Data Science Solutions

New **Startup Science** Book

From Laptop Data Science Lab To Cloud Scale Platform

Manav Sehgal startupsci.com

Data Science Solutions

From Laptop Data Science Lab To Cloud Scale Platform

Manav Sehgal

This book is for sale at http://leanpub.com/data-science-solutions

This version was published on 2017-02-07

Contents

Data Science Solutions

The field of data science, big data, machine learning, and artificial intelligence is exciting and complex at the same time. Demand-supply equation is favoring the job seeker. The market size is in billions of dollars. Venture funding, mergers, and acquisition activity is very high in this field. Enterprises are investing significantly building capabilities in data science.

Job Opportunities. One of world's largest specialized staffing agencies, Robert Half, reports on Top Ten Technology Jobs In 2017, includes Data Scientist and Big Data Engineer jobs in high demand. Salary ranges between $116,000 to $196,000 per annum, increasing around 6% over their 2016 compensation.

Market Potential. According to Michael Dell, the data economy represents the next trillion-dollar opportunity.

Venture and Acquisitions. Nearly 100 data startups across US, India, Israel, and Singapore, raised $1.7 billion in the month of December 2016.

Enterprise Investment. IDC predicts that big data and business analytics spending will increase to more than $187 billion in 2019.

Bundled along with the huge opportunity comes fair degree of hype and complexity. Data science is a multi-disciplinary field involving statistics, mathematics, behavioral psychology, Cloud computing technologies, among many other specialized areas of study. Data science is also rapidly growing with new tools, technologies, algorithms, datasets, and use cases. For a beginner in this field, the learning curve can be fairly daunting. This is where this book helps.

> The data science solutions book provides a repeatable, robust, and reliable framework to apply the right-fit workflows, strategies, tools, APIs, and domain for your data science projects.

This book takes a solutions focused approach to data science. Each chapter meets an end-to-end objective of solving for data science workflow or technology requirements. At the end of each chapter you either complete a data science tools pipeline or write a fully functional coding project meeting your data science workflow requirements.

Coding Goals

In this chapter we will code a data science project which trains a machine learning model and predicts classification of species of a flower based on given set of features. Using powerful yet easy to learn technology stack, we will accomplish our task in less than 20 lines of coding logic.

You can code along instructions in this chapter. The project is available at a dedicated GitHub repository.

```
https://github.com/Startupsci/data-science-notebooks/
```

Select the *iris-decision-tree.ipynb* file to directly run the project on GitHub or download and run locally.

The repository also contains self-learning notebooks going over prerequisite Python concepts including data structures and file operations. For this chapter you may want to run the following notebooks as well.

- python-data-structures-list.ipynb
- python-data-files.ipynb

Learning Goals

The learning goals for this chapter will introduce relevant technologies to run through most of data science solutions workflow end-to-end.

- Setup Python development environment.
- Wrangle data using Pandas library.
- Acquire and analyse Iris multi-variate dataset.
- Model using decision tree machine learning algorithm.
- Visualize using Seaborn visualization library.

Seven stages of data science solutions workflow

Every chapter in this book will go through one or more of these seven stages of data science solutions workflow.

Question. Problem. Solution.

Before starting a data science project we must ask relevant questions specific to our project domain and datasets. We may answer or solve these during the course of our project. Think of these questions-solutions as the key requirements for our data science project. Here are some templates that can be used to frame questions for our data science projects.

- Can we classify an entity based on given features if our data science model is trained on certain number of samples with similar features related to specific classes?
- Do the samples, in a given dataset, cluster in specific classes based on similar or correlated features?
- Can our machine learning model recognize and classify new inputs based on prior training on a sample of similar inputs?
- Can we analyse the sentiment of a given sample?

Acquire. Search. Create.

This stage involves data acquisition strategies including searching for datasets on popular data sources or internally within your organization. We may also create a dataset based on external or internal data sources.

The acquire stage may feedback to the question stage, refining our problem and solution definition based on the constraints and characteristics of the acquired datasets.

Wrangle. Prepare. Cleanse.

The data wrangle stage prepares and cleanses our datasets for our project goals. This workflow stage starts by importing a dataset, exploring the dataset for its features and available samples, preparing the dataset using appropriate data types and data structures, and optionally cleansing the data set for creating model training and solution testing samples.

The wrangle stage may circle back to the acquire stage to identify complementary datasets to combine and complete the existing dataset.

Analyze. Patterns. Explore.

The analyze stage explores the given datasets to determine patterns, correlations, classification, and nature of the dataset. This helps determine choice of model algorithms and strategies that may work best on the dataset.

The analyze stage may also visualize the dataset to determine such patterns.

Model. Predict. Solve.

The model stage uses prediction and solution algorithms to train on a given dataset and apply this training to solve for a given problem.

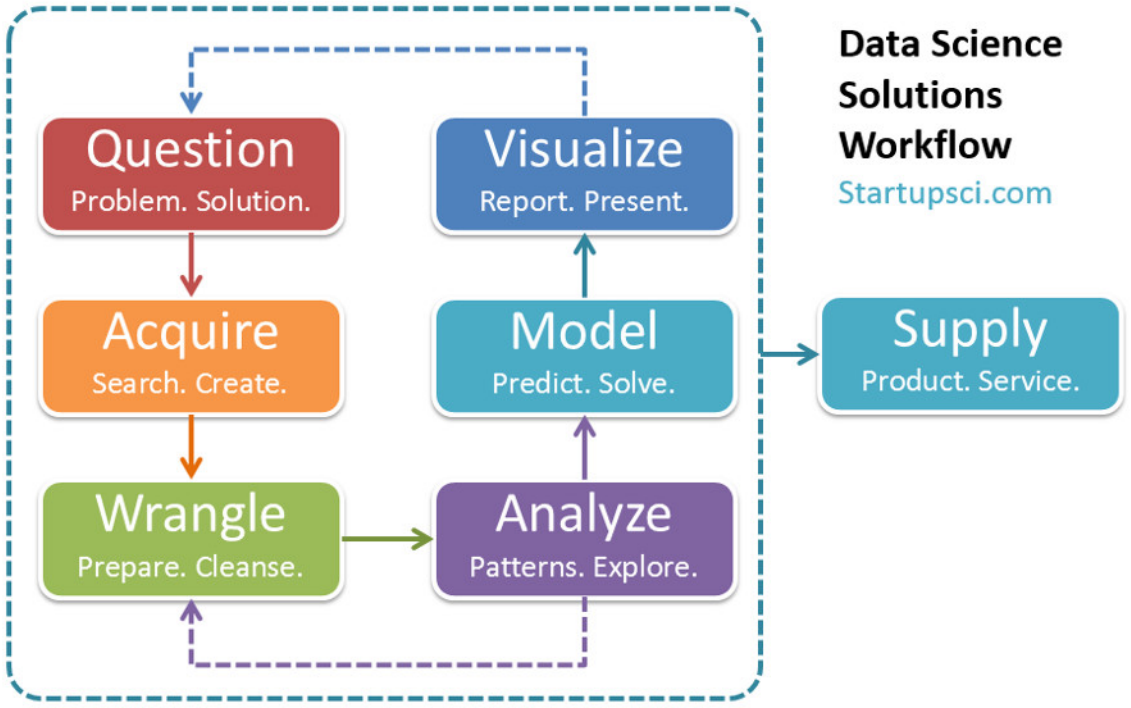
Visualize. Report. Present.

The visualization stage can help data wrangling, analysis, and modeling stages. Data can be visualized using charts and plots suiting the characteristics of the dataset and the desired results.

Visualization stage may also provide the inputs for the supply stage.

Supply. Products. Services.

Once we are ready to monetize our data science solution or derive further return on investment from our projects, we need to think about distribution and data supply chain. This stage circles back to the acquisition stage. In fact we are acquiring data from someone else's data supply chain.



Here is how we extend and enhance the data science solutions workflow.

Strategies. We will apply various strategies for our data science initiative. Data layering is one such strategy, inspired by online mapping systems, layering new data sets over base data to solve for new use cases.

Tools. We will also build our toolchain for data science solutions. We will include tools on our laptop, like Excel and Python Notebooks, and seamlessly integrate these with Cloud-scale tools.

APIs. We will be building our own API and we will consume external APIs for complementing our data science solutions workflow.

Domains. Defining the problem-solution domains is an essential step for delivering domain specific data science solutions. We intend to cover case studies in banking and financial services, retail, location intelligence, and healthcare among others.

Learning path for data science

In this book we accomplish several learning goals covering the multi-disciplinary field of data science.

Open Source Technologies. Open source technologies offering solutions across the data science stack. These include technologies like D3 for visualizing data, Hadoop, Spark, and many others.

Enterprise Tools. Commercial products offering enterprise scale, power user solutions across the data science stack. These include products like Tableau for exploring data, Trifacta for wrangling data, among others. It is important to learn about these products, while knowing the editions and alternatives available in the open source stack.

Data Science Algorithms. Algorithms to prepare and analyse datasets for data science projects. APIs offering algorithms for computer vision, speech recognition, and natural language processing are available from leading Cloud providers like Google as a paid service. Using open source libraries like Tensorflow, Theano, and Python Scikit-learn you can explore more developing your own data science models.

Cloud Platform Development. Developing full-stack frontend apps and backend APIs to deliver data science solutions and data products.

Data Science Thinking. These learning goals are inter-dependent. An open source technology may be integrated within an Enterprise product. Cloud platform may deliver the data science algorithms as APIs. The Cloud platform itself may be built using several open source technologies.

We will evolve Data Science Thinking over the course of this book using strategies and workflow which connect the learning goals into a unified and high performance project pipeline.

Data science strategies

Data science does not need to be a huge money and time sink. Data science is for everyone, not just coders and data scientists. This section lists strategies which enable marketers, CXOs, and functional managers to become power users contributing to lean agile data science projects.

Reuse Existing Skills. If you are an Excel power user you will find it easier than ever to reuse your skills, for an important workflow within most data science projects, doing data wrangling and analysis. One of the most important tasks for any data science project is to prepare the data used by the models. Improving quality of this input data and analyzing interesting patterns early in the project, can save significant downstream investments in hand-offs and iterations.

Laptop To Cloud. Our end-to-end development and prototyping environment will be setup on a laptop, including mocks for our data store. We will then scale over to the Cloud using easy, managed, and repeatable path.

Less Is More. We prefer fewer lines of code to get to the same functionality. Minimal configuration options, preferably auto-generated configuration. Few core libraries and dependencies.

Data First Design. We will design our project modeling our data first, driving our API design based on this data model and subsequently designing our client UI based on the API design.

API Is Your Currency Exchange. As you think about your data science project, think API as your first class currency which creates value for your business and your customers. At each granular aspect of your workflow, think about APIs you can exchange with other systems, workflows, departments, partners, or customers.

