

# Data Science for Startups

*Ben G Weber*

*2018-05-25*



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Why Data Science? . . . . .	5
1.2	Book Overview . . . . .	6
1.3	Tooling . . . . .	7
<b>2</b>	<b>Tracking Data</b>	<b>9</b>
2.1	What to Record? . . . . .	10
2.2	Tracking Specs . . . . .	12
2.3	Client vs Server Tracking . . . . .	13
2.4	Sending Tracking Events . . . . .	14
2.5	Message Encoding . . . . .	19
2.6	Building a Tracking API . . . . .	20
2.7	Privacy . . . . .	21
2.8	Conclusion . . . . .	21
<b>3</b>	<b>Data Pipelines</b>	<b>23</b>
3.1	Types of Data . . . . .	25
3.2	The Evolution of Data Pipelines . . . . .	25
3.3	A Scalable Pipeline . . . . .	31
3.4	Conclusion . . . . .	44
<b>4</b>	<b>Business Intelligence</b>	<b>47</b>
4.1	KPIs . . . . .	48
4.2	Reporting with R . . . . .	49
4.3	ETLs . . . . .	58
4.4	Reporting Tools . . . . .	61
4.5	Conclusion . . . . .	63
<b>5</b>	<b>Exploratory Data Analysis</b>	<b>67</b>
5.1	Summary Statistics . . . . .	68

5.2	Plotting . . . . .	71
5.3	Correlation Analysis . . . . .	76
5.4	Feature Importance . . . . .	78
5.5	Conclusion . . . . .	79
<b>6</b>	<b>Predictive Modeling</b>	<b>81</b>
6.1	Types of Predictive Models . . . . .	81
6.2	Training a Classification Model . . . . .	83
6.3	Clustering . . . . .	92
6.4	Conclusion . . . . .	100
<b>7</b>	<b>Productizing Models</b>	<b>103</b>
7.1	Building a Model Specification . . . . .	104
7.2	Batch Deployments . . . . .	106
7.3	Live Deployments . . . . .	115
7.4	Conclusion . . . . .	119

# Chapter 1

## Introduction

I recently changed industries and joined a startup company where I'm responsible for building up a data science discipline. While we already had a solid data pipeline in place when I joined, we didn't have processes in place for reproducible analysis, scaling up models, and performing experiments. The goal of this book is to provide an overview of how to build a data science platform from scratch for a startup, providing real examples using Google Cloud Platform (GCP) that readers can try out themselves.

This book is intended for data scientists and analysts that want to move beyond the model training stage, and build data pipelines and data products that can be impactful for an organization. However, it could also be useful for other disciplines that want a better understanding of how to work with data scientists to run experiments and build data products. It is intended for readers with programming experience, and will include code examples primarily in R and Java.

### 1.1 Why Data Science?

One of the first questions to ask when hiring a data scientist for your startup is how will data science improve our product? At Windfall Data, our product is data, and therefore the goal of data science aligns well with the goal of the company, to build the most accurate model for estimating net worth. At other organizations,

such as a mobile gaming company, the answer may not be so direct, and data science may be more useful for understanding how to run the business rather than improve products. However, in these early stages it's usually beneficial to start collecting data about customer behavior, so that you can improve products in the future.

Some of the benefits of using data science at a start up are:

- Identifying key business metrics to track and forecast
- Building predictive models of customer behavior
- Running experiments to test product changes
- Building data products that enable new product features

Many organizations get stuck on the first two or three steps, and do not utilize the full potential of data science. A goal of this book is to show how managed services can be used for small teams to move beyond data pipelines for just calculating run-the-business metrics, and transition to an organization where data science provides key input for product development.

## 1.2 Book Overview

Here are the topics I am covered in this book. Many of these chapters are based on my blog posts on Medium<sup>1</sup>.

- **Introduction:** This chapter provides motivation for using data science at a startup and provides an overview of the content covered in this book. Similar posts include functions of data science, scaling data science and my FinTech journey.
- **Tracking Events:** Discusses the motivation for capturing data from applications and web pages, proposes different methods for collecting tracking data, introduces concerns such as privacy and fraud, and presents an example with Google PubSub.
- **Data pipelines:** Presents different approaches for collecting data for use by an analytics and data science team, discusses approaches with flat files, databases, and data lakes, and presents an implementation using PubSub, DataFlow, and BigQuery. Similar posts include a scalable analytics pipeline and the evolution of game analytics platforms.

---

<sup>1</sup><https://medium.com/@bgweber>

- **Business Intelligence:** Identifies common practices for ETLs, automated reports/dashboards and calculating run-the-business metrics and KPIs. Presents an example with R Shiny and Data Studio.
- **Exploratory Analysis:** Covers common analyses used for digging into data such as building histograms and cumulative distribution functions, correlation analysis, and feature importance for linear models. Presents an example analysis with the Natality public data set. Similar posts include clustering the top 1% and 10 years of data science visualizations.
- **Predictive Modeling:** Discusses approaches for supervised and unsupervised learning, and presents churn and cross-promotion predictive models, and methods for evaluating offline model performance.
- **Model Production:** Shows how to scale up offline models to score millions of records, and discusses batch and online approaches for model deployment. Similar posts include Productizing Data Science at Twitch, and Productizing Models with DataFlow.
- **Experimentation:** Provides an introduction to A/B testing for products, discusses how to set up an experimentation framework for running experiments, and presents an example analysis with R and bootstrapping. Similar posts include A/B testing with staged rollouts.
- **Recommendation Systems:** Introduces the basics of recommendation systems and provides an example of scaling up a recommender for a production system. Similar posts include prototyping a recommender.
- **Deep Learning:** Provides a light introduction to data science problems that are best addressed with deep learning, such as flagging chat messages as offensive. Provides examples of prototyping models with the R interface to Keras, and productizing with the R interface to CloudML.

## 1.3 Tooling

Throughout the book, I'll be presenting code examples built on Google Cloud Platform. I choose this cloud option, because GCP provides a number of managed services that make it possible for small teams to build data pipelines, productize predictive models,

and utilize deep learning. It's also possible to sign up for a free trial with GCP and get \$300 in credits. This should cover most of the topics presented in this book, but it will quickly expire if your goal is to dive into deep learning on the cloud.

For programming languages, I'll be using R for scripting and Java for production, as well as SQL for working with data in BigQuery. I'll also present other tools such as R Shiny. Some experience with R and Java is recommended, since I won't be covering the basics of these languages.

This book is based on my blog series "Data Science for Startups"<sup>2</sup>. I incorporated feedback from these posts into book chapters, and authored the book using the excellent bookdown package (Xie, 2018).

---

<sup>2</sup><https://medium.com/p/80d022a18aec>



## Chapter 2

# Tracking Data

In order to make data-driven decisions at a startup, you need to collect data about how your products are being used. You also need to be able to measure the impact of making changes to your product and the efficacy of running campaigns, such as deploying a custom audience for marketing on Facebook. Again, collecting data is necessary for accomplishing these goals.

Usually data is generated directly by the product. For example, a mobile game can generate data points about launching the game, starting additional sessions, and leveling up. But data can also come from other sources, such as an email vendor that provides response data about which users read and click on links within an email. This chapter focuses on the first type of data, where tracking events are being generated by the product.

Why record data about product usage?

- **Track metrics:** You may want to record performance metrics for tracking product health or other metrics useful for running the business.
- **Enable experimentation:** To determine if changes to a product are beneficial, you need to be able to measure results.
- **Build data products:** In order to make something like a recommendation system, you need to know which items users are interacting with.

It's been said that data is the new oil, and there's a wide variety of reasons to collect data from products. When I first started in

the gaming industry, data tracked from products was referred to as telemetry. Now, data collected from products is frequently called tracking.

This chapter discusses what type of data to collect about product usage, how to send data to a server for analysis, issues when building a tracking API, and some concerns to consider when tracking user behavior.

## 2.1 What to Record?

One of the questions to answer when deploying a new product is:

- What data should we collect about user behavior?

The answer is that it depends on your product and intended use cases, but there are some general guidelines about what types of data to collect across most web, mobile, and native applications.

- **Installs:** How big is the user base?
- **Sessions:** How engaged is the user base?
- **Monetization:** How much are users spending?

For these three types of events, the data may actually be generated from three different systems. Installation data might come from a third party, such as Google Play or the App Store, a session start event will be generated from the client application, and spending money in an application, or viewing ads, may be tracked by a different server. As long as you own the service that is generating the data points, you can use the same infrastructure to collect data about different types of events.

Collecting data about how many users launch and log into a application will enable you to answer basic questions about the size of your base, and enable you to track business metrics such as DAU, MAU, ARPDAU, and D-7 retention. However, it doesn't provide much insight into what users are doing within an application, and it doesn't provide many data points that are useful for building data products. In order to better understand user engagement, it's necessary to track data points that are domain or product specific. For example, you might want to track the following types of events in a multiplayer shooter game for consoles:

- **GameStarted:** tracks when the player starts a single or multiplayer game.
- **PlayerSpawn:** tracks when the player spawns into the game world and tracks the class that the user is playing, such as combat medic.
- **PlayerDeath:** tracks where players are dying and getting stuck and enables calculating metrics such as KDR (kill/death ratio).
- **RankUp:** tracks when the player levels up or unlocks a new rank.

Most of these events translate well to other shooter games and other genres such as action/adventure. For a specific game, such as FIFA, you may want to record game specific events, such as:

- **GoalScored:** tracks when a point is scored by the player or opponent.
- **PlayerSubstitution:** tracks when a player is substituted.
- **RedCardReceived:** when the player receives a red card.

Like the prior events, many of these game-specific events can actually be generalized to sports games. If you're a company like EA with a portfolio of different sports titles, it's useful to track all of these events across all of your sports titles (the red card event can be generalized to a penalty event).

If we're able to collect these types of events about players, we can start to answer useful questions about the player base, such as:

- Are users that receive more red cards more likely to quit?
- Do online focused players play more than single-player focused players?
- Do users play the new career mode that was released?

A majority of tracking events are focused on collecting data points about released titles, but it's also possible to collect data during development. At Microsoft Studios, I worked with the user research team to get tracking in place for playtesting. As a result, we could generate visualizations that were useful for conveying to game teams where players were getting stuck. Incorporating these visualizations into the playtesting results resulted in a much better reception from game teams.

When you first add tracking to a product, you won't know of every event and attribute that will be useful to record, but you can make

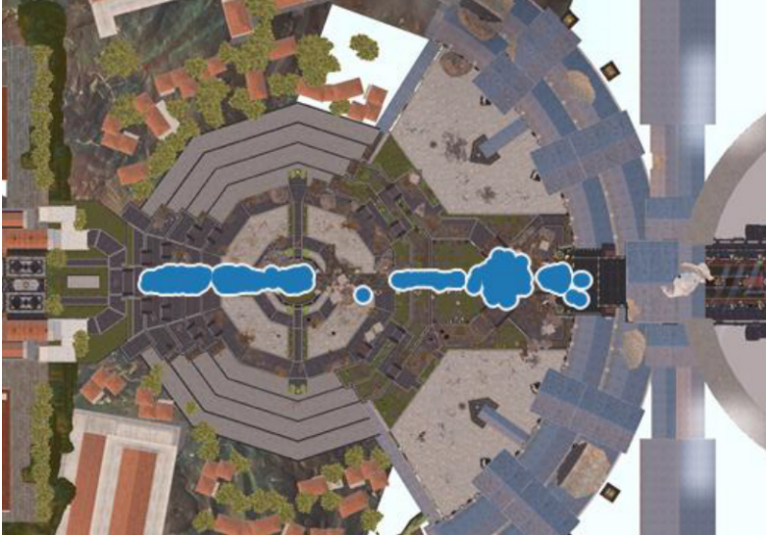


Figure 2.1: Ryse: Son of Rome Playtesting - Microsoft Studios User Research

a good guess by asking team members what types of questions they intend to ask about user behavior and by implementing events that are able to answer these questions. Even with good tracking data, you won't be able to answer every question, but if you have good coverage you can start to improve your products.

## 2.2 Tracking Specs

Some teams write tracking specifications to in order to define which tracking events need to be implemented in a product. Other teams don't have any documentation and simply take a best guess approach at determining what to record. I highly recommend writing tracking specifications as a best practice. For each event, the spec should identify the conditions for firing an event, the attributes to send, and definitions for any event-specific attributes. For example, a session start event for a web app might have the following form:

- **Condition:** fired when the user first browses to the domain. The event should not be fired when the user clicks on new

pages or uses the back button, but should fire it the user browses to a new domain and then back.

- **Properties:** web browser and version, userID, landing page, referring URL, client timestamp
- **Definitions:** referring URL should list the URL of the page that referred the user to this domain, or the application that referred the user to the web page (e.g. Facebook or Twitter).

Tracking specs are a highly useful piece of documentation. Small teams might be able to get away without having an official process for writing tracking specs, but a number of scenarios can make the documentation critical, such as implementing events on a new platform, re-implementing events for a new backend service, or having engineers leave the team. In order for specs to be useful, it's necessary to answer the following questions:

- Who is responsible for writing the spec?
- Who is responsible for implementing the spec?
- Who is responsible for testing the implementation?

