553659958 | 0.2939681857356519 |
| min | 0 | 0 | 0.0 | 0.0 |
| max | 622 | 785 | 1.0 | 1.0 |

| summary | minTriangles | maxTriangles | minCoefficient | maxCoefficient |
|---|---|---|---|---|
| count | 81096 | 81096 | 81096 | 81096 |
| mean | 5.754661142349808 | 35.651980368945445 | 0.49048921333297446 | 0.860283935358397 |
| stddev | 20.639236521699 | 85.82843448272624 | 0.3684138346533951 | 0.2578219623967906 |
| min | 0 | 0 | 0.0 | 0.0 |
| max | 617 | 785 | 1.0 | 1.0 |

Notice in this comparison that there isn't as great a difference between the coauthorship and no-coauthorship data. This could mean that these features aren't as predictive.

We can train another model by running the following code:

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient"]
triangle_model = train_model(fields, training_data)
```

And now let's evaluate the model and display the results:

```
triangle_results = evaluate_model(triangle_model, test_data)
display_results(triangle_results)
```

The predictive measures for the triangles model are shown in this table:

| measure | score |
|---|---|
| accuracy | 0.992924 |
| recall | 0.965384 |
| precision | 0.958582 |

Our predictive measures have increased well by adding each new feature to the previous model. Let's add our triangles model to our ROC curve chart with the following code:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
                                 graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
                                   triangle_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

We can see the output in Figure 8-13.

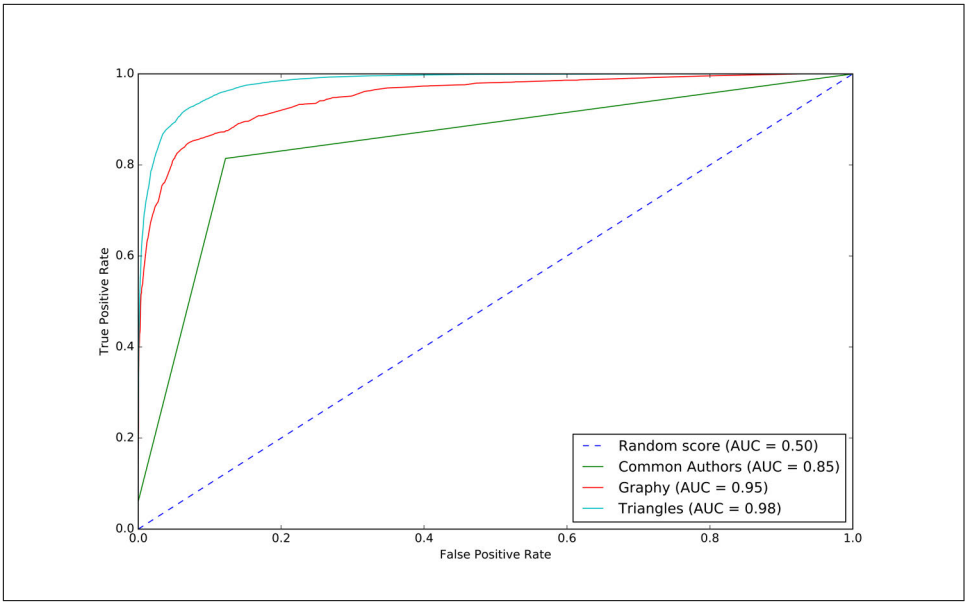


*Figure 8-13. The ROC curve for triangles model*

Our models have generally improved, and we're in the high 90s for predictive measures. This is when things usually get difficult, because the easiest gains are made but there's still room for improvement. Let's see how the important features have changed:

```
rf_model = triangle_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

The results of running that function can be seen in Figure 8-14.