Most technologies and tools covered in this book expose or consume APIs. As business users how you can contribute to API design and development is by helping define the domain vocabulary.

> How is your business described internally and externally. This formal taxonomy can create the basis of a good API design.

As a power user you can also consume APIs that your business produces or your products consume. So think of yourself as a quality assurance layer or as an internal customer evaluating and comparing the APIs on offer.

Data Science Friendly. Our choice of development environment will be suitable for data science projects.

Platform As A Service. We will prefer managed Cloud services or Platform as a Service over self-managed virtual machines or container solutions. This reduces our learning curve significantly, while focusing on best practice managed guidance for a fast, reliable minimum viable product development. All major Cloud providers are offering both vendor-managed and self-managed Cloud options.

Over time the ROI of moving from self managed to vendor managed Cloud is on similar scale as moving from own infrastructure to Cloud.

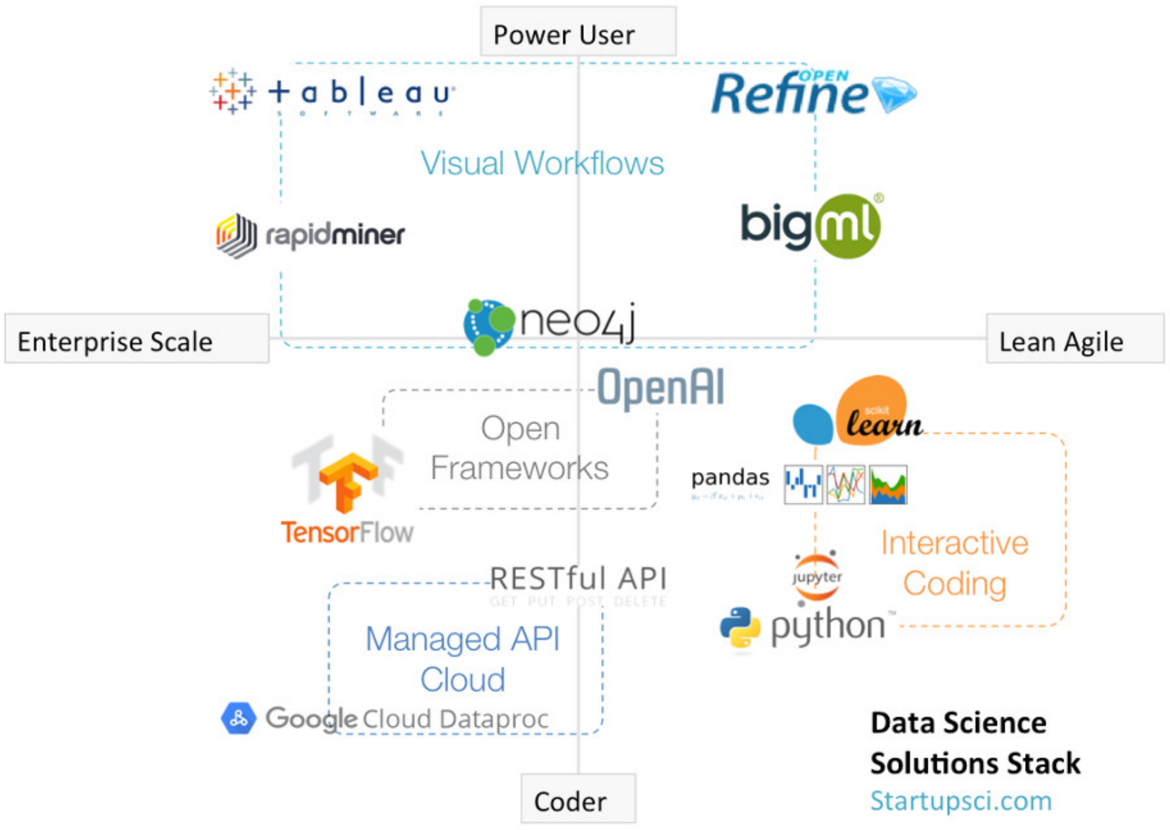
Learn As You Build. Only learn what is necessary to accomplish the current project. Doing a project end-to-end is learning in itself. We prefer a breadth-first approach to learning concepts used within a project. So, for example, if your project requires a certain programming language, we only explain the language features used within that project.

Minimize DevOps Burden. Developer operations or DevOps is a crucial function in any software team. However, most DevOps workflows require significant learning curve, constant updates in tooling, and constraints based on what is/not included in the developer toolchain. Outsource "app plumbing" to the managed developer platforms. Minimize custom DevOps workflow tooling.

Interactive Developer Experience. While user experience (UX) focuses on your external customers, developer experience (DX) addresses the needs of your data science project team. Developer experience spans tools used across the coding lifecycle, design of APIs produced and consumed by your products, and collaborative product management workflows. The objective of DX is to make coding an enjoyable, productive, and collaborative experience for the entire team, not just the coders and designers.

Technology stack for data science

There are several technologies which help achieve the data science strategies. We can map these technologies along four quadrants with the vertical axis representing power user to coder personas as data science team members. The horizontal axis represents enterprise scale complexity and budgets compared to startup speed and simplicity.



Being within any of the quadrants is dependent on your organization size and data science maturity. What is more important is to know where is your sweet spot.

Certain technologies play a fundamental ecosystem-like role connecting to others across the quadrants. These include producing and consuming RESTful APIs and coding in the Python environment.

Few technologies are centered across all quadrants, like Neo4j. These are your safe bets as you can scale these technologies or prototype with these cost effectively, while engaging technical and functional team members equally.

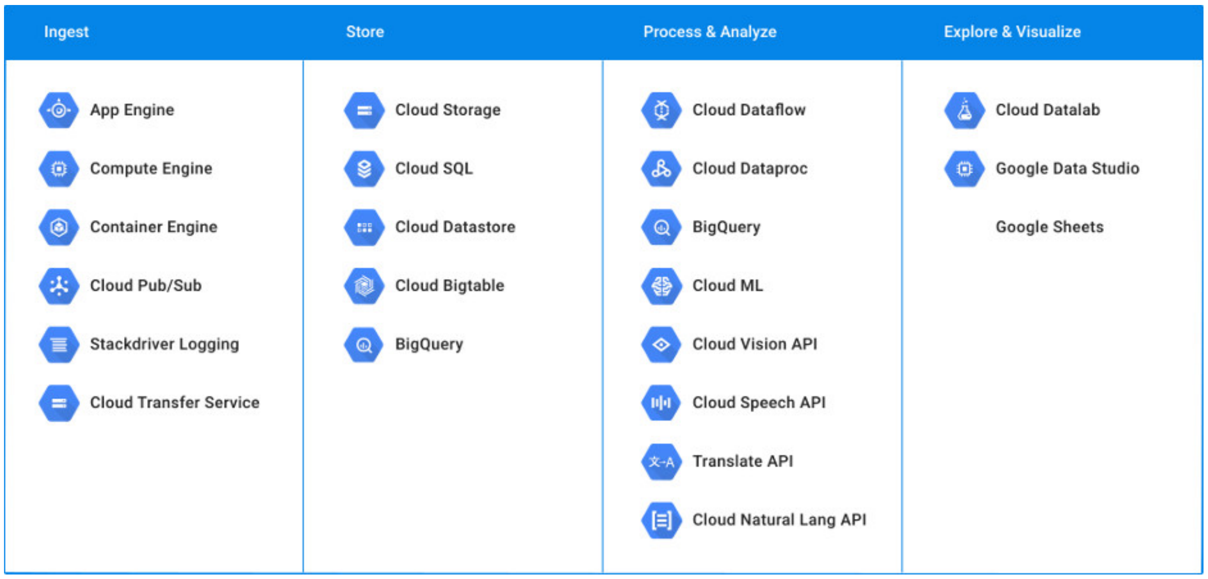
You can draw this map with your favorite tools and technologies or even move some of these over to other quadrants depending on how you put these to use. For instance, while OpenRefine is on extreme top right in our map, you could use the Python language to build powerful data wrangling expressions for OpenRefine workflow and it moves further down to the bottom right.

The important takeaway is to make this map your own and draw out your stack to understand your organization's specific use case.

Python. Python is an excellent choice as a programming language for data science. Python libraries like Pandas, SciPy, NumPy, Matplotlib, and Scikit-learn provide essential capabilities for various stages in our workflow.

Python programming environment is also ideal for high fidelity developer experience in data science. It offers interactive coding with Jupyter Notebooks which enable intermixing of documentation in Markdown/HTML, code, module outputs, plots and charts, all within one seamless, easy to share, easy to scale notebook. Jupyter notebooks can be hosted and run directly on GitHub. Google Datalab is a Cloud scale solution designed for data visualization built on Jupyter Notebooks.

Google Cloud Platform. Google Cloud platform offers several managed options including Cloud Dataproc which provides a managed Spark, Hadoop, Hive, and Pig stack for your data science projects at scale. Google Cloud Endpoints provides managed APIs at scale. For building conversational interfaces, like ChatBots, Firebase provides realtime database and user authentication as a managed service. The list goes on including BigQuery for data warehousing and Dataflow for streaming or batch data processing pipelines.