In small organizations, a data scientist might be responsible for all of the aspects of tracking. For a larger organization, it's common for the owners to be a product manager, engineering team, and testing group.

## 2.3 Client vs Server Tracking

Another consideration when setting up tracking for a product is determining whether to send events from a client application or a backend service. For example, a video-streaming web site can send data about which video a user is watching directly from the web browser, or from the backend service that is serving the video. While there are pros and cons to both approaches, I prefer setting up tracking for backend services rather than client applications if possible. Some of the benefits of server-side tracking are:

- **Trusted Source:** You don't need to expose an endpoint on the web, and you know that events are being generated from your services rather than bots. This helps avoid fraud and DDoS attacks.
- **Avoid Ad Blocking:** If you send data from a client application to an endpoint exposed on the web, some users may

block access to the endpoint, which impacts business metrics.

- **Versioning:** You might need to make changes to an event. You can update your web servers as needed, but often cannot require users to update a client application.

Generating tracking from servers rather than client applications helps avoid issues around fraud, security, and versioning. However, there are some drawbacks to server-side tracking:

- **Testing:** You might need to add new events or modify existing tracking events for testing purposes. This is often easier to do by making changes on the client side.
- **Data availability:** Some of the events that you might want to track do not make calls to a web server. For example, a console game might not connect to any web services during a session start, and instead wait until a multiplayer match starts. Also, attributes such as the referring URL may only be available for the client application and not the backend service.

A general guideline is to not trust anything sent by a client application, because often endpoints are not secured and there is no way to verify that the data was generated by your application. But client data is very useful, so it's best to combine both client and server side tracking and to secure endpoints used for collecting tracking from clients.

## 2.4 Sending Tracking Events

The goal of sending data to a server is to make the data available for analysis and data products. There's a number of different approaches that can be used based on your use case. This section introduces three different ways of sending events to an endpoint on the web and saving the events to local storage. The samples below are not intended to be production code, but instead simple proofs of concept. The next chapter covers building a pipeline for processing events. Code for the samples is available on Github<sup>1</sup>.

---

<sup>1</sup><https://github.com/bgweber/StartupDataScience/>

### 2.4.1 Web Call

The easiest way to set up a tracking service is by making web calls with the event data to a web site. This can be implemented with a lightweight PHP script, which is shown in the code block below.

```
<?php
    $message = $_GET['message'];
    if ($message != '') {
        $dataFile = fopen("telemetry.log", "a");
        fwrite($dataFile, "$message\n");
        fflush($dataFile);
        fclose($dataFile);
    }
?>
```

This php script reads the message parameter from the URL and appends the message to a local file. The script can be invoked by making a web call:

```
http://.../tracking.php?message=Hello_World
```

The call can be made from client or server using the following code:

```
// endpoint
String endPoint = "http://.../tracking.php";

// send the message
String message = "Hello_World";
URL url = new URL(endPoint + "?message=" + message);
URLConnection con = url.openConnection();
BufferedReader in = new BufferedReader(new
    InputStreamReader(con.getInputStream()));

// process the response
while (in.readLine() != null) {}
in.close();
```

This is one of the easiest ways to start collecting tracking data, but it doesn't scale and it's not secure. It's useful for testing, but

should be avoided for anything customer facing. I did use this approach in the past to collect data about players for a Mario level generator experiment<sup>2</sup>.

### 2.4.2 Web Server

Another approach you can use is setting up a web service to collect tracking events. The code below shows how to use Jetty to set up a lightweight service for collecting data. In order to compile and run the example, you'll need to include the following pom file. The first step is to start a web service that will handle tracking requests:

```
public class TrackingServer extends AbstractHandler {
    public static void main(String[] args) {
        Server server = new Server(8080);
        server.setHandler(new TrackingServer());
        server.start();
        server.join();
    }

    public void handle(String target,
        Request baseRequest, HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {

        // Process Request
    }
}
```

In order to process events, the application reads the message parameter from the web request, appends the message to a local file, and then responds to the web request.

```
// append the event data to a local file
String message = baseRequest.getParameter("message");
if (message != null) {
    BufferedWriter writer = new BufferedWriter(
        new FileWriter("tracking.log", true));
```

---

<sup>2</sup><http://www.gamasutra.com/blogs/BenWeber/20131228/207819/>



```
writer.write(message + "\n");
writer.close();
}

// service the web request
response.setStatus(HttpServletResponse.SC_OK);
```

In order to call the endpoint with Java, we'll need to modify the URL:

```
URL url = new URL(
    "http://localhost:8080/?message=" + message);
```

This approach can scale a bit more than the PHP approach, but is still insecure and not the best approach for building a production system. My advice for building a production ready tracking service is to use a stream processing system such as Kafka, Amazon Kinesis, or Google's PubSub.

### 2.4.3 Subscription Service

Using messaging services such as PubSub enables systems to collect massive amounts of tracking data, and forward the data to a number of different consumers. Some systems such as Kafka require setting up and maintaining servers, while other approaches like PubSub are managed services that are serverless. Managed services are great for startups, because they reduce the amount of DevOps support needed. But the tradeoff is cost, and it's pricier to use managed services for massive data collection.

The code below shows how to use Java to post a message to a topic on PubSub. In order to run this example, you'll need to set up a free google cloud project, and enable PubSub. More details on setting up GCP and PubSub are available online<sup>3</sup>.

```
// Set up a publisher
TopicName topicName =
    TopicName.of("projectID", "raw-events");
```

---

<sup>3</sup><https://medium.com/p/4087b66952a1>

```

Publisher publisher = Publisher
    .newBuilder(topicName).build();

//schedule a message to be published
String message = "Hello World!";
PubsubMessage msg = PubsubMessage
    .newBuilder().setData(ByteString
        .copyFromUtf8(message)).build();

// publish the message, add a callback listener
ApiFuture<String> future = publisher.publish(msg);
ApiFutures.addCallback(future,
    new ApiFutureCallback<String>() {

    public void onFailure(Throwable arg0) {}
    public void onSuccess(String arg0) {}
});

publisher.shutdown();

```

This code example shows how to send a single message to PubSub for recording a tracking event. For a production system, you'll want to implement the `onFailure` method in order to deal with failed deliveries. The code above shows how to send a message with Java, while other languages are supported including Go, Python, C#, and PHP. It's also possible to interface with other stream processing systems such as Kafka.

The next code segment shows how to read a message from PubSub and append the message to a local file. In the next chapter I'll show how to consume messages using DataFlow.

```

// set up a message handler
MessageReceiver receiver = new MessageReceiver() {
    public void receiveMessage(PubsubMessage message,
        AckReplyConsumer consumer) {
        try {
            BufferedWriter writer = new BufferedWriter(new
                FileWriter("tracking.log", true));
            writer.write(
                message.getData().toStringUtf8() + "\n");

```

```
        writer.close();
        consumer.ack();
    }
    catch (Exception e) {}
};

// start the listener for 1 minute
SubscriptionName subscriptionName =
    SubscriptionName.of("projectId", "raw-events");
Subscriber subscriber = Subscriber.newBuilder(
    subscriptionName, receiver).build();

subscriber.startAsync();
Thread.sleep(60000);
subscriber.stopAsync();
```

We now have a way of getting data from client applications and backend services to a central location for analysis. The last approach shown is a scalable and secure method for collecting tracking data, and is a managed service making it a good fit for startups with small data teams.

## 2.5 Message Encoding

One of the decisions to make when sending data to an endpoint for collection is how to encode the messages being sent, since all events that are sent from an application to an endpoint need to be serialized. When sending data over the internet, it's good to avoid language specific encodings, such as Java serialization, because the application and backend services are likely implemented in different languages. There's also versioning issues that can arise when using a language-specific serialization approach.

Some common ways of encoding tracking events are using the JSON format and Google's protocol buffers. JSON has the benefit of being human readable and supported by a wide variety of languages, while buffers provide better compression and may be better suited for certain data structures. One of the benefits of using these approaches is that a schema does not need to be defined before you can send events, since metadata about the event is included in the

message. You can add new attributes as needed, and even change data types, but this may impact downstream event processing.

When getting started with building a data pipeline, I'd recommend using JSON to get started, since it's human readable and supported by a wide variety of languages. It's also good to avoid encodings such as pipe-delimited formats, because you may need to support more complex data structures, such as lists or maps, when you update your tracking events. Here's an example of what a message might look like:

```
# JSON
{"Type":"Session","Version":1.0,"UserID":"12345",
  "Platform":"iOS"}

# Pipe delimited
Session|1.0|12345|iOS
```

What about XML? No!

## 2.6 Building a Tracking API

To build a production system, you'll need to add a bit more sophistication to your tracking code. A production system should handle the following issues:

- **Delivery Failures:** if a message delivery fails, the system should retry sending the message, and have a backoff mechanism.
- **Queueing:** if the endpoint is not available, such as a phone without a signal, the tracking library should be able to store events for later transmission, such as when wifi is available.
- **Batching:** instead of sending a large number of small requests, it's often useful to send batches of tracking events.
- **Prioritization:** some messages are more important to track than others, such as preferring monetization events over click events. A tracking library should be able to prioritize more critical events.

It's also useful to have a process in place for disabling tracking events. I've seen data pipelines explode from client applications

sending way too much data, and there was no way of disabling the clients from sending the problematic event without turning off all tracking.

Ideally, a production level system should have some sort of auditing in place, in order to validate that the endpoints are receiving all of the data being sent. One approach is to send data to a different endpoint built on a different infrastructure and tracking library, but that much redundancy is usually overkill. A more lightweight approach is to add a sequential counting attribute to all events, so if a client sends 100 messages, the backend can use this attribute to know how many events the client attempted to send and validate the result.

## 2.7 Privacy

There's privacy concerns to consider when storing user data. When data is being made available to analytics and data science teams, all personally identifiable information (PII) should be stripped from events, which can include names, addresses, and phone numbers. In some instances, user names, such as a player's gamertag on Steam, may be considered PII as well. It's also good to strip IP addresses from any data being collected, to limit privacy concerns. The general recommendation is to collect as much behavioral data as needed to answer questions about product usage, while avoiding the need to collect sensitive information, such as gender and age. If you're building a product based on sensitive information, you should have strong user access controls in place to limit access to sensitive data. Policies such as GDPR are setting new regulations for collecting and processing data, and GDPR should be reviewed before shipping a product with tracking.

## 2.8 Conclusion

Tracking data enables teams to answer a variety of questions about product usage, enables teams to track the performance and health of products, and can be used to build data products. This chapter discussed some of the issues involved in collecting data about user behavior, and provided examples for how to send data from a client

application to an endpoint for later analysis. Here are the key takeaways to from this chapter:

- Use server-side tracking if possible. It helps avoid a wide variety of issues.
- QA/test your tracking events. If you're sending bad data, you may be drawing incorrect conclusions from your data.
- Have a versioning system in place. You'll need to add new events and modify existing events, and this should be a simple process.
- Use JSON for sending events. It's human readable, extensible, and supported by a wide variety of languages
- Use managed services for collecting data. You won't need to spin up servers and can collect huge amounts of data.

As you ship more products and scale up your user base, you may need to change to a different data collection platform, but this advice is a good starting point for shipping products with tracking.

## Chapter 3

# Data Pipelines

Building data pipelines is a core component of data science at a startup. In order to build data products, you need to be able to collect data points from millions of users and process the results in near real-time. While the previous chapter discussed what type of data to collect and how to send data to an endpoint, this chapter will discuss how to process data that has been collected, enabling data scientists to work with the data. The chapter on model production will discuss how to deploy models on this data platform.

Typically, the destination for a data pipeline is a data lake, such as Hadoop or parquet files on S3, or a relational database, such as Redshift. An data pipeline should have the following properties:

- **Low Event Latency:** Data scientists should be able to query recent event data in the pipeline, within minutes or seconds of the event being sent to the data collection endpoint. This is useful for testing purposes and for building data products that need to update in near real-time.
- **Scalability:** A data pipeline should be able to scale to billions of data points, and potentially trillions as a product scales. A high performing system should not only be able to store this data, but make the complete data set available for querying.
- **Interactive Querying:** A high functioning data pipeline should support both long-running batch queries and smaller interactive queries that enable data scientists to explore tables and understand the schema without having to wait min-

utes or hours when sampling data.

- **Versioning:** You should be able to make changes to your data pipeline and event definitions without bringing down the pipeline and suffering data loss. This chapter will discuss how to build a pipeline that supports different event definitions, in the case of changing an event schema.
- **Monitoring:** If an event is no longer being received, or tracking data is no longer being received for a particular region, then the data pipeline should generate alerts through tools such as PagerDuty.
- **Testing:** You should be able to test your data pipeline with test events that do not end up in your data lake or database, but that do test components in the pipeline.

There's a number of other useful properties that a data pipeline should have, but this is a good starting point for a startup. As you start to build additional components that depend on your data pipeline, you'll want to set up tooling for fault tolerance and automating tasks.