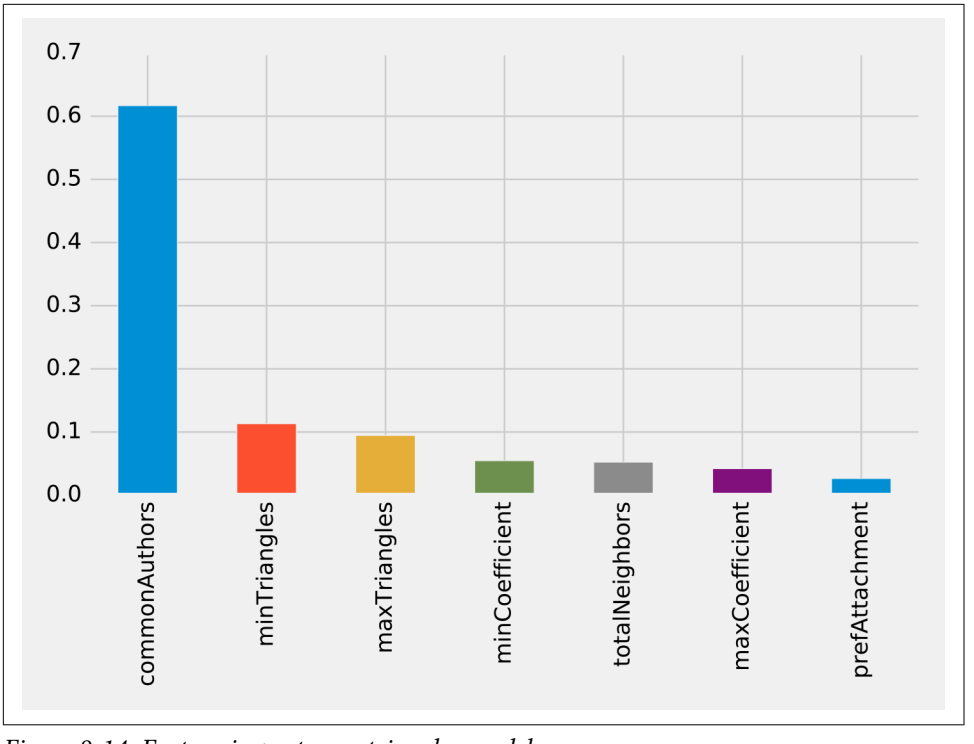


*Figure 8-14. Feature importance: triangles model*

The `common authors` feature still has the greatest single impact on our model. Perhaps we need to look at new areas and see what happens when we add community information.

## Predicting Links: Community Detection

We hypothesize that nodes that are in the same community are more likely to have a link between them if they don't already. Moreover, we believe that the tighter a community is, the more likely links are.

First, we'll compute more coarse-grained communities using the Label Propagation algorithm in Neo4j. We do this by running the following query, which will store the community in the property `partitionTrain` for the training set and `partitionTest` for the test set:

```
CALL algo.labelPropagation("Author", "CO_AUTHOR_EARLY", "BOTH",
  {partitionProperty: "partitionTrain"});

CALL algo.labelPropagation("Author", "CO_AUTHOR", "BOTH",
  {partitionProperty: "partitionTest"});
```

We'll also compute finer-grained groups using the Louvain algorithm. The Louvain algorithm returns intermediate clusters, and we'll store the smallest of these clusters in the property `louvainTrain` for the training set and `louvainTest` for the test set:

```
CALL algo.louvain.stream("Author", "CO_AUTHOR_EARLY",
                         {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTrain = smallestCommunity;

CALL algo.louvain.stream("Author", "CO_AUTHOR",
                         {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTest = smallestCommunity;
```

We'll now create the following function to return the values from these algorithms:

```
def apply_community_features(data, partition_prop, louvain_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           CASE WHEN p1[$partitionProp] = p2[$partitionProp] THEN
                     1 ELSE 0 END AS samePartition,
           CASE WHEN p1[$louvainProp] = p2[$louvainProp] THEN
                     1 ELSE 0 END AS sameLouvain
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]} for
                               row in data.collect()],
        "partitionProp": partition_prop,
        "louvainProp": louvain_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

We can apply this function to our training and test DataFrames in Spark with the following code:

```
training_data = apply_community_features(training_data,
                                         "partitionTrain", "louvainTrain")
test_data = apply_community_features(test_data, "partitionTest", "louvainTest")
```

And we can run this code to see whether pairs of nodes belong in the same partition:

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
```

```
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('samePartition',
             F.when(F.col("samePartition") == 0, "False")
                                  .otherwise("True"))
              .groupby("samePartition")
              .agg(F.count("label").alias("count"))
              .select("samePartition", "count")
              .toPandas())
    values.set_index("samePartition", drop=True, inplace=True)
    values.plot(kind="bar", ax=axs[index], legend=None,
                title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Same Partition")

plt.tight_layout()
plt.show()
```

We see the results of running that code in Figure 8-15.

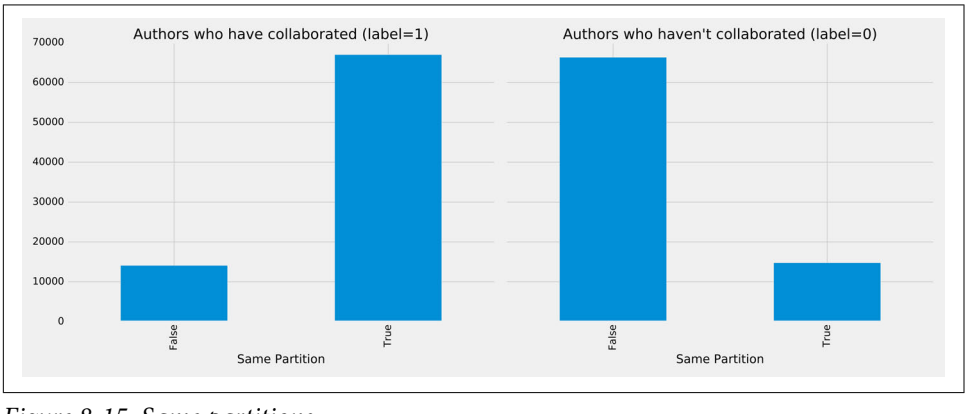


*Figure 8-15. Same partitions*

It looks like this feature could be quite predictive—authors who have collaborated are much more likely to be in the same partition than those who haven't. We can do the same thing for the Louvain clusters by running the following code:

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('sameLouvain',
              F.when(F.col("sameLouvain") == 0, "False")
                                  .otherwise("True"))
              .groupby("sameLouvain")
              .agg(F.count("label").alias("count"))
              .select("sameLouvain", "count")
              .toPandas())
```

```
        values.set_index("sameLouvain", drop=True, inplace=True)
        values.plot(kind="bar", ax=axs[index], legend=None,
                    title=f"Authors who {title} (label={label})")
        axs[index].xaxis.set_label_text("Same Louvain")

    plt.tight_layout()
    plt.show()
```

We can see the results of running that code in Figure 8-16.

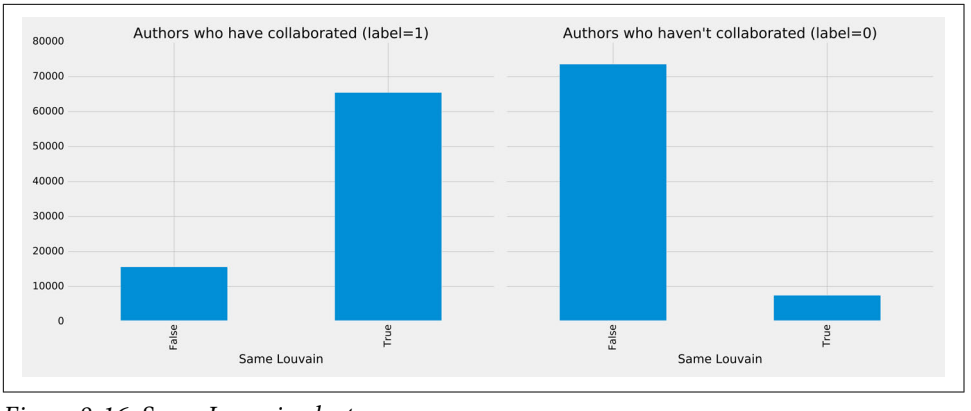


*Figure 8-16. Same Louvain clusters*

It looks like this feature could be quite predictive as well—authors who have collaborated are likely to be in the same cluster, and those who haven't are very unlikely to be in the same cluster.

We can train another model by running the following code:

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient",
          "samePartition", "sameLouvain"]
community_model = train_model(fields, training_data)
```