Ingest	Store	Process & Analyze	Explore & Visualize
App Engine	Cloud Storage	Cloud Dataflow	Cloud Datalab
Compute Engine	Cloud SQL	Cloud Dataproc	Google Data Studio
Container Engine	Cloud Datastore	BigQuery	Google Sheets
Cloud Pub/Sub	Cloud Bigtable	Cloud ML	
Stackdriver Logging	BigQuery	Cloud Vision API	
Cloud Transfer Service		Cloud Speech API	
		Translate API	
		Cloud Natural Lang API	

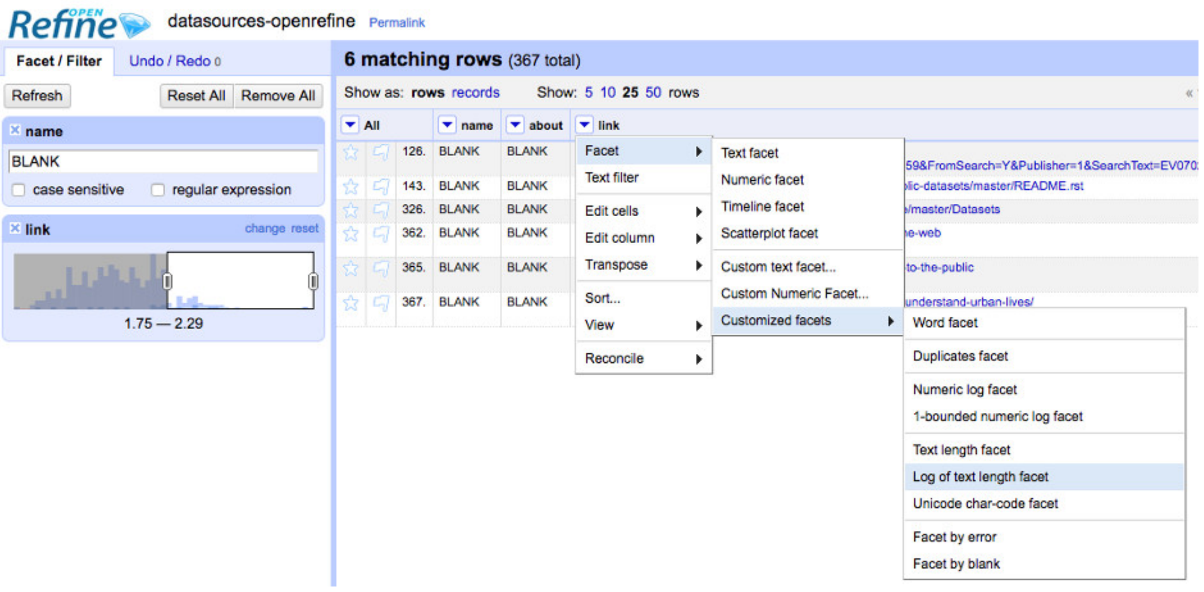
This image is sourced from the Google Solutions article Data Lifecycle on Google Cloud Platform.

```
https://cloud.google.com/solutions/data-lifecycle-cloud-platform
```

Neo4j Graph Database. Graph database is what runs behind the scenes when you search on Google. When you connect with your friends on Facebook, your connections and interests form a graph database that Facebook calls the Open Graph. In this book we will learn to create our own knowledge graph database.

Neo4j graph database exposes an API for every database operation. Using Python you can query a Neo4j graph served on a Laptop or Cloud, complete with user authentication and ACID transactions, in less than 7 lines of code!

OpenRefine Data Wrangling. We will use OpenRefine as an alternative solution to using Python and Pandas for our data wrangling workflow. OpenRefine is an open source tool which starts with a CSV or Excel file among many other data sources. It uses familiar tabular, editable cells, sortable columns based layout to present and manipulate data. It also adds interactive power tools for data wrangling and analysis of your data using algorithms and techniques used by data scientists.



All	name	about	link
126.	BLANK	BLANK	59&FromSearch=Y&Publisher=1&SearchText=EV070
143.	BLANK	BLANK	lic-datasets/master/README.rst
326.	BLANK	BLANK	/master/Datasets
362.	BLANK	BLANK	e-web
365.	BLANK	BLANK	to-the-public
367.	BLANK	BLANK	understand-urban-lives/

For example you can use algorithms to cluster values in your data columns to identify duplicates. You can also plot numerical values in a histogram to understand the distribution and outliers within your data columns or features.

Tensorflow. Google open sources TensorFlow, a library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

Originally authored by the Google Brain team, primary use case of TensorFlow is in Machine Learning and AI using neural networks. TensorFlow adopts Python as its primary language. It is possible to write a basic machine learning TensorFlow application within 15 lines of easy to understand Python code.

Universe. OpenAI is a non-profit artificial intelligence research company $1B funded by Reid Hoffman, Elon Musk, and Peter Thiel among other Silicon Valley leaders. OpenAI open sources Universe, a software platform for evaluating and training intelligent agents across games, websites and other applications. These AI agents “operate” the games and apps like humans do, using keyboard, mouse, and “seeing” the screen pixels. Universe makes more than 1,000 training environments available, including popular games like GTA planned on the roadmap. You can create your first AI agent on OpenAI Universe in just 9 lines of Python code.

Starting a data science project

Let us roll up our sleeves and apply the workflow stages and development strategies to code our first data science project. Our first step is defining the problem-solution space. We will pick a fundamental dataset for this project, the Iris flower species classification dataset. It consists of three Iris flower species, setosa, verginica, and versicolor, identified by four features, their sepal length and width, and petal length and width. As you can imagine each flower is distinct with subtle variations across all these features.

The problem question for our data science project is as follows.

> Can our data science model predict the species and classify a flower based on the four features, if it has seen a modest sample of such flowers?

As you will note, that this classification problem presents itself in variety of fields and business areas. For example, segmenting customers based on their behavior on a website. If the behavior or user actions can be classified as features, collecting adequate number of samples for known user segments, such a model could predict what future offers may be relevant for new users exhibiting similar behavior, that may have worked for known user segments in the past.

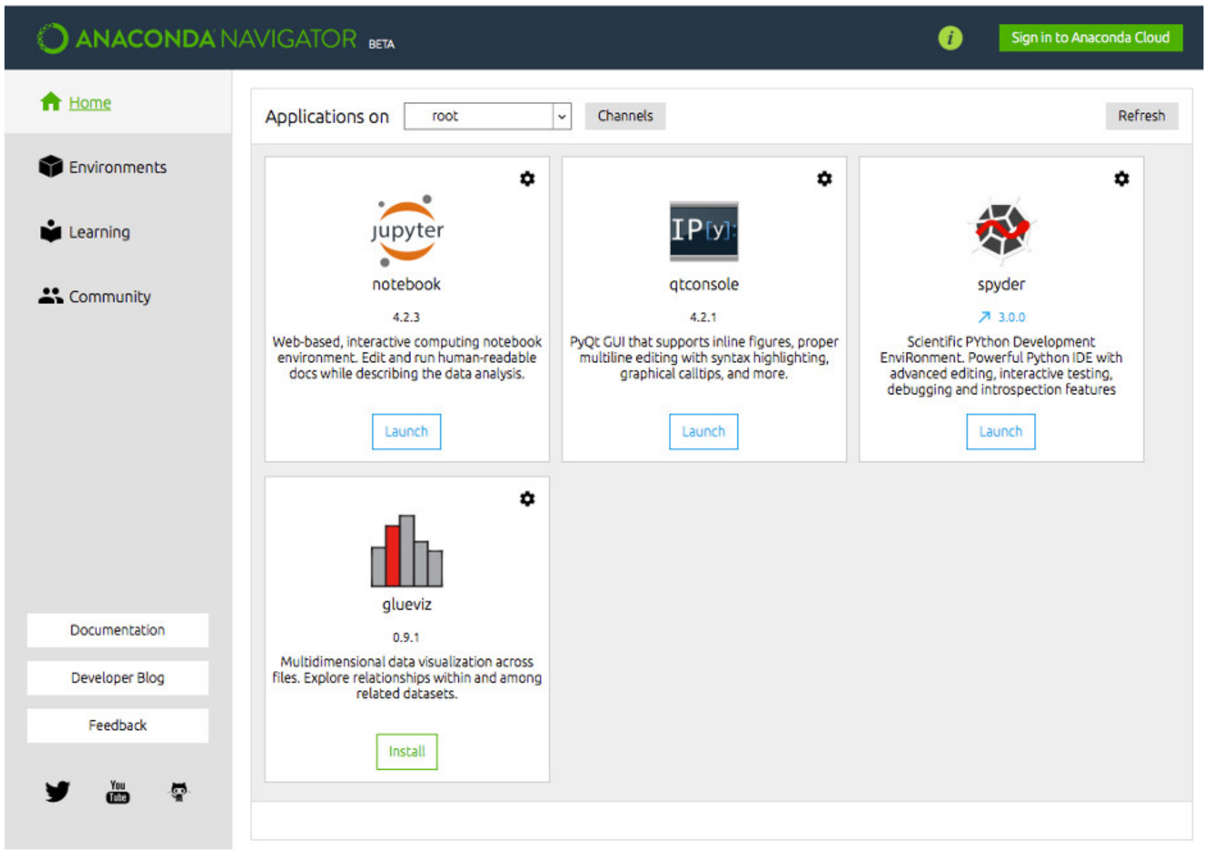
Before we start our next step of acquiring the Iris dataset, let us quickly setup our development environment.

Anaconda Python environment manager

We start by setting up Python development environment on your laptop. Mac OS X comes with Python 2.7 installed, however we will use a popular Python dependency, package, and environment manager called Conda. Visit the downloads page for Anaconda (https://www.continuum.io/downloads) and choose the Python version 2.7 graphical installer for your OS. Windows and Mac versions are available.

Python 2.7 vs 3.x. Python has two major releases. Newest release 3.x has several language enhancements. However, 2.7 has much better support from existing ecosystem of libraries and platforms, including the Google Cloud Platform. Google App Engine standard is only available for Python 2.7. We will mostly use Python 2.7 as it provides 3.x porting using established tools like Modernize and Futurize.

Once Anaconda is installed, you can access the Anaconda Navigator from Applications.



Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Multidimensional data visualization across files. Explore relationships within and among related datasets.

To check that the Python version we are using is provided by Anaconda, we can open a new Mac Terminal window and type *python -V* command.

```
python -V
Python 2.7.12 :: Anaconda 4.2.0 (x86_64)
```

Alternatively you can install Python and its dependencies manually, however this is not the recommended path if you are relatively new to Python development.

Jupyter Python Notebook

Now we are ready to write our first few lines of code. We can fire up Python interactive notebook by Jupyter, included with Anaconda.

You do so by changing to folder where your project lives and firing the following command in the Terminal.

```
jupyter notebook
```

This runs the Jupyter notebook server and opens the listing for current folder in your browser. Create a new notebook using the *New* button on top right using the *Python (Conda root)* option to choose the Anaconda installed version of Python. If you have downloaded the code for this project into this folder, you can simply click on the *iris-decision-tree.ipynb* file in the list and go from there.

Benefits of using notebooks

- Great for rapid prototyping as you can incrementally develop while checking intermediate results of your code snippets.
- Suitable for data science coding as it involves multiple steps and workflow stages, like a pipeline of using multiple tools or libraries.
- Works out-of-the-box with Anaconda installed Python packages.
- Easy to share and publish. When published on GitHub, can run interactively on GitHub without any setup.
- Writing print function for the last printable statement in cells is not required. Saves a few keystrokes.
- Can include formatted HTML using Markdown documentation inline with code snippets and results. Great for data blogging use cases.
- Can convert a notebook to PDF, HTML, and other formats using *jupyter nbconvert* command from the Terminal.

Watch out when using notebooks

- Notebooks maintain state of variables used in prior cells. If you remove or rename a variable, chances are the old variable still remains. This can lead to bugs which are hard to track. When in doubt, shutdown and restart a notebook and run all cells.
- Visualizing and displaying inline charts and plots from libraries like matplotlib requires you to declare *%matplotlib inline* statement at the start of your notebook.