This chapter will show how to set up a scalable data pipeline that sends tracking data to a data lake, database, and subscription service for use in data products. I'll discuss the different types of data in a pipeline, the evolution of data pipelines, and walk through an example pipeline implemented on GCP.

Before deploying a data pipeline, you'll want to answer the following questions, which resemble our questions about tracking specs:

- Who owns the data pipeline?
- Which teams will be consuming data?
- Who will QA the pipeline?

In a small organization, a data scientist may be responsible for the pipeline, while larger organizations usually have an infrastructure team that is responsible for keeping the pipeline operational. It's also useful to know which teams will be consuming the data, so that you can stream data to the appropriate teams. For example, marketing may need real-time data of landing page visits to perform attribution for marketing campaigns. And finally, the data quality of events passed to the pipeline should be thoroughly inspected on a regular basis. Sometimes a product update will cause a tracking event to drop relevant data, and a process should be set up to capture these types of changes in data.



## 3.1 Types of Data

Data in a pipeline is often referred to by different names based on the amount of modification that has been performed. Data is typically classified with the following labels:

- **Raw:** Is tracking data with no processing applied. This is data stored in the message encoding format used to send tracking events, such as JSON. Raw data does not yet have a schema applied. It's common to send all tracking events as raw events, because all events can be sent to a single endpoint and schemas can be applied later on in the pipeline.
- **Processed:** Processed data is raw data that has been decoded into event specific formats, with a schema applied. For example, JSON tracking events that have been translated into a session start events with a fixed schema are considered processed data. Processed events are usually stored in different event tables/destinations in a data pipeline.
- **Cooked:** Processed data that has been aggregated or summarized is referred to as cooked data. For example, processed data could include session start and session end events, and be used as input to cooked data that summarizes daily activity for a user, such as number of sessions and total time on site for a web page.

Data scientists will typically work with processed data, and use tools to create cooked data for other teams. This chapter discusses how to build a data pipeline that outputs processed data, while the Business Intelligence chapter will discuss how to add cooked data to your pipeline.

## 3.2 The Evolution of Data Pipelines

Over the past two decades the landscape for collecting and analyzing data has changed significantly. Rather than storing data locally via log files, modern systems can track activity and apply machine learning in near real-time. Startups might want to use one of the earlier approaches for initial testing, but should really look to more recent approaches for building data pipelines. Based on my experience, I've noticed four different approaches to pipelines:

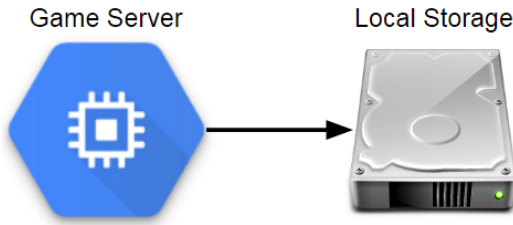


Figure 3.1: Components in a pre-database Analytics Architecture

- **Flat File Era:** Data is saved locally on game servers
- **Database Era:** Data is staged in flat files and then loaded into a database
- **Data Lake Era:** Data is stored in Hadoop/S3 and then loaded into a DB
- **Serverless Era:** Managed services are used for storage and querying

Each of the steps in this evolution support the collection of larger data sets, but may introduce additional operational complexity. For a startup, the goal is to be able to scale data collection without scaling operational resources, and the progression to managed services provides a nice solution for growth.

The data pipeline that we'll walk through in the next section of this chapter is based on the most recent era of data pipelines, but it's useful to walk through different approaches because the requirements for different companies may fit better with different architectures.

### 3.2.1 Flat File Era

I got started in data science at Electronic Arts in 2010, before EA had an organization built around data. While many game companies were already collecting massive amounts of data about gameplay, most telemetry was stored in the form of log files or other flat file formats that were stored locally on the game servers. Nothing could be queried directly, and calculating basic metrics such as monthly active users (MAU) took substantial effort.

At Electronic Arts, a replay feature was built into Madden NFL 11 which provided an unexpected source of game telemetry. After

every game, a game summary in an XML format was sent to a game server that listed each play called, moves taken during the play, and the result of the down. This resulted in millions of files that could be analyzed to learn more about how players interacted with Madden football in the wild.

Storing data locally is by far the easiest approach to take when collecting gameplay data. For example, the PHP approach presented in the last chapter is useful for setting up a lightweight analytics endpoint. But this approach does have significant drawbacks.

This approach is simple and enables teams to save data in whatever format is needed, but has no fault tolerance, does not store data in a central location, has significant latency in data availability, and has standard tooling for building an ecosystem for analysis. Flat files can work fine if you only have a few servers, but it's not really an analytics pipeline unless you move the files to a central location. You can write scripts to pull data from log servers to a central location, but it's not generally a scalable approach.

### 3.2.2 Database Era

While I was at Sony Online Entertainment, we had game servers save event files to a central file server every couple of minutes. The file server then ran an ETL process about once an hour that fast loaded these event files into our analytics database, which was Vertica at the time. This process had a reasonable latency, about one hour from a game client sending an event to the data being queryable in our analytics database. It also scaled to a large volume of data, but required using a fixed schema for event data.

When I was at Twitch, we used a similar process for one of our analytics databases. The main difference from the approach at SOE was that instead of having game servers scp files to a central location, we used Amazon Kinesis to stream events from servers to a staging area on S3. We then used an ETL process to fast load data into Redshift for analysis. Since then, Twitch has shifted to a data lake approach, in order to scale to a larger volume of data and to provide more options for querying the datasets.

The databases used at SOE and Twitch were immensely valuable for both of the companies, but we did run into challenges as we scaled the amount of data stored. As we collected more detailed

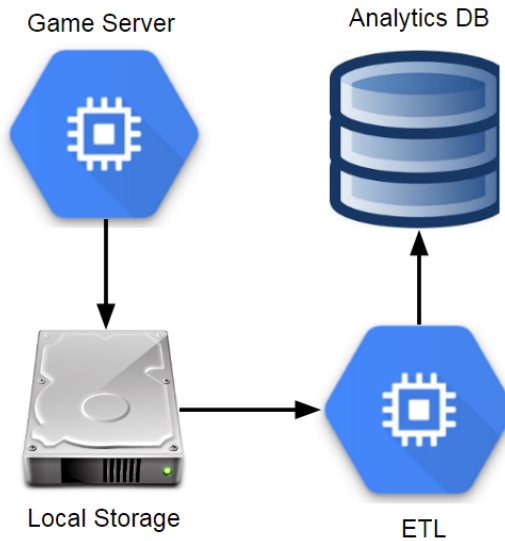


Figure 3.2: Components in an ETL-based Analytics Architecture

information about gameplay, we could no longer keep complete event history in our tables and needed to truncate data older than a few months. This is fine if you can set up summary tables that maintain the most important details about these events, but it's not an ideal situation.

One of the issues with this approach is that the staging server becomes a central point of failure. It's also possible for bottlenecks to arise where one game sends way too many events, causing events to be dropped across all of the titles. Another issue is query performance as you scale up the number of analysts working with the database. A team of a few analysts working with a few months of gameplay data may work fine, but after collecting years of data and growing the number of analysts, query performance can be a significant problem, causing some queries to take hours to complete.

The main benefits of this approach are that all event data is available in a single location queryable with SQL and great tooling is available, such as Tableau and DataGrip, for working with relational databases. The drawbacks are that it's expensive to keep all data loaded into a database like Vertica or Redshift, events need to have a fixed schema, and truncating tables may be necessary.

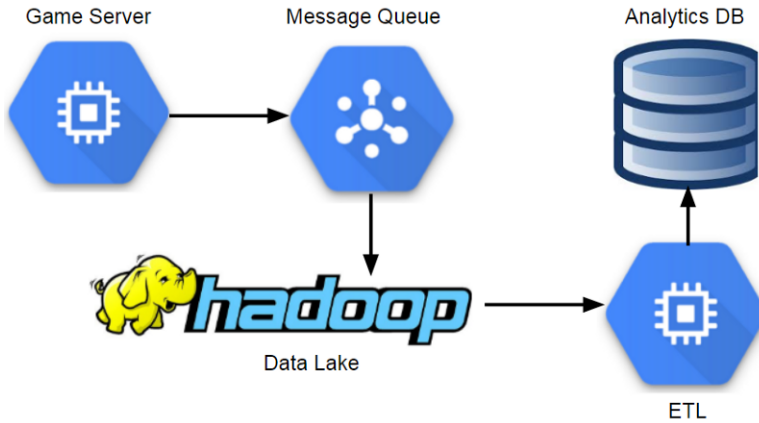


Figure 3.3: Components in a Data Lake Analytics Architecture

Another issue with using a database as the main interface for data is that machine learning tools such as Spark's MLlib cannot be used effectively, since the relevant data needs to be unloaded from the database before it can be operated on. One of the ways of overcoming this limitation is to store gameplay data in a format and storage layer that works well with Big Data tools, such as saving events as Parquet files on S3. This type of configuration became more population in the next era, and gets around the limitations of needing to truncate tables and the reduces the cost of keeping all data.

### 3.2.3 Data Lake Era

The data storage pattern that was most common while I was working as a data scientist in the games industry was a data lake. The general pattern is to store semi-structured data in a distributed database, and run ETL processes to extract the most relevant data to analytics databases. A number of different tools can be used for the distributed database: at Electronic Arts we used Hadoop, at Microsoft Studios we used Cosmos, and at Twitch we used S3.

This approach enables teams to scale to massive volumes of data, and provides additional fault tolerance. The main downside is that it introduces additional complexity, and can result in analysts having access to less data than if a traditional database approach was

used, due to lack of tooling or access policies. Most analysts will interact with data in the same way in this model, using an analytics database populated from data lake ETLs.

One of the benefits of this approach is that it supports a variety of different event schemas, and you can change the attributes of an event without impacting the analytics database. Another advantage is that analytics teams can use tools such as Spark SQL to work with the data lake directly. However, most places I worked at restricted access to the data lake, eliminating many of the benefits of this model.

This approach scales to a massive amount of data, supports flexible event schemas, and provides a good solution for long-running batch queries. The down sides are that it may involve significant operational overhead, may introduce large event latencies, and may lack mature tooling for the end users of the data lake. An additional drawback with this approach is that usually a whole team is needed just to keep the system operational. This makes sense for large organizations, but may be overkill for smaller companies. One of the ways of taking advantage of using a data lake without the cost of operational overhead is by using managed services.

### 3.2.4 Serverless Era

In the current era, analytics platforms incorporate a number of managed services, which enable teams to work with data in near real-time, scale up systems as necessary, and reduce the overhead of maintaining servers. I never experienced this era while I was working in the game industry, but saw signs of this transition happening. Riot Games is using Spark<sup>1</sup> for ETL processes and machine learning, and needed to spin up infrastructure on demand. Some game teams are using elastic computing methods for game services, and it makes sense to utilize this approach for analytics as well.

This approach has many of the same benefits as using a data lake, autoscales based on query and storage needs, and has minimal operational overhead. The main drawbacks are that managed services can be expensive, and taking this approach will likely result in using platform specific tools that are not portable to other cloud providers.

---

<sup>1</sup><https://databricks.com/customers/riot-games>

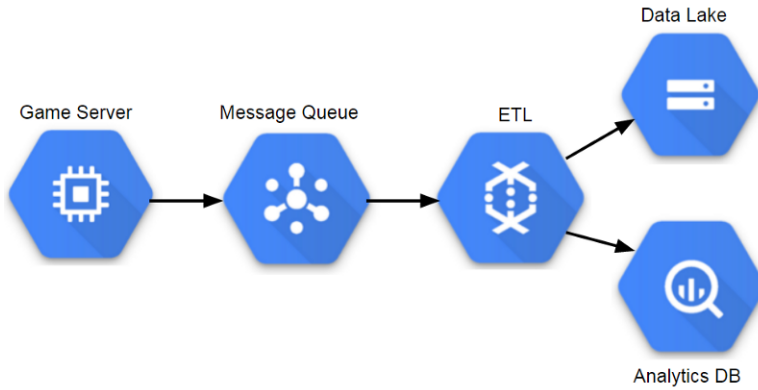


Figure 3.4: Components in a managed Analytics Architecture

In my career I had the most success working with the database era approach, since it provided the analytics team with access to all of the relevant data. However, it wasn't a setup that would continue to scale and most teams that I worked on have since moved to data lake environments. In order for a data lake environment to be successful, analytics teams need access to the underlying data, and mature tooling to support their processes. For a startup, the serverless approach is usually the best way to start building a data pipeline, because it can scale to match demand and requires minimal staff to maintain the data pipeline. The next section will walk through building a sample pipeline with managed services.

### 3.3 A Scalable Pipeline

We'll build a data pipeline that receives events using Google's PubSub as an endpoint, and save the events to a data lake and database. The approach presented here will save the events as raw data, but I'll also discuss ways of transforming the events into processed data.