And now let's evaluate the model and display the results:

```
community_results = evaluate_model(community_model, test_data)
display_results(community_results)
```

The predictive measures for the community model are:

| measure | score |
|---|---|
| accuracy | 0.995771 |
| recall | 0.957088 |
| precision | 0.978674 |

Some of our measures have improved, so for comparison let's plot the ROC curve for all our models by running the following code:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
          graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
          triangle_results["roc_auc"])

add_curve(plt, "Community",
          community_results["fpr"], community_results["tpr"],
          community_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

We can see the output in Figure 8-17.

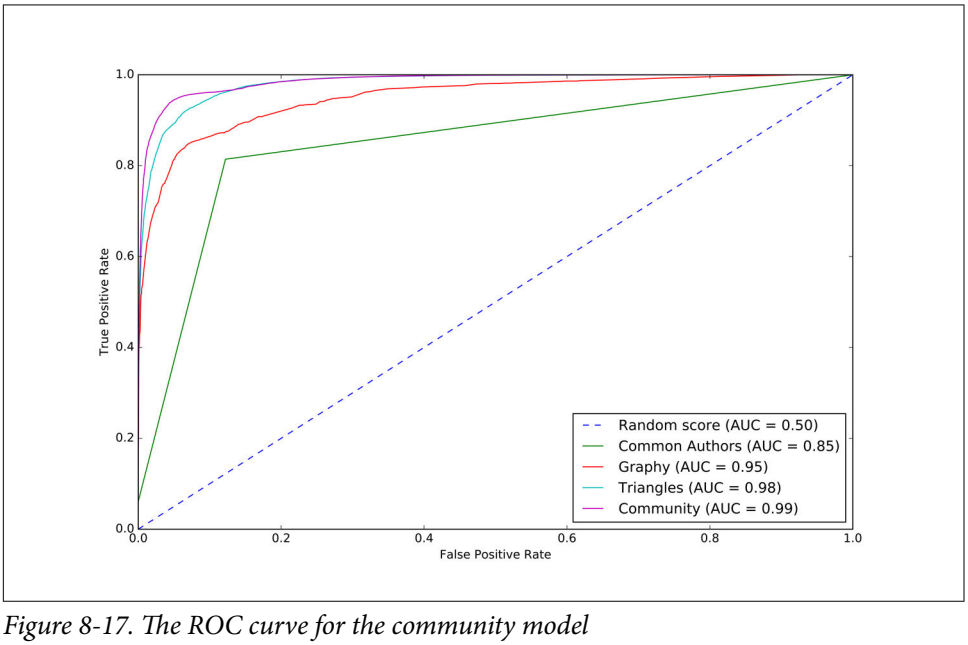


*Figure 8-17. The ROC curve for the community model*

We can see improvements with the addition of the community model, so let's see which are the most important features:

```
rf_model = community_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

The results of running that function can be seen in Figure 8-18.

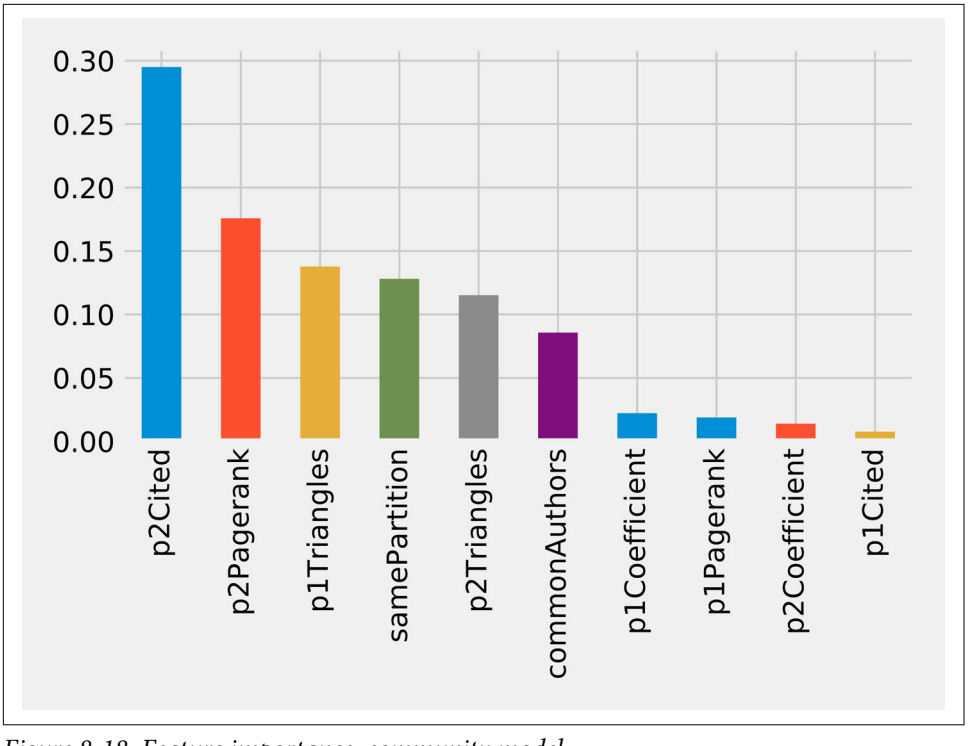


*Figure 8-18. Feature importance: community model*

Although the common authors model is overall very important, it's good to avoid having an overly dominant element that might skew predictions on new data. Community detection algorithms had a lot of influence in our last model with all the features included, and this helps round out our predictive approach.

We've seen in our examples that simple graph-based features are a good start, and then as we add more graphy and graph algorithm–based features, we continue to improve our predictive measures. We now have a good, balanced model for predicting coauthorship links.