Importing dependencies

We start our Python notebook by importing the required libraries or dependencies.

```
%matplotlib inline
import pandas as pd
import numpy as np
from sklearn import datasets
from sklearn import tree
import seaborn as sns
import random as rnd
```

The *sklearn* library is one of the most popular Python libraries for machine learning and data science, called *scikit-learn*. The *pandas* library makes it easy to process and analyse datasets. The *numpy* dependency stands for numerical Python and among other capabilities it offers performance optimized data structures for manipulating arrays and lists. The *seaborn* library is useful for visualizing data using plots and charts. We also import a random number generator for our project.

Acquire and wrangle data

The next stage in our workflow is to acquire the required Iris dataset and prepare it for analysis. Fortunately *sklearn* offers several sample datasets using a single line of code.

```
iris = datasets.load_iris()
```

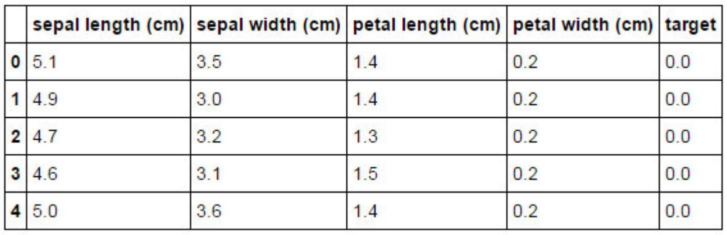
The Iris dataset is well prepared and created for the purpose of data science learning projects, so it works out-of-the-box and does not need much preparation or cleansing.

```
df = pd.DataFrame(data=np.c_[iris['data'], iris['target']],
                  columns=iris['feature_names'] + ['target'])

df.head()
```

When using Jupyter Python notebook you can see output of the last statement on each cell without using the *print()* function. This only works for statements which return a printable output. When converting this code to a Python compiled project you will need to wrap around these statements using the print function as appropriate.

All we need to do is to convert the dataset to a *pandas* DataFrame, which enables us to view the data as a table and also perform basic analysis.



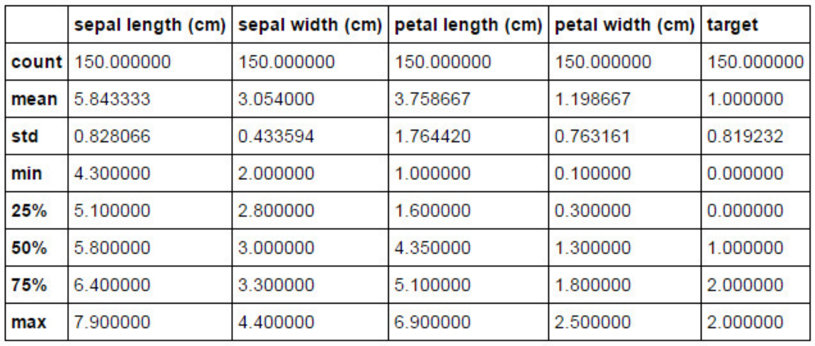
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0

As you will note the samples listed are varied in terms of length and width features. The target feature specifies the species or classification of these samples using numerical ids zero for setosa, one for versicolor, and two for verginica.

Analyse and visualize

We can use DataFrame functions to further explore our dataset.

```
df.describe()
```



	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667	1.000000
std	0.828066	0.433594	1.764420	0.763161	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Using the *describe* function we can learn several aspects of our dataset.

- There are 150 samples (count) available in our dataset.
- The classifications (target) include numerical ids 0, 1, and 2.
- The dataset variables contain variety of numerical values distributed across different ranges.
- The dataset is ordered on target, containing around 50 samples (25%, 50%, 75% rows) per species.

We can confirm that the first few samples are ordered by target=0 or *setosa* specie.

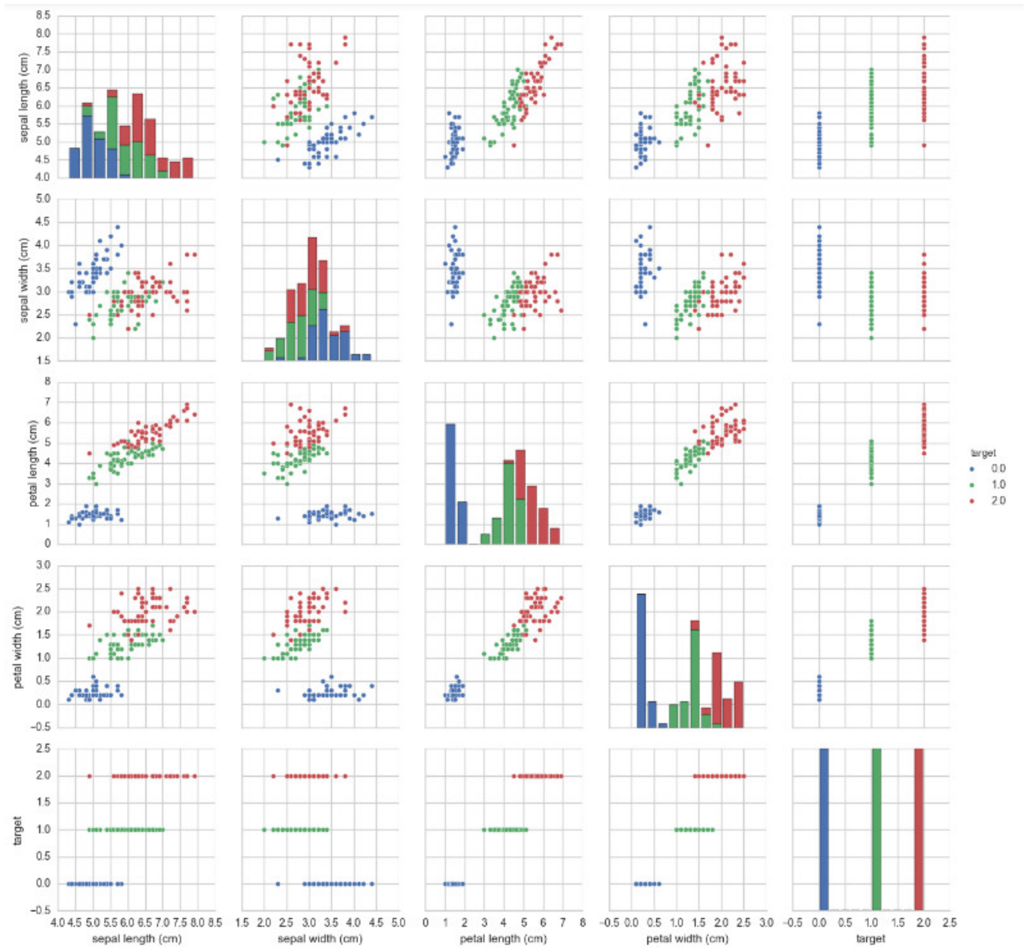
```
df.head()
```

Similarly the last few samples are ordered by target=2 or *verginica* specie.

```
df.tail()
```

Visualizing the dataset also helps us understand the underlying relationships and patterns.

```
sns.set_style('whitegrid')
sns.pairplot(df, hue='target')
```



Our dataset is 150 rows x 5 columns of numbers. There is only so much we can learn from staring at set of 750 numbers.

The visualization helps us identify patterns in our dataset which guide us in deciding an appropriate model for next stage of our workflow.

- Studying the scatter plots, the features exhibit somewhat linear correlation among length and width of Sepal and Petal across species. As the length increases/decreases, so does the width of the Sepal and Petal. This is more pronounced in Petals when compared to Sepals.
- The scatter plots also clearly indicate that the species form identifiable clusters based on their features.

- The histogram suggests a combination of overlapping and distinct distributions of features across species.

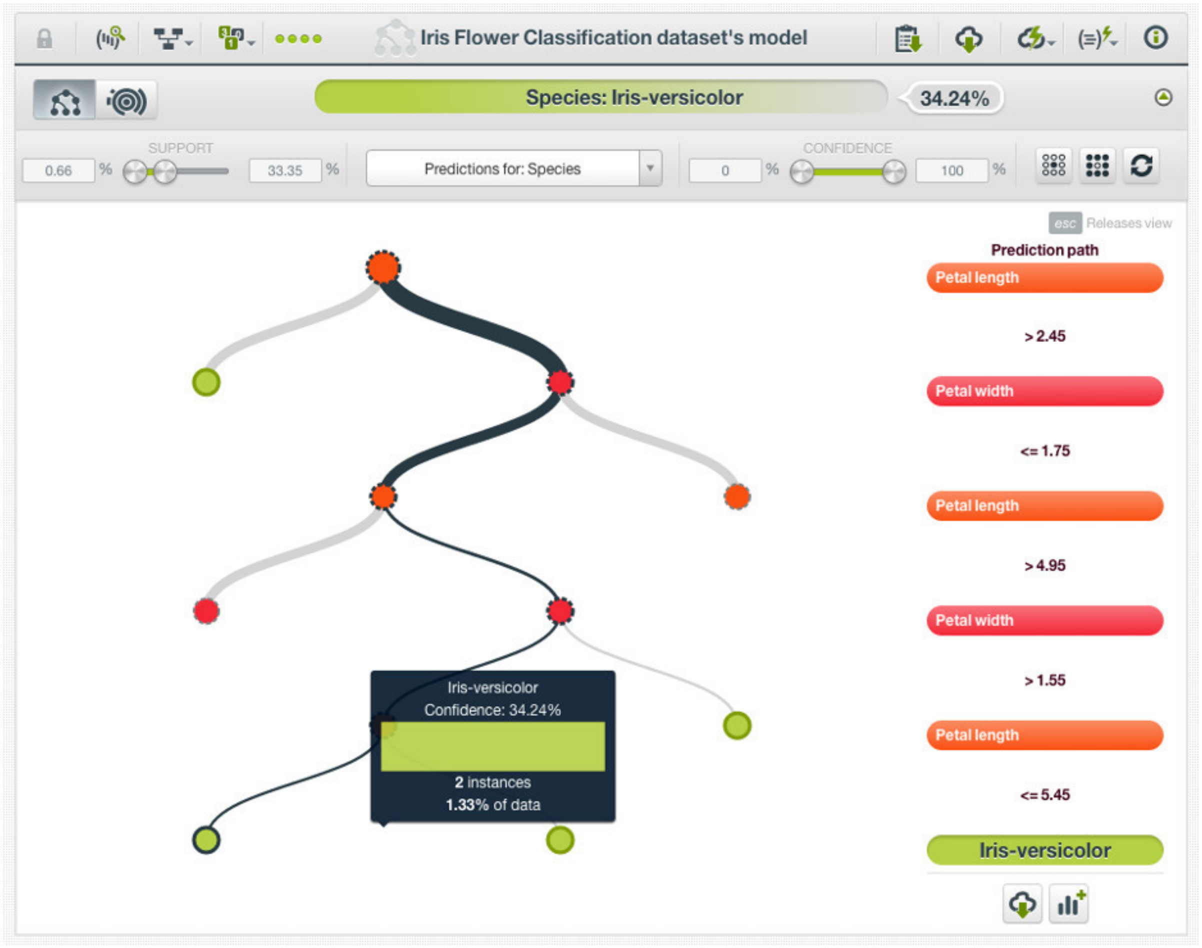
Based on these observations and the question we are solving in this project, we choose the appropriate model algorithm for our task.