The data pipeline that performs all of this functionality is relatively simple. The pipeline reads messages from PubSub and then transforms the events for persistence: the BigQuery portion of the pipeline converts messages to TableRow objects and streams directly to BigQuery, while the AVRO portion of the pipeline batches

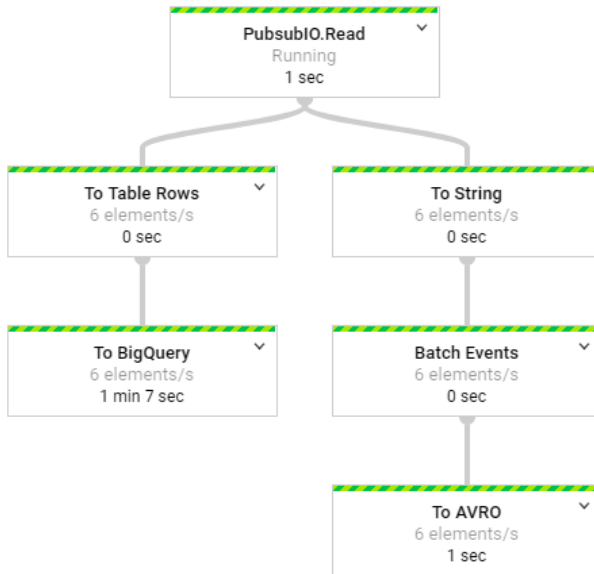


Figure 3.5: The streaming pipeline deployed to Google Cloud

events into discrete windows and then saves the events to Google Storage. The graph of operations is shown in the figure above.

### 3.3.1 Setting up the Environment

The first step in building a data pipeline is setting up the dependencies necessary to compile and deploy the project. I used the following maven dependencies to set up environments for the tracking API that sends events to the pipeline, and the data pipeline that processes events.

```

<!-- Dependencies for the Tracking API -->
<dependency>
  <groupId>com.google.cloud</groupId>
  <artifactId>google-cloud-pubsub</artifactId>
  <version>0.32.0-beta</version>
</dependency>
</dependencies>

```



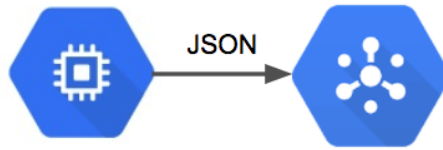


Figure 3.6: Sending events from a server to a PubSub topic

```
<!-- Dependencies for the data pipeline -->
<dependency>
  <groupId>com.google.cloud.dataflow</groupId>
  <artifactId>google-cloud-dataflow-java-sdk-all
    </artifactId>
  <version>2.2.0</version>
</dependency>
```

I used Eclipse to author and compile the code for this tutorial, since it is open source. However, other IDEs such as IntelliJ provide additional features for deploying and monitoring DataFlow tasks. Before you can deploy jobs to Google Cloud, you'll need to set up a service account for both PubSub and DataFlow. Setting up these credentials is outside the scope of this book, and more details are available in the Google documentation<sup>2</sup>.

An additional prerequisite for running this data pipeline is setting up a PubSub topic on GCP. I defined a raw-events topic that is used for publishing and consuming messages for the data pipeline. Additional details on creating a PubSub topic are available online<sup>3</sup>.

To deploy this data pipeline, you'll need to set up a java environment with the maven dependencies listed above, set up a Google Cloud project and enable billing, enable billing on the storage and BigQuery services, and create a PubSub topic for sending and receiving messages. All of these managed services do cost money, but there is a free tier that can be used for prototyping a data pipeline.

---

<sup>2</sup><https://cloud.google.com/bigquery/docs/authentication>

<sup>3</sup><https://cloud.google.com/pubsub/docs/quickstart-console>

### 3.3.2 Publishing Events

In order to build a usable data pipeline, it's useful to build APIs that encapsulate the details of sending event data. The Tracking API class provides this functionality, and can be used to send generated event data to the data pipeline. The code below shows the method signature for sending events, and shows how to generate sample data.

```
// send a batch of events
for (int i=0; i<10000; i++) {

    // generate event names
    String type = Math.random() < 0.5 ? "Session"
        (Math.random() < 0.5 ? "Login" : "MatchStart");

    // create attributes to send
    HashMap attributes = new HashMap();
    attributes.put("userID", (int)(Math.random()*100));
    attributes.put("deviceType", Math.random() < 0.5 ?
        "Android" : (Math.random() < 0.5 ? "iOS" : "Web"));

    // send the event
    tracking.sendEvent(type, "V1", attributes);
}
```

The tracking API establishes a connection to a PubSub topic, passes events as a JSON format, and implements a callback for notification of delivery failures. The code used to send events is provided below, and is based on Google's PubSub example.

```
// Setup a PubSub connection
TopicName topicName = TopicName.of(projectID, topicID);
Publisher publisher = Publisher
    .newBuilder(topicName).build();

// Specify an event to send
String event =
    {"type":"session","version":"1"};
```



Figure 3.7: Streaming event data from PubSub to DataFlow

```
// Convert the event to bytes
ByteString data = ByteString
    .copyFromUtf8(event.toString());

//schedule a message to be published
PubsubMessage msg =
    PubsubMessage.newBuilder().setData(data).build();

// publish the message
ApiFuture<String> future = publisher.publish(msg);
ApiFutures.addCallback(future, this);
```

The code above enables apps to send events to a PubSub topic. The next step is to process these events in a fully-managed environment that can scale as necessary to meet demand.

### 3.3.3 Storing Events

One of the key functions of a data pipeline is to make instrumented events available to data science and analytics teams for analysis. The data sources used as endpoints should have low latency and be able to scale up to a massive volume of events. The data pipeline defined in this tutorial shows how to output events to both BigQuery and a data lake that can be used to support a large number of analytics business users.

The first step in this data pipeline is reading events from a PubSub topic and passing ingested messages to the DataFlow process. DataFlow provides a PubSub connector that enables streaming of PubSub messages to other DataFlow components. The code below shows how to instantiate the data pipeline, specifying streaming

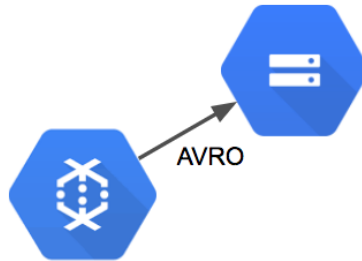


Figure 3.8: Batching events to AVRO format and saving to GS.

mode, and to consume messages from a specific PubSub topic. The output of this process is a collection of PubSub messages that can be stored for later analysis.

```
// set up pipeline options
Options options = PipelineOptionsFactory.fromArgs(args)
    .withValidation().as(Options.class);
options.setStreaming(true);
Pipeline pipeline = Pipeline.create(options);

// read game events from PubSub
PCollection<PubsubMessage> events = pipeline
    .apply(PubsubIO.readMessages().fromTopic(topic));
```

The first way we want to store events is in a columnar format that can be used to build a data lake. While this chapter doesn't show how to utilize these files in downstream ETLs, having a data lake is a great way to maintain a copy of your data set in case you need to make changes to your database. The data lake provides a way to backload your data if necessary due to changes in schemas or data ingestion issues. The portion of the data pipeline allocated to this process is shown below.

For AVRO, we can't use a direct streaming approach. We need to group events into batches before we can save to flat files. The way this can be accomplished in DataFlow is by applying a windowing function that groups events into fixed batches. The code below applies transformations that convert the PubSub messages into String objects, group the messages into 5 minute intervals, and

output the resulting batches to AVRO files on Google Storage. To summarize, the code batches events into 5 minute windows and then exports the events to AVRO files on Google Storage.

```
// AVRO output portion of the pipeline
events.apply("To String",
    ParDo.of(new DoFn<PubsubMessage, String>() {

        @ProcessElement
        public void processElement(ProcessContext c) {
            String msg = new String(c.element().getPayload());
            c.output(msg);
        }
    })))

// Batch events into 5 minute windows
.apply("Batch Events", Window.<String>into(
    FixedWindows.of(Duration.standardMinutes(5)))
    .triggering(AfterWatermark.pastEndOfWindow())
    .discardingFiredPanels()
    .withAllowedLateness(Duration.standardMinutes(5)))

// Save the events in ARVO format
.apply("To AVRO", AvroIO.write(String.class)
    .to("gs://your_gs_bucket/avro/raw-events.avro")
    .withWindowedWrites()
    .withSuffix(".avro"));
```

The result of this portion of the data pipeline is a collection of AVRO files on google storage that can be used to build a data lake. A new AVRO output is generated every 5 minutes, and downstream ETLs can parse the raw events into processed event-specific table schemas. The image below shows a sample output of AVRO files.

In addition to creating a data lake, we want the events to be immediately accessible in a query environment. DataFlow provides a BigQuery connector which serves this functionality, and data streamed to this endpoint is available for analysis after a short duration. This portion of the data pipeline is shown below.

The data pipeline converts the PubSub messages into TableRow objects, which can be directly inserted into BigQuery. The code









<input type="checkbox"/>	Name	Size	Type
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:02:00.000Z-2018-03-27...	2.46 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:02:00.000Z-2018-03-27...	2.43 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:03:00.000Z-2018-03-27...	2.5 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:03:00.000Z-2018-03-27...	2.52 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:04:00.000Z-2018-03-27...	2.96 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:04:00.000Z-2018-03-27...	2.99 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:05:00.000Z-2018-03-27...	2.41 KB	application/octet-stream
<input type="checkbox"/>	 raw-events.avro2018-03-27T04:05:00.000Z-2018-03-27...	2.38 KB	application/octet-stream

Figure 3.9: AVRO files saved to Google Storage

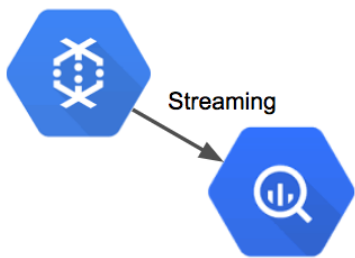


Figure 3.10: Streaming events from DataFlow to BigQuery

below consists of two apply methods: a data transformation and a IO writer. The transform step reads the message payloads from PubSub, parses the message as a JSON object, extracts the event-Type and eventVersion attributes, and creates a TableRow object with these attributes in addition to a timestamp and the message payload. The second apply method tells the pipeline to write the records to BigQuery and to append the events to an existing table.

```
// parse the PubSub events and create rows
events.apply("To Table Rows", ParDo.of(new
    DoFn<PubsubMessage, TableRow>() {

@ProcessElement
public void processElement(ProcessContext c) {
    String msg = new String(c.element().getPayload());

    // parse the json message for attributes
    JsonObject jsonObject =
        new JsonParser().parse(msg).getAsJsonObject();
    String type = jsonObject.get("type").getString();
    String eventVersion = jsonObject.
        get("eventVersion").getString();
    String serverTime = dateFormat.format(new Date());

    // create and output the table row
    TableRow record = new TableRow();
    record.set("eventType", type);
    record.set("eventVersion", eventVersion);
    record.set("serverTime", serverTime);
    record.set("message", message);
    c.output(record);
}
}))
//stream the events to Big Query
.apply("To BigQuery",BigQueryIO.writeTableRows()
    .to(table)
    .withSchema(schema)
    .withCreateDisposition(
        CreateDisposition.CREATE_IF_NEEDED)
    .withWriteDisposition(WriteDisposition.WRITE_APPEND))
```

Row	eventType	eventVersion	server_time	message
1	Login	V1	2018-03-25 19:18:56.720	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
2	Login	V1	2018-03-25 19:19:27.131	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
3	Login	V1	2018-03-25 20:35:27.784	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
4	Login	V1	2018-03-25 20:36:35.106	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
5	Login	V1	2018-03-25 20:40:49.629	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
6	Login	V1	2018-03-25 20:35:21.833	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"Web"}
7	Login	V1	2018-03-25 20:35:25.961	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"IOS"}
8	Login	V1	2018-03-25 20:36:20.849	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"IOS"}
9	Login	V1	2018-03-25 20:36:41.328	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"IOS"}
10	Login	V1	2018-03-25 20:36:24.757	{"eventType":"Login","eventVersion":"V1","userID":"0","deviceType":"IOS"}

Figure 3.11: Game event records queried from the raw-events table in BigQuery

Each message that is consumed from PubSub is converted into a TableRow object with a timestamp and then streamed to BigQuery for storage. The result of this portion of the data pipeline is that events will be streamed to BigQuery and will be available for analysis in the output table specified by the DataFlow task. In order to effectively use these events for queries, you'll need to build additional ETLs for creating processed event tables with schematized records, but you now have a data collection mechanism in place for storing tracking events.