Using graphs for connected feature extraction can significantly improve our predictions. The ideal graph features and algorithms vary depending on the attributes of the data, including the network domain and graph shape. We suggest first considering the predictive elements within your data and testing hypotheses with different types of connected features before fine-tuning.

### Reader Exercises

There are several areas to investigate, and ways to build other models. Here are some ideas for further exploration:

- How predictive is our model on conference data that we did not include?
- When testing new data, what happens when we remove some features?
- Does splitting the years differently for training and testing impact our predictions?
- This dataset also has citations between papers; can we use that data to generate different features or predict future citations?

## Summary

In this chapter, we looked at using graph features and algorithms to enhance machine learning. We covered a few preliminary concepts and then walked through a detailed example integrating Neo4j and Apache Spark for link prediction. We illustrated how to evaluate random forest classifier models and incorporate various types of connected features to improve our results.

## Wrapping Things Up

In this book, we covered graph concepts as well as processing platforms and analytics. We then walked through many practical examples of how to use graph algorithms in Apache Spark and Neo4j. We finished with a look at how graphs enhance machine learning.

Graph algorithms are the powerhouse behind the analysis of real-world systems—from preventing fraud and optimizing call routing to predicting the spread of the flu. We hope you join us and develop your own unique solutions that take advantage of today's highly connected data.