Model and predict

Based on our question, we are solving for what is known in the data science and machine learning field as a classification problem. There are several algorithms available for classification solutions. These include Decision Trees, Support Vector Machines (SVM), Stochastic Gradient Descent (SGD), Nearest Neighbors (KNN), and Gaussian Process Classification (GCP), among others.

Based on our observations, nature of our dataset, and problem-solution requirement we select the Decision Tree algorithm for our model.

- Requires minimal data preparation. We will work on the as provided dataset without changing it.
- Can handle numerical and categorical data.
- Can perform multi-class classification on a given dataset.



Prediction path
Petal length
> 2.45
Petal width
<= 1.75
Petal length
> 4.95
Petal width
> 1.55
Petal length
<= 5.45
Iris-versicolor

Think of a decision tree as set of if-then-else conditional statements. The conditions are the features and variables. For example sepal length > 2, or petal width < 0.5. The result of the statements are classes or species based on meeting certain conditions or features.

Machine learning algorithms need to be trained on a number of samples and targets or classes related to these samples. Following this training, the algorithm can be used to predict a solution based on a test sample.

Let us create a training dataset from our overall Iris dataset.

```
X = iris.data[0:150, :]
X.shape
```

This results in *(150, 4)* as the shape of the dataset which indicates 150 samples across four features. The variable *X* now represents of training features dataset.

```
Y = iris.target[0:150]
Y.shape
```

Similarly we extract the training target or classification dataset as the *Y* variable.

Next we create our test samples from the original dataset. This is a recommended approach for validating our models using the same dataset to train and test.

```
setosa_index = rnd.randrange(0, 49)
test_setosa = [iris.data[setosa_index, :]]
X = np.delete(X, setosa_index, 0)
Y = np.delete(Y, setosa_index, 0)
test_setosa, iris.target_names[iris.target[setosa_index]], X.shape, Y.\
shape
```

The randomized output *([array([5.1, 3.3, 1.7, 0.5])], 'setosa', (149, 4), (149,))* displays sample List and target classification name along with the new shape of our *X* and *Y* training data.

Note that we are randomizing the test sample extraction from the training dataset. We are also removing the test sample from this dataset so see if our model can still recognize the given specie based on the remaining samples we are using for training the model.

```
virginica_index = rnd.randrange(100, 150)
test_virginica = [iris.data[virginica_index, :]]
X = np.delete(X, virginica_index, 0)
Y = np.delete(Y, virginica_index, 0)
test_virginica, iris.target_names[iris.target[virginica_index]], X.sha\
pe, Y.shape
```

The output *([array([6.7, 3.3, 5.7, 2.1])], 'virginica', (148, 4), (148,))* displays sample List and target classification name along with the new shape of our *X* and *Y* training data.

We extract our third test sample in a similar manner.

```
versicolor_index = rnd.randrange(50, 99)
test_versicolor = [iris.data[versicolor_index, :]]
X = np.delete(X, versicolor_index, 0)
Y = np.delete(Y, versicolor_index, 0)
test_versicolor, iris.target_names[iris.target[versicolor_index]], X.s\
hape, Y.shape
```

Now all we need to do is train our model with the training dataset we prepared earlier, or the remaining 147 samples after extracting the test samples.

```
model_tree = tree.DecisionTreeClassifier()
model_tree.fit(X, Y)
```

Once our model is trained, we will use it to predict solution to our question. Given a set of features, to which species or classification does this flower belong.

```
pred_tree_setosa = model_tree.predict(test_setosa)
print('Decision Tree predicts {} for test_setosa'
    .format(iris.target_names[pred_tree_setosa]))
```

The output of this code *Decision Tree predicts ['setosa'] for test_setosa* accurately predicts the flower species.

```
pred_tree_virginica = model_tree.predict(test_virginica)
print('Decision Tree predicts {} for test_virginica'
    .format(iris.target_names[pred_tree_virginica]))
```

The output of this code *Decision Tree predicts [‘virginica’] for test_virginica* accurately predicts the flower species.

```
pred_tree_versicolor = model_tree.predict(test_versicolor)
print('Decision Tree predicts {} for test_versicolor'
    .format(iris.target_names[pred_tree_versicolor]))
```

We expect similar results for our third specie prediction.

An important point to note is that our test samples were excluded from the training samples. The machine learning model was able to predict classification for the new samples with 100% accuracy based on prior training. Your laptop is now a relatively intelligent Botanist!

You can run the project several times, every time the test samples are randomly extracted from the training dataset and prediction results are checked.

Well done. You just completed an end-to-end data science workflow complete with machine learning model training and prediction, data wrangling and visualization, acquisition and analysis. All within 20 lines of Python logic. Data science and machine learning is easy and fun!

Neo4j Graph Database

Graph databases are the first class citizens of the data science and machine learning world. Google's PageRank and Knowledge Graph are Graph databases. So is Facebook's Social/Open Graph. LinkedIn and PayPal also use graph databases for their respective use cases. Some other use cases for Graph databases include fraud detection, supply chain management, traffic management, realtime recommendation engines, marketing & web analytics, IT & network operations, master data management, social networks, and identity & access management.

In this chapter we implement the specifications for a data science catalog using one of world's leading graph databases Neo4j.

Coding Goals

In this chapter we will learn to code in Neo4j Cypher Graph Query Language to start building the Data Science Graph.

You can code along instructions in this chapter. The code, import data, and Data Science Graph database in Neo4j is available for download at a dedicated GitHub repository.

```
https://github.com/Startupsci/data-science-graph-code
```

You can clone this database in a local folder and start your Neo4j community server and Browser on this folder. Login is *neo4j* and password is *demo* for this database.

Learning Goals

Learning goals for this chapter will cover basics of setting up a Neo4j Graph Database. We will learn how to create, update, read, and delete data from our Graph database.

- Learn about benefits of graph databases.
- Setup Neo4j community edition Server, Browser, and Editor.
- Learn about Cypher Graph Query Language.
- Import CSV to graph database.
- Create nodes, relationships, constraints, and indexes.
- Transform from relational model to graph model.
- Profile and performance optimize your graph queries and model.
- Query your graph for statistics.

Benefits of graph databases

Network. Our brain thinks by making connections between known concepts and new ones. This is how the popular idea communication method of mind mapping evolved. Relational databases store information in tables, rows, and columns. Although they are called relational, ironically it is the entity to entity relationships that become harder to decipher in a larger database. Hierarchical databases impose fixed parent-child relationships based on hierarchy of documents, offering complex solutions like MapReduce to query on multi-dimensional relationships that exist in the real world. Graph databases are designed using nodes or concepts connected with each other using directed relationships, just like mind maps. They are easier to visualize and faster to query.

Refactoring. Refactoring relational database schema becomes more difficult over time without breaking existing code that may rely on prior structure. Hierarchical, NoSQL, or schema-less databases became popular for this reason, however querying such databases on multiple relationships among existing entities is infamously slow and complex to develop. Graph databases combine the best of both the worlds. Relationships are first-class citizens and structure is schema-less. This makes refactoring a breeze no matter how large or complex the database becomes.

Joins vs Relationships. Many relational database queries which return valuable insights about our data, require a join or a set of joins among multiple entities. Join queries can get really complex as number of entities and fields grow. The performance of relational joins degrades exponentially with number of entities and fields involved. In a graph database relationships are first-class citizens. Querying on these relationships is relatively much easier. The performance of multi-relationship queries is linear.

Scalability. Relational databases traditionally scale vertically (much more expensively), adding more compute and storage to existing servers. Hierarchical databases scale horizontally over server farms or the Cloud more easily, however parallel computation is a challenge. Graph databases can scale horizontally like hierarchical database without compromising on performance, easily enabling parallel computation.

Install Neo4j community edition

You can download Neo4j community edition from their official website.

```
https://neo4j.com/download/community-edition/
```

Community edition is well suited for learning about Graph databases and prototyping your first Graph apps. The edition provides graph model database, native graph processing and storage, ACID transactions, Cypher Graph Query Language, language drivers for popular languages like Python, REST API, high performance native API, and HTTPS support via plugin.

Once you install Neo4j you can access it from *Applications* folder. Running it will open a dialog box requesting location for your Neo4j database. Choose any empty folder or the default folder suggested. Once the database server starts the dialog box provides a link to your local database server browser at http://localhost:7474/browser/.

> When you select an empty folder, Neo4j automatically creates an empty database within this folder. The total footprint of an empty Graph database is around 200KB, which is relatively much smaller than most databases. You can truly operate Neo4j starting from your Laptop and scale up to Cloud when required.

Neo4j Browser is to Graph Databases what Jupyter Notebook is to Python. It is an interactive web based tool which enables you to explore your Graph database. It also enables you to develop your database using the Cypher Graph Query Language.

Atom editor support for Cypher

While you will only need the Neo4j Browser for managing your database, you may benefit from saving your frequently used Cypher queries in files and editing these using a code editor as needed. Here is how to use the Atom editor with Cypher.

Save Cypher script files with *.cql* or *.cypher* extension.

```
apm install language-cypher
```

Install Atom package for color coding Cypher statements.

Cypher Graph Query Language

Cypher is the declarative query language for Neo4j, which means it is natural language like and defines *what* operation you are performing (declarative) instead of *how* to perform the operation (imperative).

Node is a circle. In Cypher, using closest text symbol of circle, round brackets are used to represent a node. Like (n) is used to represent the node n.

Think of a node as representing a sample.

Labels are types or roles. Labels are prefixed with a colon and a variable, like *n:LabelName* indicating they describe a variable's role in the graph.

Think of a label as representing a set of samples forming a single training or testing dataset.

Properties are name:value pairs. Properties are described like in JSON as name:value pairs enclosed within parenthesis. So *(n:Node {property:'value'})* defines a string property for node n. Nodes can contain zero to many properties.

Think of a property as representing a feature within a sample.

Relationships are lines and arrows. Relationships are depicted in Cypher as dashes with or without less-than or greater-than arrow indicating directed relationships. So *m:Movie-[:CAST]->a:Actor* depicts a directed relationship from a Movie to an Actor via relationship of type CAST. You can also create variables for relationships *[c:CAST]* which you can refer in your multi-part query. Similar to nodes, relationships can also contain zero to many properties. So *[:CAST {lead: true}]* defines a relationship with a boolean property called *lead* setting *true* as its value.