### 3.3.4 Deploying and Auto Scaling

With DataFlow you can test the data pipeline locally or deploy to the cloud. If you run the code samples without specifying additional attributes, then the data pipeline will execute on your local machine. In order to deploy to the cloud and take advantage of the auto scaling capabilities of this data pipeline, you need to specify a new runner class as part of your runtime arguments. In order to run the data pipeline, I used the following runtime arguments:

```
--runner=org.apache.beam.runners.dataflow.DataflowRunner
--jobName=game-analytics
--project=your_project_id
--tempLocation=gs://temp-bucket
```

Once the job is deployed, you should see a message that the job has been submitted. You can then click on the DataFlow console



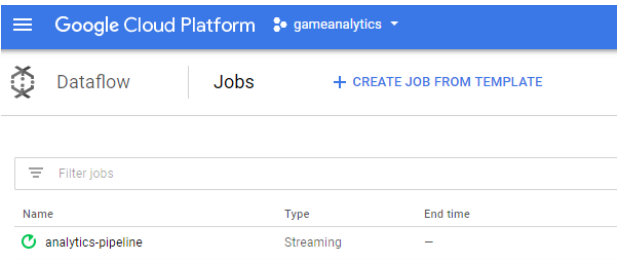


Figure 3.12: The steaming data pipeline running on Google Cloud



Figure 3.13: An example of Dataflow auto scaling.

to see the task. The runtime configuration specified above will not default to an auto scaling configuration. In order to deploy a job that scales up based on demand, you'll need to specify additional attributes:

```
--autoscalingAlgorithm=THROUGHPUT_BASED
--maxNumWorkers=30
```

Additional details on setting up a DataFlow task to scale to heavy workload conditions are available from Spotify<sup>4</sup>. The image below shows how DataFlow can scale up to meet demand as necessary.

3.3.5 Raw to Processed Events

The pipeline presented so far saves tracking events as raw data. To translate these events to processed data, we'll need to apply event

<sup>4</sup><https://labs.spotify.com/2016/03/10/>

specific schemas. There's a few different approaches we can take with this pipeline:

- Apply the schemas in the current DataFlow pipeline and save to BigQuery
- Apply the schemas in the pipeline and send to a new PubSub
- Apply additional attributes to the raw events and send to a new PubSub
- Use downstream ETLs to apply schemas

The first approach is the simplest, but it doesn't provide a good solution for updating the event definitions if needed. This approach can be implemented as shown in the code below, which shows how to filter and parse MatchStart events for entry into BigQuery.

```
events.apply("To MatchStart Events", ParDo.of(
    new DoFn<PubsubMessage, TableRow>() {

@ProcessElement
public void processElement(ProcessContext c) {
    String msg = new String(c.element().getPayload());
    JsonObject jsonObject = new
        JsonParser().parse(msg).getAsJsonObject();
    String eventType = jsonObject.get("type");
    String version = jsonObject.get("eventVersion");
    String serverTime = dateFormat.format(new Date());

    // Filter for MatchStart events
    if (eventType.equals("MatchStart")) {

        TableRow record = new TableRow();
        record.set("eventType", eventType);
        record.set("eventVersion", version);
        record.set("server_time", serverTime);

        // event specific attributes
        record.set("userID", jsonObject.get("userID"));
        record.set("type", jsonObject.get("deviceType"));
        c.output(record);
    }
}}))
.apply("To BigQuery", BigQueryIO.writeTableRows())
```

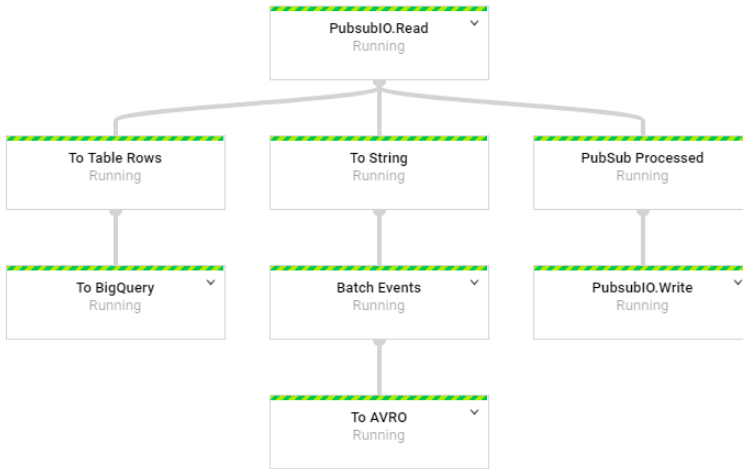


Figure 3.14: The streaming pipeline with an additional output,

In order to implement this approach, you'd need to create a new DoFn implementation for each type of event. The second approach is similar to the first, but instead of passing the parsed events to BigQuery, they are passed to a new PubSub topic. It's possible to send multiple types of events to a single topic or create a topic per event. The drawback of using the first two approaches is that the message parsing logic is part of the raw event pipeline. This means that changing event definitions involves restarting the pipeline.

A third approach that can be used is sending raw events with additional attributes to another PubSub topic. A second DataFlow job can then be set up to parse events as needed. The code below shows how to parse raw events, add additional attributes to the PubSub message for filtering, and publish the events to a second topic. This approach enables event definitions to be changed without restarting the raw event pipeline.

```
// topic for raw events with additional attributes
private static String processed =
    "projects/your_project_id/topics/processed-events";

events.apply("PubSub Processed",
```

```

ParDo.of(new DoFn<PubsubMessage, PubsubMessage>() {

    @ProcessElement
    public void processElement(ProcessContext c) {
        String msg = new String(c.element().getPayload());

        // parse the JSON message for attributes
        JsonObject jsonObject = new
            JsonParser().parse(msg).getAsJsonObject();
        String eventType = jsonObject.get("eventType");

        // Add additional attributes for filtering
        HashMap<String, String> atts = new HashMap();
        atts.put("EventType", eventType);
        PubsubMessage out = new PubsubMessage(
            msg.getBytes(), atts);
        c.output(out);
    }
}))
.apply(PubsubIO.writeMessages().to(processed));

```

A fourth approach that can be used is having downstream ETLs processes apply schemas to the raw events and break apart the raw events table into event specific tables. We'll cover this approach in the next chapter.

## 3.4 Conclusion

This chapter has provided an introduction to building a data pipeline for a startup. We covered the types of data in a pipeline, desired properties of a high functioning data pipeline, the evolution of data pipelines, and a sample pipeline built on GCP. The full source code for this sample pipeline is available on Github<sup>5</sup>.

There is now a variety of tools available that make it possible to set up an analytics pipeline for an application with minimal effort. Using managed resources enables small teams to take advantage of serverless and autoscaling infrastructure to scale up to massive event volumes with minimal infrastructure management. Rather

---

<sup>5</sup><https://github.com/bgweber/GameAnalytics>

than using a data vendor's off-the-shelf solution for collecting data, you can record all relevant data for your app. While the approach presented here isn't directly portable to other clouds, the Apache Beam library used to implement the core functionality of this data pipeline is portable and similar tools can be leveraged to build scalable data pipelines on other cloud providers.



## Chapter 4

# Business Intelligence

A lot of the heavy lifting involved in setting up data science at a startup is convincing the product team to instrument and care about data. If you're able to achieve this goal, the next step is being able to answer all sorts of questions about product health within your organization. A novice data scientist might think that this type of work is outside the role of a data scientist, but identifying key metrics for product health is one of the core facets of the role.

I've titled this chapter as business intelligence, because once you've set up a data pipeline, a data scientist in a startup is expected to answer every question about data. This is not surprising given the new flood of data, but also a time for a data scientist to set expectations for the rest of the organization. As a data scientist in a startup, your function is not to answer data questions, but to inform the leadership about what metrics should be important.

This chapter covers the basics of how to turn raw data into cooked data that can summarize the health of a product. I'll discuss a few different approaches to take when working with raw data, including SQL queries, R markdown, and vendor tools. The general takeaway is to show that several options are available for processing data sets, and you should choose a solution that fits the goals of your team. I'll discuss past experiences with tools such as Tableau, and provide recommendations for scaling automated reporting across a team.

We'll use two data sources for this chapter. The first is a public data set that we'll aggregate and summarize with key metrics. The

second is data generated by the tracking API in the second chapter of this series. We'll focus on the second data set for transforming raw to processed data, and the first data set for processed to cooked data.

## 4.1 KPIs

Key Performance Indicators (KPIs) are used to track the health of a startup. It's important to track metrics that capture engagement, retention, and growth, in order to determine if changes made to the product are beneficial. As the data scientist at a startup, your role has the responsibility of identifying which metrics are important. This function aligns with the data science competency of domain knowledge, and is one of the areas where a data scientist can be highly influential.

KPIs that are established by an early data scientist can have have a resounding impact. For example, many of the past companies I worked at had company goals based on past analyses of data scientists. At Electronic Arts we were focused on improving session metrics, at Twitch we wanted to maximize the amount of content watched, and at Sony Online Entertainment we wanted to improve retention metrics for free-to-play titles. These were game industry metrics, but there are more general metrics such as engagement, growth, and monetization that are important to track when building a company.

It's important when building a data science discipline at a startup to make sure that your team is working on high impact work. One of the problems I've seen at past companies is data scientists getting pulled into data engineering and analytics type of work. This is expected when there's only one data person at the company, but you don't want to support too many manual data processes that won't scale. This is why setting up reproducible approaches for reporting and analysis is important. It should be trivial to rerun an analysis months down the road, and it should be possible for another team member to do so with minimal direction.

My main advice for new data scientists to prevent getting overwhelmed with requests from product managers and other teams is to set up an interface to the data science team that buffers direct requests. Instead of having anyone at the company being able to



ask the data science team how things are performing, a baseline set of dashboards should be set up to track product performance. Given that a data scientist may be one of the first data roles at a startup, this responsibility will initially lie with the data scientist and it's important to be familiar with a number of different tools in order to support this function at a startup.

## 4.2 Reporting with R

One of the key transitions that you can make at a startup as a data scientist is migrating from manual reporting processes to reproducible reports. R is a powerful programming language for this type of work, and can be used in a number of different ways to provide automated reporting capabilities. This section discusses how to use R for creating plots, generating reports, and building interactive web applications. While many of these capabilities are also provided by Python and the Jupyter suite, the focus on automation is more important than the language used to achieve this goal.

It's possible to achieve some of this type of functionality with Excel or Google Sheets, but I would advise against this approach for a startup. These tools are great for creating charts for presentations, but not suitable for automated reporting. It's not sustainable for a data scientist to support a startup based on these types of reports, because so many manual steps may be necessary. Connectors like ODBC in Excel may seem useful for automation, but likely won't work when trying to run reports on another machine.

This section covers three approaches to building reports with R: using R directly to create plots, using R Markdown to generate reports, and using Shiny to create interactive visualizations. All of the code listed in this section is available on Github<sup>1</sup>.

### 4.2.1 Base R

Consider a scenario where you are part of a NYC startup in the transportation sector, and you want to determine what type of

---

<sup>1</sup><https://github.com/bgweber/StartupDataScience/tree/master/BusinessIntelligence>

payment system to use to maximize the potential of growing your user base. Luckily, there's a public data set that can help with answering this type of question: BigQuery's NYC Taxi and Limousine Trips public data set<sup>2</sup>. This collection of trip data includes information on payments that you can use to trend the usage of payment types over time.

The first approach we'll use to answer this question is using a plotting library in R to create a plot. I recommend using the RStudio IDE when taking this approach. Also, this approach is not actually "Base R", because I am using two additional libraries to accomplish the goal of summarizing this data set and plotting the results. I'm referring to this section as Base R, because I am using the built-in visualization capabilities of R.

One of the great aspects of R is that there's a variety of different libraries available for working with different types of databases. The `bigrquery` library provides a useful connector to BigQuery that can be used to pull data from the public data set within an R script. The code for summarizing the payment history over time and plotting the results as a chart are shown below.

```
library(bigrquery)
library(plotly)
project <- "your_project_id"
sql <- "SELECT
  substr(cast(pickup_datetime as String), 1, 7) as date
  ,payment_type as type
  ,sum(total_amount) as amount
FROM `nyc-tlc.yellow.trips`
group by 1, 2"
df <- query_exec(sql, project = project,
  use_legacy_sql = FALSE)
plot_ly(df, x = ~date, y = ~amount,
  color = ~type) %>% add_lines()
```

The first part of this script, which includes everything except for the last line, is responsible for pulling the data from BigQuery. It loads the necessary libraries, states a query to run, and uses `bigrquery` to fetch the result set. Once the data has been pulled into a data frame, the second part of the script uses the `plotly` library to display

---

<sup>2</sup><https://cloud.google.com/bigquery/public-data/nyc-tlc-trips>

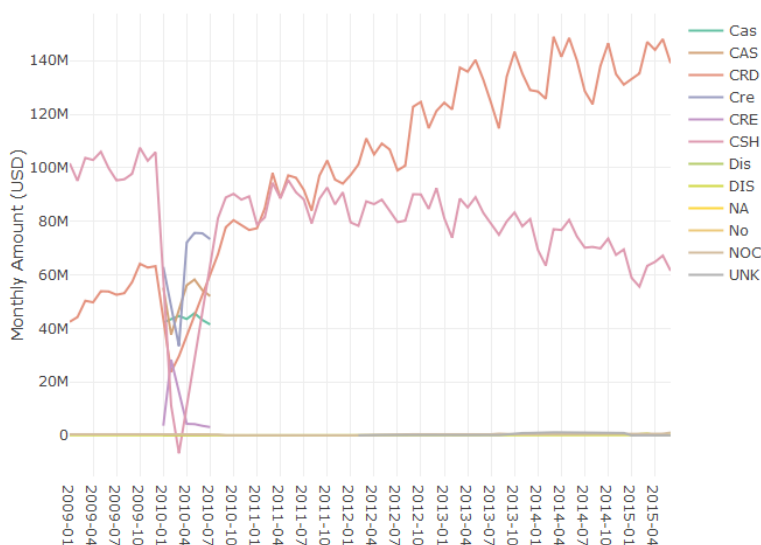


Figure 4.1: Monthly Spending by Payment

the results as a line chart. Some additional formatting steps have been excluded from the script, and the full code listing is available on Github. In RStudio, the chart will show up as an interactive plot in the IDE, and Jupyter provides similar functionality. The result of this code snippet is shown in the chart below.