Think of relationships as categorical features.

You can reference the cheat sheet on Cypher commands and syntax.

```
https://neo4j.com/docs/cypher-refcard/current/
```

Cypher naming conventions

Labels. Use *CamelCase* starting with a capital. Labels are nouns, so name these accordingly. You can have more than one labels describing a node. A node can play many roles, have many types.

Relationships. Use *:SNAKE_CAPS* start with a colon, all caps relationship names with underscore for multiple words. Relationships are verbs, so name these accordingly. Prefer single word relationship names.

Variables. When using multiple CREATE and relationship definition statements together, name the variables in *lowerCamelCase* convention, using short version of title or name property value. So if the name property value for your Movie node is *The Matrix*, then a variable *theMatrix* can uniquely identify this movie which you can refer when creating relationships.

When you are adding a single node at a time it is ok to use generic variable names like *m* or *movie*, shortening or replicating the node label *Movie*.

When querying your Graph database, name the variables based on use case. So if you are searching for movies released this month, you could name your variable as *latestReleases* representing your use case.

Properties. We like staying consistent. So we follow same *lowerCamelCase* naming convention for properties as variables. Properties for nodes are adjectives while properties for relationships are adverbs, so name these accordingly.

Neo4j browser commands

:help Help on popular commands. Suffix a command to get help specific the the command, like *:help clear* will get you a result frame displaying help on the *:clear* command.

:clear Clear all the result frames from the Neo4j Browser.

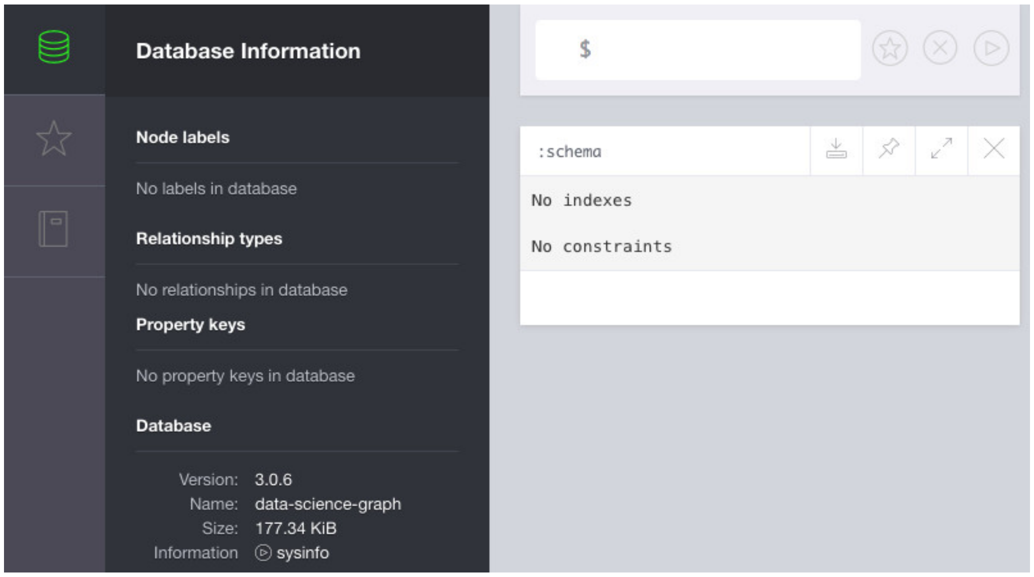
:server change-password Change existing database server password. No server restart required.

:schema Lists indexes and constraints created within a Neo4j database.

Creating Data Science Graph

Now we are finally ready to create the Data Science Graph. We searched, we scraped, we wrangled, now let us graph!

With the Neo4j Browser open, let us type *:schema* command in the Editor. The result Frame will display *No indexes* and *No constraints* result.



Node labels
No labels in database
Relationship types
No relationships in database
Property keys
No property keys in database
Database
Version: 3.0.6
Name: data-science-graph
Size: 177.34 KiB
Information sysinfo

```
:schema
No indexes
No constraints
```

This is a blank Graph Database staring back at us with a lot of potential. Let us start populating the graph.

Preparing data for import

Our CSV files are almost ready for import. Following the naming conventions rename the *name* feature in the three files as *datasetName*, *datasourceName*, and *categoryName* for enabling Neo4j import to automatically pick up the node text to display in the visual graph. The *Category_id* feature can also be renamed to *categoryName* following the conventions. We can also drop the *cloud* feature from our *datasources.csv* as this property only makes sense for datasets.

Copy the three CSV files into the folder containing your Neo4j graph database under a new folder called *import*.

Constraints and indexes

Before we import the data let us setup some uniqueness constraints and indexes. We expect the *datasetName*, *datasourceName*, and *categoryName* features in their respective files to be unique as we ensured during the wrangling stage of our workflow. Creating a uniqueness constraint on these properties in the graph database also automatically creates indexes.

We add the constraints playing the following statements in the Editor.

```
CREATE CONSTRAINT ON (d:Dataset)
ASSERT d.datasetName IS UNIQUE;
```

The result frame will display *Added 1 constraint, statement executed in 132 ms.* result.

Next statement adds constraint on nodes with the *Datasource* label.

```
CREATE CONSTRAINT ON (ds:Datasource)
ASSERT ds.datasourceName IS UNIQUE;
```

Final statement adds constraint on nodes with the *Category* label.

```
CREATE CONSTRAINT ON (c:Category)
ASSERT c.categoryName IS UNIQUE;
```

Typing the *:schema* command in the Editor now displays following result frame.

```
Indexes
  ON :Category(categoryName)     ONLINE (for uniqueness constraint)
  ON :Dataset(datasetName)       ONLINE (for uniqueness constraint)
  ON :Datasource(datasourceName) ONLINE (for uniqueness constraint)

Constraints
  ON (dataset:Dataset) ASSERT dataset.datasetName IS UNIQUE
  ON (datasource:Datasource) ASSERT datasource.datasourceName IS UNIQUE
  ON (category:Category) ASSERT category.categoryName IS UNIQUE
```

Import CSV samples as nodes

Now all we need to do is import the data in a new Neo4j graph database. Our data import will be checked against the constraints we just setup.

Let us start by importing the *categories.csv* file as nodes with *Category* label.

```
LOAD CSV WITH HEADERS FROM "file:///categories.csv" AS sample
CREATE (c:Category)
SET c = sample;
```

Using the *LOAD CSV* clause the statement loads the CSV samples and iterates over each sample with the *AS* clause. Using the *CREATE* clause the statement then creates a node with label *Category* for each iteration. Using *SET* clause the statement sets each node's properties to sample features.

The result frame will display *Added 31 labels, created 31 nodes, set 31 properties…* message.

Next we import the *datasources.csv* file setting up the samples as nodes with *Datasource* label.

```
LOAD CSV WITH HEADERS FROM "file:///datasources.csv" AS sample
CREATE (ds:Datasource)
SET ds = sample;
```

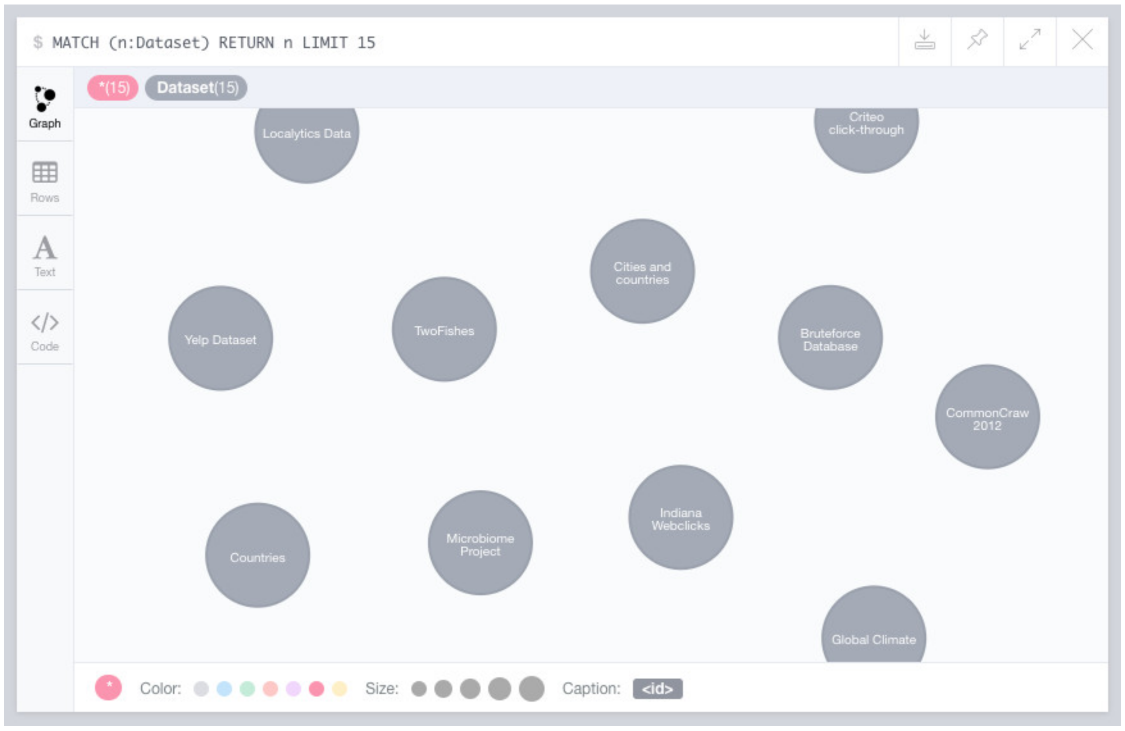
The result *Added 332 labels, created 332 nodes, set 1935 properties…* indicates number of rows converted to nodes, and number of features x samples converted to properties.

```
LOAD CSV WITH HEADERS FROM "file:///datasets.csv" AS sample
CREATE (d:Dataset)
SET d = sample;
```

The result *Added 61 labels, created 61 nodes, set 366 properties…* confirms creation of our dataset nodes.

At this stage we can visualize our graph database.

```
MATCH (n:Dataset) RETURN n LIMIT 25;
```



```
$ MATCH (n:Dataset) RETURN n LIMIT 15
```

Note that we can change the size of the node circle by clicking on *(15)* label on top left of the result frame and then choosing Size bubble at the bottom. This new selected size will now default for all the subsequent result frames.

Clicking on any of the node bubbles will open a property sheet at the bottom of the result frame. You can also click on the left-arrow icon at bottom right of the property sheet to expand if there are more properties to display.

Clicking on any of the node bubbles will also open a circular context menu with three icons for options. One marked x removes the node from the visualization. The lock icon unlocks the node the relayout the graph visualization. The cross-arrows icon opens connected graph or related nodes.

Create relationships

So far our graph database mimics the tabular or relational data our CSVs represent. Let us start changing our data model from relational to graph oriented step-by-step. Our first step is to create relationships among Category, Dataset and Datasource nodes. These relationships already exist as *categoryName* features or columns in our CSV files.

Let us describe our Dataset-Category relationship as an English statement.

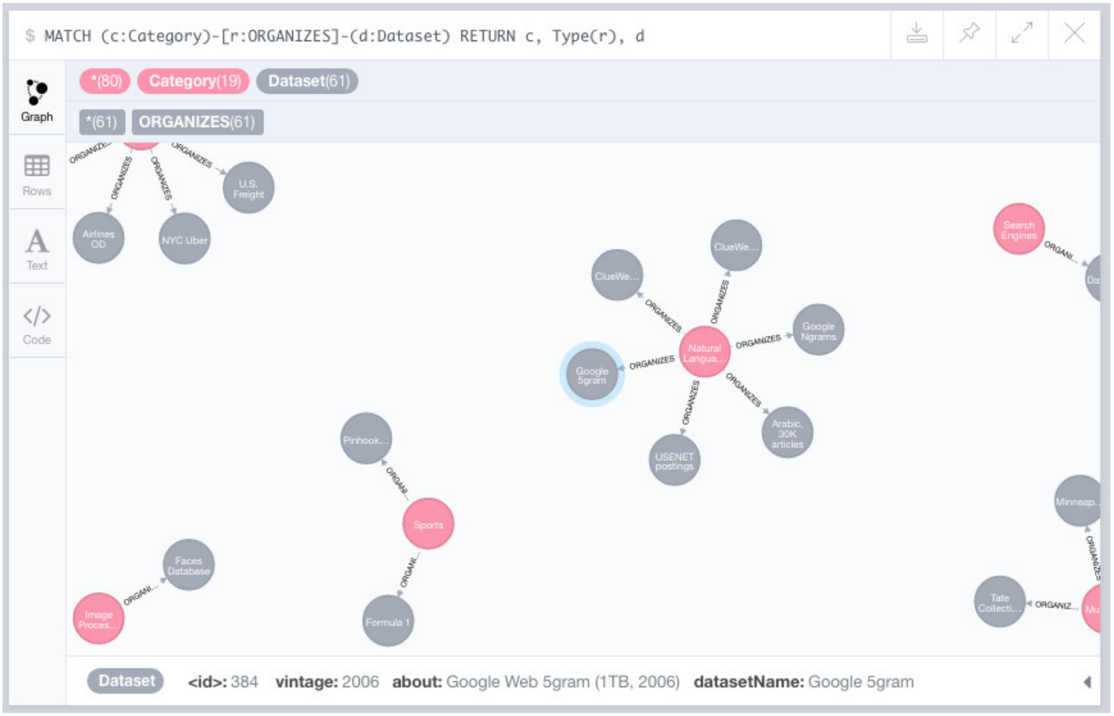
> A Category (noun) ORGANIZES (verb) a Dataset (noun).

This translates into Cypher as follows.

```
MATCH (d:Dataset), (c:Category)
WHERE d.categoryName = c.categoryName
CREATE (c)-[:ORGANIZES]->(d);
```

The result frame displays *Created 61 relationships...* message. We can visualize the resulting relationships using following cypher statement.

```
MATCH (c:Category)-[r:ORGANIZES]-(d:Dataset)
RETURN c, Type(r), d
```



```
$ MATCH (c:Category)-[r:ORGANIZES]-(d:Dataset) RETURN c, Type(r), d
```

Dataset <id>: 384 vintage: 2006 about: Google Web 5gram (1TB, 2006) datasetName: Google 5gram

We can also generate some statistics on our graph using following Cypher command.

```
MATCH (c:Category)-[r:ORGANIZES]->(d:Dataset)
RETURN c.categoryName, Type(r), Count(d) as datasets
ORDER BY datasets DESC;
```

This will display result frame in *Rows* mode with tabular listing of sorted count of datasets by categories.

Next we will establish the relationships for Datasource nodes.

```
MATCH (ds:Datasource), (c:Category)
WHERE ds.categoryName = c.categoryName
CREATE (c)-[:ORGANIZES]->(ds);
```

The result frame displays *Created 332 relationships...* message.

We can now generate statistics for Dataset, Datasource, and Category nodes.

```
MATCH (c:Category)-[r:ORGANIZES]->(n)
RETURN c.categoryName, Type(r), Count(n) as nodes
ORDER BY nodes DESC
```

Now that we have represented our ORGANIZES relationships, we no longer need the *categoryName* properties within Dataset and Datasource nodes. We can remove these properties with following commands issued one at a time in the Editor.

```
MATCH (ds:Datasource)
REMOVE ds.categoryName;

MATCH (d:Dataset)
REMOVE d.categoryName;
```

The result frame will display *Set xxx properties...* message.

Profiling graph queries

Let us continue to optimize our database from relational model to the graph model. In this section we will learn about profiling our graph database for performance. More specifically profiling queries on our graph database. As you may have realized by now, the nature of queries you can code directly depends on how you organize the graph database structure. As you advance your understanding of graph models, you will learn to structure your graph based on nature of queries required by your problem domain.

For now let us learn the easier way, bottom up, from our still relational-like graph model.

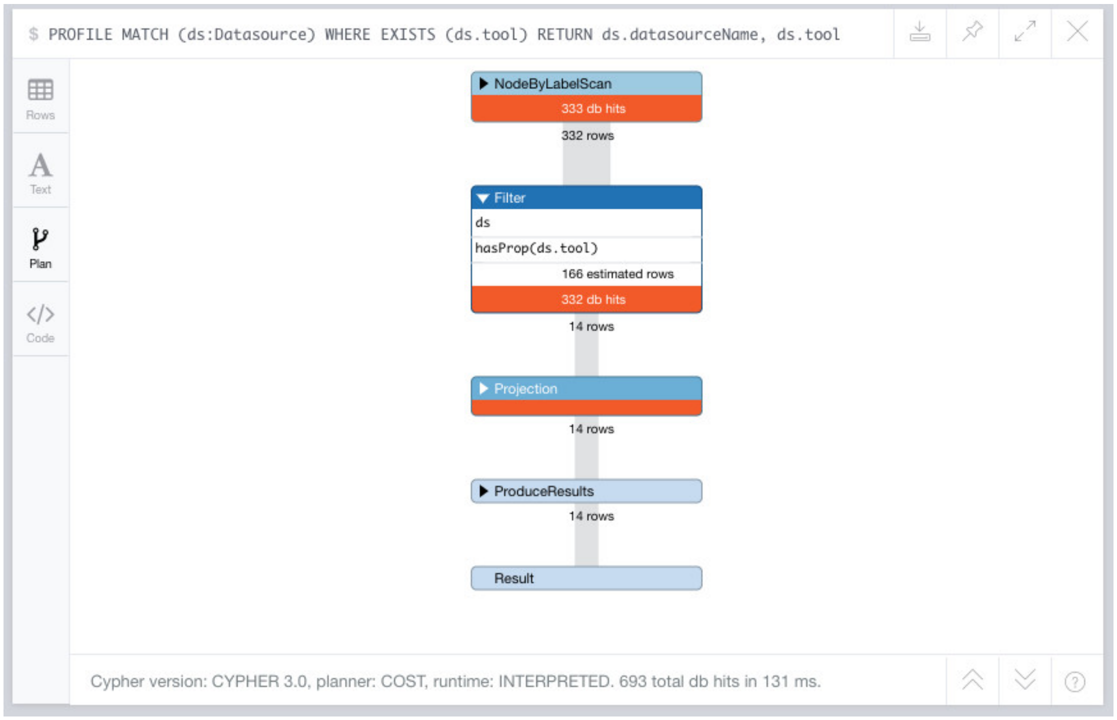
During our data wrangling workflow we created a *tool* feature which specifies if the given data source is a specialized tool. Let us review the values assigned as we set this feature to a node property.

```
MATCH (ds:Datasource)
WHERE EXISTS (ds.tool)
RETURN ds.datasourceName, ds.tool;
```

Storing this categorical feature as a property is not optimal. Only some nodes have this property set however in most queries all nodes will be checked.

To get a sense of query performance we can prefix the any Cypher statement with the *PROFILE* clause. This will not execute the query. Instead it displays profiling results simulating query execution. Based on number database hits and other parameters, the PROFILE clause returns the estimated query run time, and a visualization of each component of the query.

Profiling the above mentioned query to check *tool* property, we note the cost as 693 total db hits in 131 ms.


$ PROFILE MATCH (ds:Datasource) WHERE EXISTS (ds.tool) RETURN ds.datasourceName, ds.tool

Cypher version: CYPHER 3.0, planner: COST, runtime: INTERPRETED. 693 total db hits in 131 ms.

Now let us assume we converted the *tool* property to a label. Let us profile a query to list all labels for Datasource. This will approximate what will happen if we were to convert the property to a label and query for similar results.

```
PROFILE MATCH (ds:Datasource)
RETURN ds.datasourceName, Labels(ds);
```

The profiling result is amazing. While the database hits have gone up to 1,329 hits (2X prior query), the execution time is gone down to just 31 ms (3X savings). This is because labels are automatically indexed by Neo4j and queries perform best when using labels.

Now that we have simulated what will happen when we change from a property to a label, we can safely perform the refactoring.

First we set the new label based on the property value. Run these statements one by one in the Editor.

```
MATCH (ds:Datasource)
WHERE ds.tool="Data Catalog"
SET ds:DataCatalog;

MATCH (ds:Datasource)
WHERE ds.tool="Data API"
SET ds:DataAPI;

MATCH (ds:Datasource)
WHERE ds.tool="Data Explorer"
SET ds:DataExplorer;

MATCH (ds:Datasource)
WHERE ds.tool="Database Search"
SET ds:DatabaseSearch;
```

When the statements execute, the result frame will display *Added xx labels...* based on number of nodes impacted.

Now we remove the redundant property.

```
MATCH (ds:Datasource) REMOVE ds.tool;
```

This results in *Set 14 properties...* message.