The query calculates the total monthly spend by payment type for taxi trips in NYC, using data from 2009 to 2015. The results show that credit cards (CRD) are now the preferred payment method over cash (CSH). To answer the initial question about what type of payment system to implement, I'd recommend starting with a system that accepts credit cards.

One topic worth bringing up at this point is data quality, since the chart has a number of different labels that seem to represent the same values. For example CAS and CSH both likely refer to cash payments and should be grouped together to get an accurate total of cash payments. Dealing with these types of issues is outside the scope of this approach, but there are a few methods that can be used for this type of scenario. The easiest but least scalable approach is to write queries that account for these different types:

```
,sum(case when payment_type in ('CSH', 'CAS')
      then amount else 0 end) as cash_payments
```

A different approach that can be used is creating a dimension table that maps all of the raw `payment_type` values to sanitized type values. This process is often called attribute enrichment, and is useful when building out cooked data sets from raw or processed data.

We’ve answered the first question about determining the most popular payment method, but what if we have a second question about whether or not the transportation market in NYC is growing? We can easily plot data to answer this question using the existing data:

```
total <- aggregate(df$Amount,
                  by=list(Category=df$Date), FUN=sum)
plot_ly(total, x = ~Category, y = ~x) %>% add_lines()
```

This code computes the total monthly payments across all of the different payment types, and plots the aggregate value as a single line chart. The results are shown in the figure below. Based on the initial observation of this data, the answer to the second question is unclear. There’s been a steady increase in taxi spending in NYC from 2009 to 2013, with seasonal fluctuations, but spending peaked in summer of 2014. It’s possible that Uber and Lyft account for this trend, but further analysis is needed to draw a firm conclusion.

This section has shown how to use R to generate plots from summarized data in BigQuery. While this sample used a fixed data set, the same approach could be used with a live data set that grows over time, and rerunning the script will include more recent data. This is not yet automated reporting, because it involves manually running the code in an IDE or notebook. One approach that could be used is outputting the plot to an image file, and running the script as part of a cron job. The result of this approach is an image of the plot that gets updated on a regular schedule. This is a good starting point, but there are more elegant solutions for automated reporting in R.

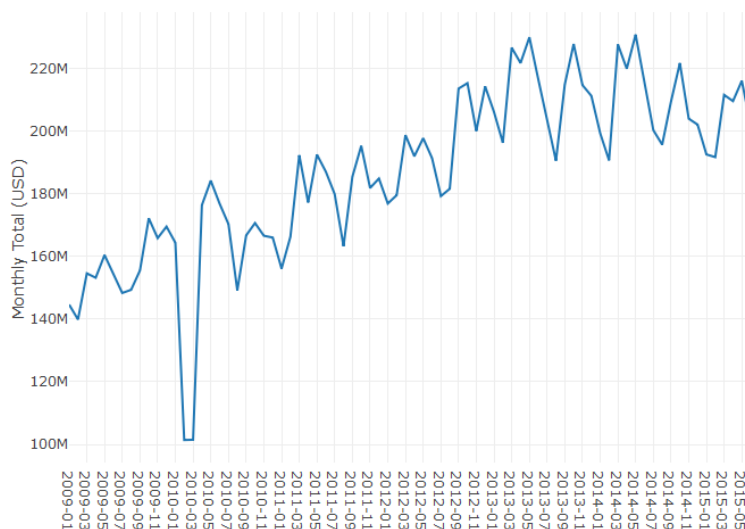


Figure 4.2: Total Monthly Spending

### 4.2.2 R Markdown

Let's say you want to perform the same analysis as before, but want to produce a report each time you run the script. R Markdown provides this capability, and can use R code to generate PDFs, word documents (DOCX), and web pages (HTML). You can even write books with R Markdown! R Markdown extends standard markdown to support inline R snippets that can be used to generate visualizations. The embedded R code can perform almost any standard R functionality, including using R libraries and making connections to databases. This means we can convert the code above into an R markdown file, and run the script regularly to build automated reporting.

The markdown snippet below is the previous R code now embedded in a report that will generate an HTML file as output. The first part of the file is metadata about the report, including the desired output. Next, markdown is used to add commentary to the report. And finally, a R code block is used to pull data from BigQuery and plot the results. The resulting plotly object is embedded into the document when running this report.

```

---
title: "Business Intelligence"
output: html_document
---

## Taxi Payments
R Markdown can outputs reports as PDF or HTML.

`` ` {r echo=FALSE, message=FALSE, warning=FALSE}
library(bigrquery)
library(plotly)
project <- "your_project_id"
sql <- "SELECT
  substr(cast(pickup_datetime as String), 1, 7) as date
  ,payment_type as type
  ,sum(total_amount) as amount
FROM `nyc-tlc.yellow.trips`
group by 1, 2"
df <- query_exec(sql, project = project,
  use_legacy_sql = FALSE)
plot_ly(df, x = ~date, y = ~amount,
  color = ~type) %>% add_lines()
` ``

```

The resulting HTML document is shown in the figure below. It includes that same plot as before, as well as the markdown text listed before the code block. This output can be more useful than an image, because the plotly charts embedded in the file are interactive, rather than rendered images. It's also useful for creating reports with a variety of different charts and metrics.

To automate creating this report, you can again set up a cron job. The command for converting the Rmd file to a report is:

```
Rscript -e "rmarkdown::render('BI.Rmd')"
```

We now have a way of generating reports, and can use cron to start building an automated reporting solution. However, we don't yet have charts that provide filtering and drill-down functionality.

# Business Intelligence

Ben Weber

May 21, 2018

## Taxi Payments

R Markdown can outputs reports as PDF or HTML.

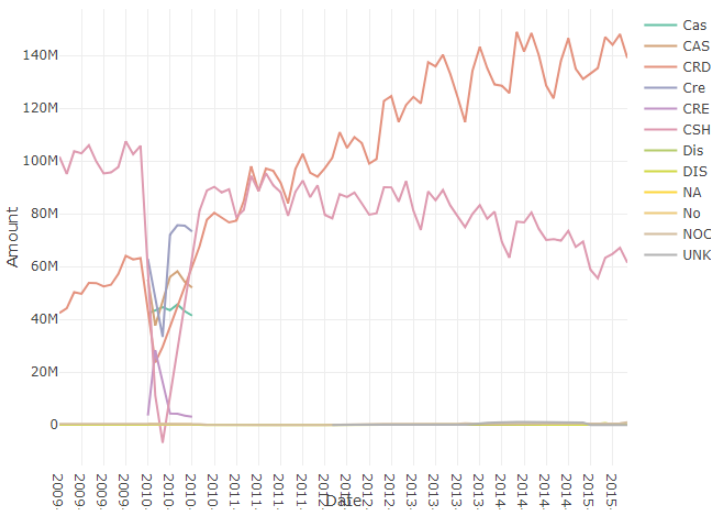


Figure 4.3: The report generated from the R Markdown file.

### 4.2.3 R Shiny

Shiny is a solution for building dashboards directly in R. It provides functionality for building reports with filtering and drill-down capabilities, and can be used as an alternative to tools such as Tableau. When using Shiny, you specify the UI components to include in the report and the behaviors for different components in a report, such as applying a filter based on changes to a slider component. The result is an interactive web app that can run your embedded R code.

I've created a sample Shiny application based on the same code as the above reports. The first part of the code is the same, we pull data from BigQuery to a dataframe, but we also include the shiny library. The second part of the code defines the behavior of different components (server), and the layout of different components (ui). These functions are passed to the shinyApp call to launch the dashboard.

```
library(shiny)
library(bigrquery)
library(plotly)
project <- "your_project_id"
sql <- "SELECT
substr(cast(pickup_datetime as String), 1, 7) as date
,payment_type as type
,sum(total_amount) as amount
FROM `nyc-tlc.yellow.trips`
group by 1, 2"
df <- query_exec(sql, project = project,
                  use_legacy_sql = FALSE)
server <- function(input, output) {
  output$plot <- renderPlotly({
    plot_ly(df[df$date >= input$year, ], x = ~date,
            y = ~amount, color = ~type) %>% add_lines()
  })
}
ui <- shinyUI(fluidPage(
  sidebarLayout(
    sidebarPanel(
      sliderInput("year", "Start Year:",
```



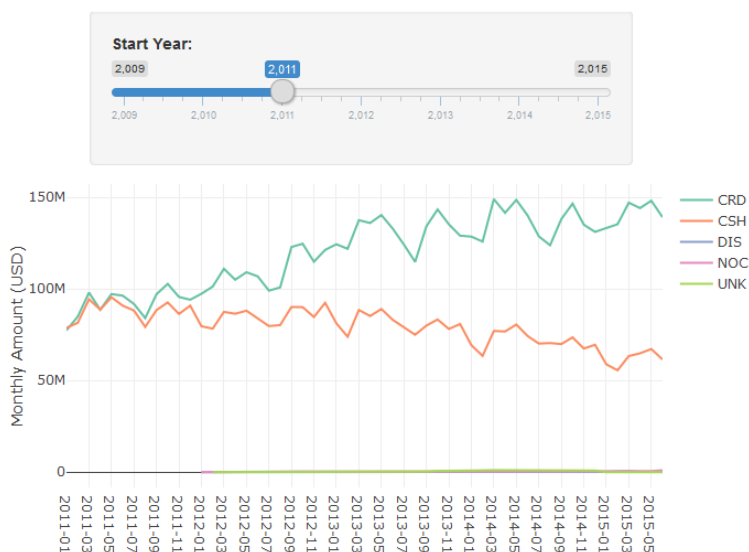


Figure 4.4: An interactive Chart in R Shiny.

```

    min = 2009, max = 2015, value = 2012)
  ),
  mainPanel(plotlyOutput("plot"))
)
))
shinyApp(ui = ui, server = server)

```

The UI function specifies how to lay out the components in the dashboard. I started with the Hello Shiny example, which includes a slider and histogram, and modified the layout to use a `plotlyOutput` object instead of a `plotOutput`. The slider specifies the years to allow for selection, and sets a default value. The behavior function specifies how to respond to changes in UI components. The plot is the same as behavior, with one modification, it now filters on the starting data when using the data frame `dfdate >= inputyear`. The result is the interactive dashboard shown below. Moving the slider will now filter the years that are included in the chart.

I've now shown three different ways to generate reports using R. If you need interactive dashboards, then Shiny is a great tool to

explore, while if you're looking to build static reports, then R Markdown is a great solution. One of the key benefits of both of these approaches is that you can embed complex R logic within your charts, such as using Facebook's prophet library to add forecasted values to your charts.

## 4.3 ETLs

In the chapter on data pipelines, I discussed using raw, processed, and cooked data. Most reports used for business intelligence should be based on cooked data, where data is aggregated, enriched, and sanitized. If you use processed or raw data instead of cooked data when building reports, you'll quickly hit performance issues in your reporting pipeline. For example, instead of using the `nyc-tlc.yellow.trips` table directly in the R section above, I could have created a table with the aggregate values precomputed.

ETL is an abbreviation of Extract-Transform-Load. One of the main uses of these types of processes is to transform raw data into processed data or processed data into cooked data, such as aggregation tables. One of the key challenges in setting up aggregates tables is keeping the tables updated and accurate. For example, if you started tracking cash payments using a new abbreviation (e.g. CAH), you would need to update the aggregation process that computes monthly cash payments to include this new payment type.

One of the outputs of the data pipeline is a raw events table, that includes data for all of the tracking events encoded as JSON. One of the types of ETL processes we can set up is a raw to processed data transformation. In BigQuery, this can be implemented for the login event as follows:

```
create table tracking.logins as (  
  select eventVersion,server_time  
    ,JSON_EXTRACT_SCALAR(message, '$.userID') userID  
    ,JSON_EXTRACT_SCALAR(message, '$.deviceType') type  
  from tracking.raw_events  
  where eventType = 'Login'  
)
```

This query filters on the login events in the raw events table, and uses the JSON extract scalar function to parse elements out of the JSON message. The result of running this DDL statement will be a new table in the tracking schema that includes all of the login data. We now have processed data for logins with `userID` and `deviceType` attributes that can be queried directly.

In practice, we'll want to build a table like this incrementally, transforming only new data that has arrived since the last time the ETL process ran. We can accomplish this functionality using the approach shown in the SQL code below. Instead of creating a new table, we are inserting into an existing table. With BigQuery, you need to specify the columns for an insert operation. Next, we find the last time when the login table was updated, represented as the `updateTime` value. And finally, we use this result to join on only login events that have occurred since the last update. These raw events are parsed into processed events and added to the logins table.

```
insert into tracking.logins
  (eventVersion,server_time, userID, deviceType)
with lastUpdate as (
  select max(server_time) as updateTime
  from tracking.logins
)
select eventVersion,server_time
  ,JSON_EXTRACT_SCALAR(message, '$.userID') userID
  ,JSON_EXTRACT_SCALAR(message, '$.deviceType') type
from tracking.raw_events e
join lastUpdate l
  on e.server_time > updateTime
where eventType = 'Login'
```

A similar approach can be used to create cooked data from processed data. The result of the login ETL above is that we now can query against the `userID` and `deviceType` fields directly. This processed data makes it trivial to calculate useful metrics such as daily active users (DAU), by platform. An example of computing this metric in BigQuery is shown below.

Date	deviceType	DAU
2018-03-25	Android	9995
2018-03-25	Web	9814
2018-03-25	iOS	9825
2018-03-26	Android	2786
2018-03-26	Web	1553
2018-03-26	iOS	1493

Figure 4.5: Cooked Data: DAU by Platform.

```
create table metrics.dau as (
  select substr(server_time, 1, 10) as Date
    ,deviceType, count(distinct userID) as DAU
  from `tracking.logins`
  group by 1, 2
  order by 1, 2
)
```

The result of running this query is a new table with the DAU metric precomputed. A sample of this data is shown in the Cooked Data table. Similar to the previous ETL, in practice we'd want to build this metric table using an incremental approach, rather than rebuilding using the complete data set. A slightly different approach would need to be taken here, because DAU values for the current day would need to be updated multiple times if the ETL is ran multiple times throughout the day.

Once you have a set of ETLs to run for your data pipeline, you'll need to schedule them so that they run regularly. One approach you can take is using cron to set up tasks, such as:

```
bq query --flagfile=/etls/login_etl.sql
```

It's important to set up monitoring for processes like this, because a failure early on in a data pipeline can have significant downstream impacts. Tools such as Airflow can be used to build out complex data pipelines, and provide monitoring and alerting.

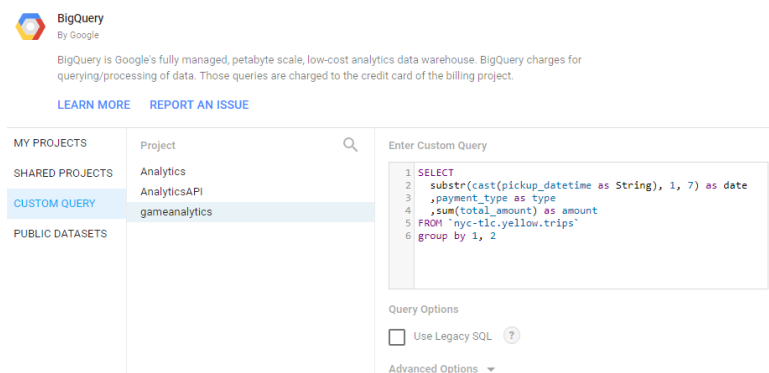


Figure 4.6: Setting up a Custom Data Source in Data Studio.

## 4.4 Reporting Tools

While R does provide useful tools for performing business intelligence tasks, it's not always the best tool for building automated reporting. This is common when reporting tools need to be used by technical and non-technical users and vendor solutions for building dashboards are often useful for these types of scenarios. Here are a few of the different tools I've used in the past.

### 4.4.1 Google Data Studio

If you're already using GCP, then Google Data Studio is worth exploring for building dashboards to share within your organization. However, it is a bit clunkier than other tools, so it's best to hold off on building dashboards until you have a mostly complete spec of the reports to build.

The image above shows how to set up a custom query in Google Data Studio to pull the same data sets as used in the R reports. The same report as before, now implemented with Data Studio is shown below.

The main benefit of this tool is that it provides many of the collaboration features built into other tools, such as Google Docs and Google Sheets. It also refreshes reports as necessary to keep data from becoming stale, but has limited scheduling options available.

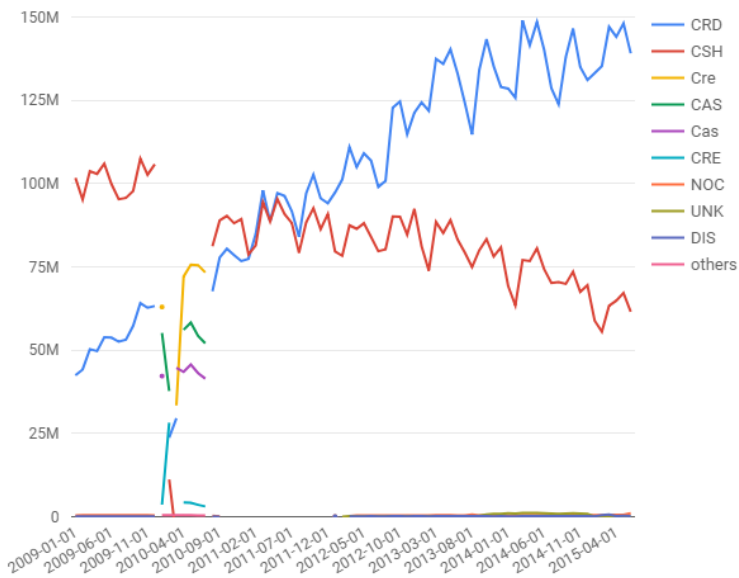


Figure 4.7: The Taxi Report recreated in Google Data Studio.

#### 4.4.2 Tableau

One of the best visualization tools I've used is Tableau. It works well for the use case of building dashboards when you have a complete spec, and well as building interactive visualizations when performing exploratory analysis. The heatmap for DC Universe Online was built with Tableau, and is one of many different types of visualizations that can be built.

The main benefit of Tableau is ease-of-use in building visualizations and exploring new data sets. The main drawback is pricing for licenses, and a lack of ETL tooling, since it is focused on presentation rather than data pipelines.

#### 4.4.3 Mode

At Twitch, we used a vendor tool called Mode Analytics. Mode made it simple to share queries with other analysts, but has a

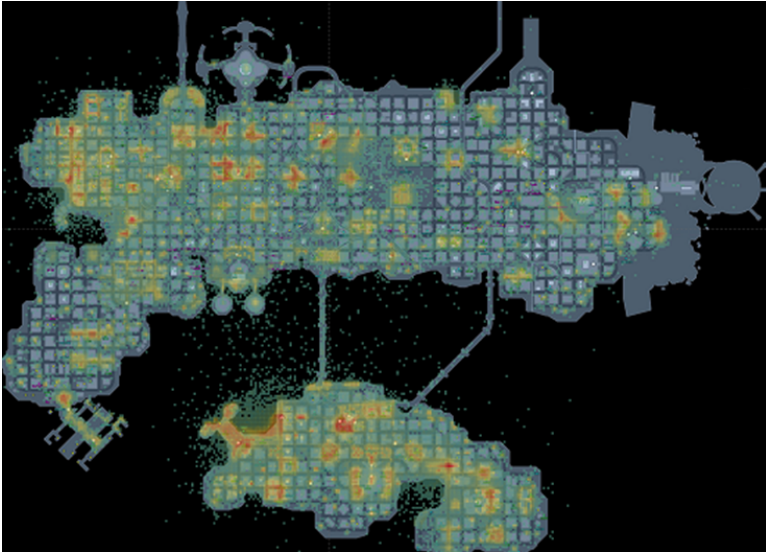


Figure 4.8: A heatmap in Tableau for the game DC Universe Online.

rather limited selection of visualization capabilities, and also was focused on only presentation and not ETL type tasks.

#### 4.4.4 Custom Tooling

Another approach that can be used is creating custom visualizations using tools such as D3.js and Protovis. At Electronic Arts, D3 was used to create customer dashboards for game teams, such as the Data Cracker tool built by Ben Medler for visualizing playtesting data in Dead Space 2. Using custom tooling provides the most flexibility, but also requires maintaining a system, and is usually substantially more work to build.

## 4.5 Conclusion

One of the key roles of a data scientist at a startup is making sure that other teams can use your product data effectively. Usually this takes the form of providing dashboarding or other automated

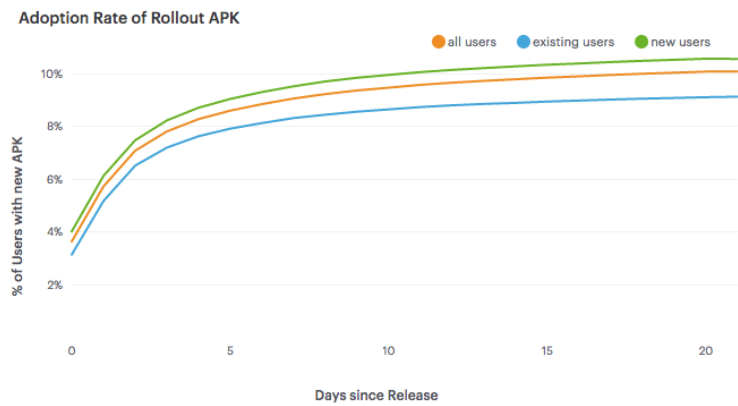


Figure 4.9: Line Charts in Mode Analytics.



Figure 4.10: The Data Cracker Tool for Dead Space 2. Source: GDC Vault 2011.



reporting, in order to provide KPIs or other metrics to different teams. It also includes identifying which metrics are important for the company to measure.

This chapter has presented three different ways for setting up automated reporting in R, ranging from creating plots directly in R, using R Markdown to generate reports, and using Shiny to build dashboards. We also discussed how to write ETLs for transforming raw data to processed data and processed data to cooked data, so that it can be used for reporting purposes. And the last section discussed some different vendor solutions for reporting, along with tradeoffs.

After setting up tooling for business intelligence, most of the pieces are in place for digging deeper into data science type of work. We can move beyond retrospective types of questions, and move forward to forecasting, predictive modeling, and experimentation.



## Chapter 5

# Exploratory Data Analysis

Once you've set up a data pipeline and collected data about user behavior, you can start exploring the data in order to determine how to improve your product. Exploratory Data Analysis (EDA) is the process of investigating a data set to understand the shape of the data, correlations between features, and signals in the data that may be useful for building predictive models.

It's useful to be able to perform this task in both a query language and a scripting language. R provides useful tools for quickly understanding the shape of a data set, but can only analyze data sets that can fit in memory. To work with massive data sets, SQL is useful for computing summary stats and distributions across a complete data set.

This chapter presents four types of exploratory analyses, including computing summary statistics, plotting features, correlation analysis, and weighting feature importance for a simple linear model. The goal of performing this type of work is to be able to better understand user behavior, determine how to improve a product, and investigate if the data provides useful signals.

We'll use the Natality BigQuery data set for the EDA examples. This data set provides information about births in the USA over the past 50 years. The goal of this analysis is to determine which factors

are correlated with birth weight, and build a linear regression model to predict outcomes.

## 5.1 Summary Statistics

The first thing we'll want to do when exploring a new data set is computing summary statistics that help us understand the data set. This can include statistics such as the mean and median values, and well as extremes such as minimum and maximum values. R provides a function called `summary` that calculates these statistics for every column in a data frame. An example using this function is shown in the code snippet below.

```
library(bigrquery)
project <- "your_project_id"
sql <- "
  SELECT year, plurality, mother_age, father_age,
         gestation_weeks, ever_born,
         mother_married, weight_pounds
  FROM `bigquery-public-data.samples.natality`
  order by rand()
  LIMIT 100000"
df <- query_exec(sql, project = project,
                  use_legacy_sql = FALSE)
summary(df[, 2:5])
```

The script queries the Natality data set in BigQuery and pulls a sample data set locally. Next, the second through fifth columns of the result set are passed to the `summary` function. The result of calling this function is shown in the figure below. For each of the columns, the function calculates min, mean, max values, and the 25th, 50th, and 75th percentiles for each attribute. It also counts the number of instances of missing (NA) values.

`Summary` provides a quick way of understanding a data set, but usually requires further digging to really understand the shape of the data. The next section shows how to use histograms to build a more complete understanding of a data set. One of the interesting features in the data set is the `plurality` column, which describes the number of children carried as part of the pregnancy (litter size).

```
> summary(df[, 2:5])
  plurality      mother_age      father_age      gestation_weeks
Min.   :1.000   Min.   :11.00   Min.   :14.00   Min.   :17.00
1st Qu.:1.000   1st Qu.:22.00   1st Qu.:25.00   1st Qu.:38.00
Median :1.000   Median :26.00   Median :30.00   Median :39.00
Mean   :1.026   Mean   :26.27   Mean   :38.95   Mean   :41.94
3rd Qu.:1.000   3rd Qu.:30.00   3rd Qu.:37.00   3rd Qu.:41.00
Max.   :4.000   Max.   :50.00   Max.   :99.00   Max.   :99.00
NA's   :2670                      NA's   :3397
```

Figure 5.1: Summary Stats for the Natality Data Set.