Now let us profile the new query to list all datasources which have the tools labels.

```
PROFILE MATCH (ds:Datasource)
WHERE (ds:DataAPI) OR (ds:DataExplorer)
OR (ds:DataCatalog) OR (ds:DatabaseSearch)
RETURN ds.datasourceName, Labels(ds);
```

While the query db hits increase to 1,696, the run time reduces to 48ms. We can do even better. Let us add another label called *Tool* for all the nodes which have tool labels.

```
MATCH (ds:Datasource)
WHERE (ds:DataAPI) OR (ds:DataExplorer)
OR (ds:DataCatalog) OR (ds:DatabaseSearch)
SET ds:Tool;
```

Now we have a much shorter query to display the same results as our first query. This brevity is a good indication of optimized query and graph model.

```
MATCH (ds:Tool)
RETURN ds.datasourceName, Labels(ds);
```

Profiling this query results in drastic improvements of only 85 database hits and 48ms run time.

This way iteratively profiling queries is a good strategy for refactoring our graph model.

Create a new node

Now that we have imported our graph from CSV, setup relationships, refactored properties to labels, we should start using our graph to create new data directly within the Neo4j Browser.

Let us create a new datasource for GitHub. Let us start by searching for the new datasource by link to ensure it does not exist in the database.

```
MATCH (ds:Datasource)
WHERE ds.link CONTAINS 'github.com'
RETURN ds.datasourceName, ds.link;
```

The result returns with zero rows. We can now safely create a new datasource.

```
CREATE (ds:Datasource:DataRepo:Tool
  { datasourceName: 'GitHub',
  about: 'GitHub Repository',
  link: 'https://github.com/'});
```

Result frame displays *Added 3 labels, created 1 node, set 3 properties...* message.

Now let us add a new Category to organize datasources like GitHub.

```
CREATE (c:Category {categoryName: 'Mixed'});
```

Then we can create a relationship between the new category and the node we just created.

```
MATCH (c:Category {categoryName: 'Mixed'}), (ds:DataRepo)
CREATE (c)-[:ORGANIZES]->(ds);
```

Transform property into a relationship

We have a property in Dataset nodes called *cloud* which indicates if the dataset is stored on a Cloud provider. We can list the datasets which have this property specified using the following query.

```
MATCH (d:Dataset)
WHERE EXISTS (d.cloud)
RETURN d.datasetName, d.cloud;
```

There are several issues with this property.

- The property only exists for a few nodes among the label Dataset.
- The property is redundant and is already represented as Datasource nodes.
- It is costly to query on this property.

Following the guidelines from profiling section earlier, we realize we can convert this property into a relationship, thereby optimizing our queries and database model. The relationship can be described in English simply as follows.

> Datasource STORES a Dataset.

```
MATCH (d:Dataset), (ds:Datasource)
WHERE ds.datasourceName=d.cloud
CREATE (ds)-[:STORES]->(d);
```

This statement results in *Created 24 relationships…* message on successful execution.

We can check the resulting graph visualization.

```
MATCH (ds:Datasource)-[]-(d:Dataset) RETURN ds, d;
```

All that remains is to remove the redundant property from our graph database.

```
MATCH (d:Dataset) REMOVE d.cloud;
```

We just transformed a property into a relationship in our graph.

Relating Awesome List

We sourced all our datasets from the Awesome Public Datasets page. It is a valid datasource and we should establish a relationship with the Datasets sourced from this datasource.

Let us create a new node to represent the Awesome Public Datasets datasource.

```
CREATE (ds:Datasource:DataList:Tool
  { datasourceName: 'Awesome Datasets',
  about: 'GitHub Awesome Public Datasets',
  link: 'https://github.com/caesar0301/awesome-public-datasets'});
```

Describing the relationship we want to create in English.

> The DataList called “Awesome Datasets” LISTS all Dataset(s).

```
MATCH (d:Dataset), (ds:Datasource)
WHERE ds.datasourceName='Awesome Datasets'
CREATE (ds)-[:LISTS]->(d);
```

The result frame displays *Created 61 relationships...* message.

Now we relate the Awesome datasource to the category Mixed.

```
MATCH (c:Category {categoryName: 'Mixed'}),
(ds:Datasource {datasourceName: 'Awesome Datasets'})
CREATE (c)-[:ORGANIZES]->(ds);
```

Category relationships

Our categories can be related within themselves. The category *Natural Language* is a subset of the category *Machine Learning* from domain perspective. We can represent this relationship like so.

```
MATCH (ml:Category {categoryName: 'Machine Learning'}),
(nlp:Category {categoryName: 'Natural Language'})
CREATE (nlp)-[:SUBSET_OF]->(ml);
```

The category *Image Processing* is also a subset of *Machine Learning* which we can represent using similar Cypher.

```
MATCH (ml:Category {categoryName: 'Machine Learning'}),
(image:Category {categoryName: 'Image Processing'})
CREATE (image)-[:SUBSET_OF]->(ml);
```

Similarly for categories Neuroscience and Healthcare.

```
MATCH (neuro:Category {categoryName: 'Neuroscience'}),
(health:Category {categoryName: 'Healthcare'})
CREATE (neuro)-[:SUBSET_OF]->(health);
```

We could also create an adjacency relationship between *Healthcare* and *Biology* as these are adjacent domains.

```
MATCH (bio:Category {categoryName: 'Biology'}),
(health:Category {categoryName: 'Healthcare'})
CREATE (health)-[:ADJACENT]->(bio);
```

Graph design patterns

In this section we will review some of the design patterns and best practices we are following in this chapter.

Avoid Cartesian Product. Avoid MATCH queries which may return a Cartesian product set. A cartesian product set is a result which may have multiple permutation and combinations, each of these may be appropriate as result of the query, however the memory and performance throughput increases combinatorially and unnecessarily. For example the query, *MATCH (a:A), (b:B), (c:C)* has a Cartesian product result set equalling count of A nodes x count of B nodes x count of C nodes. So (A, B, C) and (B, C, A) are equivalent results, however only one is required. Instead use a *WHERE* clause with *OR* clause for matching among multiple labels where possible. When writing a query using Cartesian product, the Neo4j editor warns as we are using the Editor. Profiling such a query clearly shows a Cartesian product step as well.

> Please note that even the Neo4j official sample Movie Graph uses cartesian product for creating relationships between two node types as variables to both node types are needed for creating the relationship. Any way to avoid this, please create an issue in the GitHub repository for this chapter?

Categorical features as Labels. Categorical features take finite set of values. Using the values as Labels for nodes containing these features makes graph queries more intuitive and better performing.

Entities as Nodes. If the properties can be treated as an entity within your problem domain, then make them a node/label and create appropriate relationships. This is specially true for lookup properties like State codes, or Country names, which may have values shared across referencing nodes.

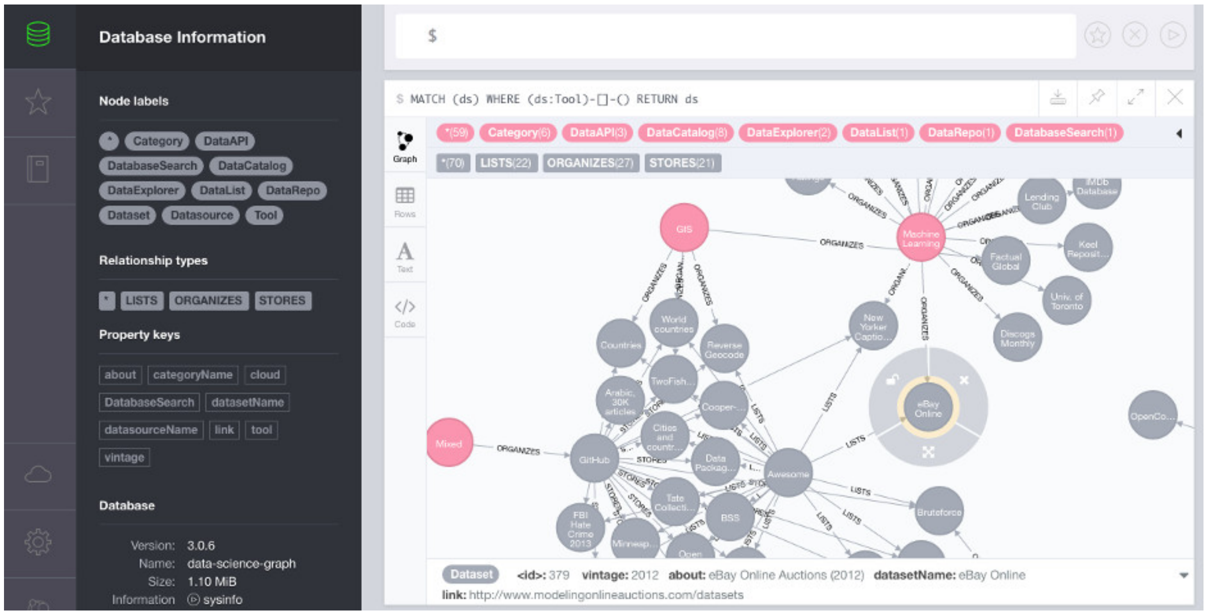
Avoid WITH clause. Cypher queries can be *piped* endlessly forming a long set of such queries using the WITH clause. This reduces readability. Introduces potential issues with variable scoping.

Index Selectively. Index properties selectively based on how these properties are used within queries. The trade-off is database read performance verses write operations required for keeping indexes up-to-date.

Avoid BLOBs. Avoid storing BLOB data as properties within graph database as this takes long time to read. Instead use specialized BLOB stores and reference URLs within graph database nodes.

Explicit Labels and Relationships. Where possible specify labels and relationship types explicitly in the query to limit the target graph that the query runs on.

Always Keep Optimizing. Use PROFILE with most of your complex queries. Remember, good short query returning a solution indicates a good graph data model design.



Database Information

Node labels

* Category DataAPI DatabaseSearch DataCatalog DataExplorer DataList DataRepo Dataset Datasource Tool

Relationship types

* LISTS ORGANIZES STORES

Property keys

about categoryName cloud DatabaseSearch datasetName datasourceName link tool vintage

Database

Version: 3.0.6
Name: data-science-graph
Size: 1.10 MiB
Information: sysinfo

```
$ MATCH (ds) WHERE (ds:Tool)-[]-() RETURN ds
```

*(59) Category(6) DataAPI(3) DataCatalog(8) DataExplorer(2) DataList(1) DataRepo(1) DatabaseSearch(1)

*(70) LISTS(22) ORGANIZES(27) STORES(21)

Dataset <id>: 379 vintage: 2012 about: eBay Online Auctions (2012) datasetName: eBay Online
link: http://www.modelingonlineauctions.com/datasets

We have come a long way so far. You just designed a data catalog which you can use for your data science projects. We started with a simple list of links on the Awesome Public Datasets page. We now have a semantic graph database with 10 labels, five relationship types, nine property keys, and more than 400 nodes. All within 1MB of database footprint. All database operations are query driven using the powerful and flexible Cypher Graph Query Language. And we are just getting started my friend!