The median is slightly above 1, because of the rare occurrence of twins, triples, or even more children. Due to the skewness of this distribution, the summary stats do not provide a good overview of how common twins or triplets are in human pregnancies.

To find out how common twins and triplets are we can use the `squidf` library, which enables queries to be performed on R data frames. The code below shows how to count the number of pregnancies that result in multiple children being delivered. The results show that twins have a frequency of about 2.4%, triplets occur in about 0.1% of pregnancies, and quadruplets have a frequency of about 0.009%.

```
library(squidf)
df <- df[!is.na(df$plurality), ]
squidf("select plurality, sum(1) as Babies
from df group by 1
order by 1")
```

These results are based on a sample of 1,000,000 pregnancies. Ideally, we want to calculate these statistics across our compute data set. Aggregate values such as min, max, and mean are easy to compute, because SQL has aggregations for these operations built-in. However, calculating the median and 25th and 75th percentiles is often non trivial. If you try to apply the `percentile_cont` operation to the complete data set of 138M pregnancies, BigQuery will throw an exception, because this is an analytic function that will shuffle all of the records to a single node.

There's a few different approaches to work around this limitation. BigQuery does provide an approximate quantiles function that will support this size of data. You can also partition the data set using a

randomly generated value, such as `rand()*10` and take the average, to get an approximate result. These approaches work well for the 25th, 50th, and 75th percentile values, but are not as accurate for extreme results, such as the 99.9th percentile. A third approach is to provide a partition key to split up the data, preventing too much data from being shuffled to a single machine. We can use the year field, as shown in the following query.

```
select year, sum(1) births, min(father_age) Min
,avg(Q1) Q1, avg(Q2) Median
,round(avg(father_age), 1) Mean
,avg(Q3) Q3, max(father_age) Max
from (
  select year, father_age
    ,percentile_cont(father_age, 0.25)
      over (partition by year) as Q1
    ,percentile_cont(father_age, 0.50)
      over (partition by year) as Q2
    ,percentile_cont(father_age, 0.75)
      over (partition by year) as Q3
  FROM `bigquery-public-data.samples.natality`
  where mod(year, 5) = 0 and year >= 1980
)
group by 1 order by 1
```

This approach calculates the 25th, 50th, and 75th percentiles for the `father_age` column. The `percentile_cont` function is an analytic function, meaning that it returns a result for every record, and needs to be aggregated to create a summary statistic. The query above shows how to compute the same statistics provided by the R summary function using BigQuery, while partitioning the data by the year of the birth. This query generates the following table, which shows statistics about the father age.

There's a few interesting trends in this data worth noting. The first is the consistent value of 99 years, as the maximum father age. This is likely a data quality issue, and 99 may be set as the age when the age of the father is unknown. Another trend is that the median father age has been increasing over the years, from 28 in 1980 to 32 in 2005. This isn't normalized for the first child, but the summary statistics do indicate a trend that families are starting to have children later in life.

Row	year	births	Min	Q1	Median	Mean	Q3	Max
1	1980	3310301	10	24.0	28.0	37.0	34.0	99
2	1985	3765064	10	25.0	30.0	38.3	36.0	99
3	1990	4162917	10	26.0	31.0	41.0	38.0	99
4	1995	3903012	10	26.0	31.0	40.5	38.0	99
5	2000	4063823	10	26.0	31.0	39.7	38.0	99
6	2005	4145619	10	26.0	32.0	40.0	38.0	99

Figure 5.2: Birth Summary Statistics by Year (Father Age).

## 5.2 Plotting

Summary statistics provide a quick way of getting a snapshot of a data set, but often don't provide a good overview of how data is distributed. To build a more complete understanding of the shape of a data set, it's useful to plot data in various ways, such as using histograms.

```
hist(df$weight_pounds, main = "Distribution of Weights",
     ,xlab = "Weight")
```

The R code above shows how to plot a histogram of the birth weights in our sampled data set. The result is the following histogram, which seems to indicate that the value is normally distributed. The majority of births weights are between 6 and 9 pounds.

```
plot(ecdf(df$weight_pounds), main = "CDF of Weights",
     xlab = "Weight", ylab = "CDF")
```

Histograms are useful for visualizing distributions, but they do have a number of problems, such as the number of buckets influencing whether the distribution is unimodal vs bimodal. It's also usually difficult to determine percentiles directly from a histogram. One type of visualization that works better for conveying this type of information is the cumulative distribution function (CDF). The code above shows how to visualize the same data, but uses a CDF.

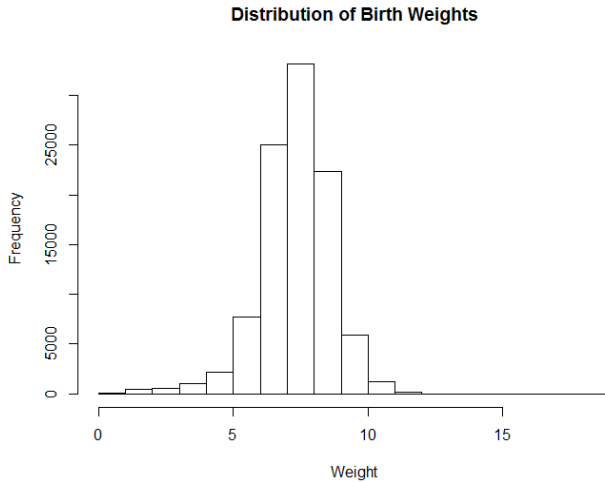


Figure 5.3: Distribution of Birth Weights.

The code applies the empirical cumulative distribution function, which is similar to computing the running sum and dividing by the total number of instances in the data set. One of the key benefits of CDF plots is that you can directly read percentile values from the visualization. Both types of visualizations can be generated using SQL and then plotting the results in a tool like Excel, but histograms are usually computationally less expensive, since they do not require analytic functions.

Another useful trick to explore when investigating data sets is transforming features, such as applying a log transform. This is useful when a data set is not normally distributed. For example, session data is usually a single-sided distribution with a long tail, and applying a log transform often provides a better summarization of the data when plotted.

```
hist(df$gestation_weeks, main = "Gestation Weeks",  
     xlab = "Weeks")  
hist(log(df$gestation_weeks), main = "Gestation  
     Weeks (Log Transform)", xlab = "log(Weeks)")
```

The code above shows how to transform the gestation weeks feature with a log transform. This is useful for distributions with long



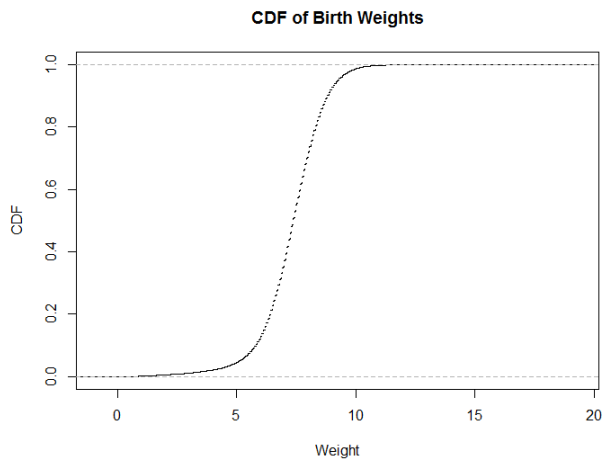


Figure 5.4: The same data, plotted as a CDF.

tails, and only positive values. The code generates the following two plots, which show similar results. For this example the transformation was not useful, but it's a trick worth exploring when you need to fit data from a skewed distribution to something closer to a normal distribution. You can also apply a square root transformation to generate useful plots.

An alternative to CDFs that conveys information about percentiles directly is box plots. Box plots show the 25th, 50th, and 75th percentiles, as well as interquartile ranges and outliers. This type of

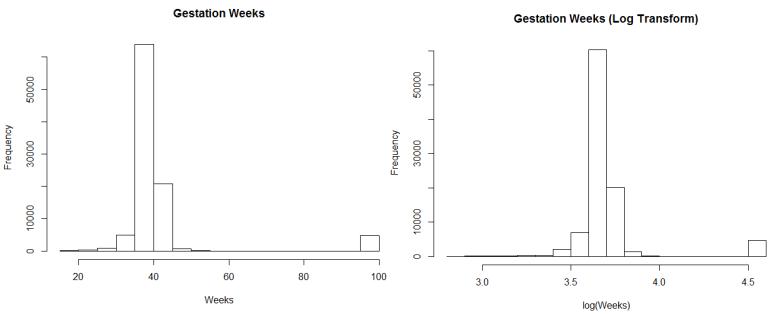


Figure 5.5: Log Transforming Gestation Weeks.

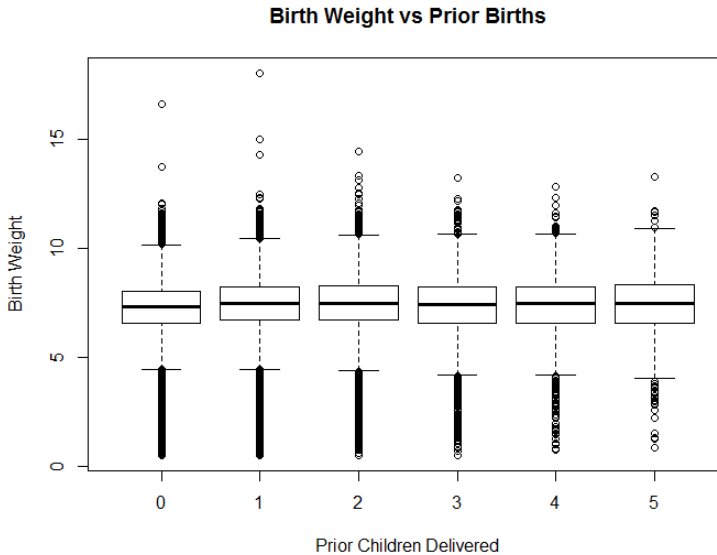


Figure 5.6: Birth weight based on number of previous children.

plot is often useful when comparing median values across different partitions of data.

```
sample <- df[df$ever_born <= 6, ]
sample$ever_born <- sample$ever_born - 1
boxplot(weight_pounds~ever_born,data=sample,
        main="Birth Weight vs Prior Births",
        xlab="Prior Children", ylab="Birth Weight")
```

An example of creating a box plot in R is shown in the code snippet above. This code creates a plot of different quartiles ranges, based on the number of previous children delivered by the mother. There's no strong pattern indicated by this plot, but it does look like the first child delivered has a slightly lower weight than the following children delivered by a mother.

Another type of useful visualization is scatter plots, which compare the values of two different features in a data set. Scatter plots are useful for showing whether two features are strongly or weakly correlated. The R code below shows how to plot a comparison of

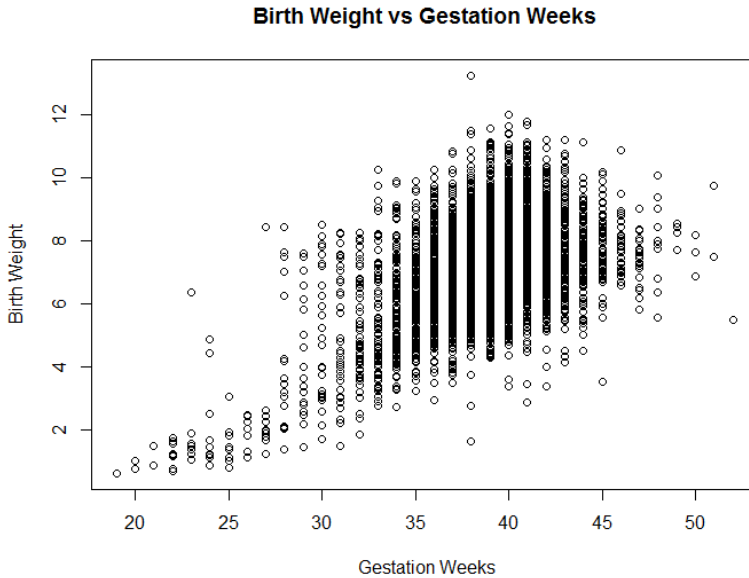


Figure 5.7: A comparison of the gestation period and birth weight.

the gestation period and birth weight. I've filtered out values with gestations periods longer than 90 weeks, because these are suspect.

```
sample <- df[1:10000, ]
sample <- sample[sample$gestation_weeks < 90, ]
plot(sample$gestation_weeks, sample$weight_pounds,
     main = "Birth Weight vs Gestation Weeks",
     xlab= " Gestation Weeks", ylab = "Birth Weight")
```

The results are shown in the following visualization. It's clear from this plot that significantly shorter gestation periods result in lower birth weights, but it's unclear how strong the correlation is for periods longer than 35 weeks. Overall, the features have an R-squared value of 0.25 for the sample data set. In order to determine which features are correlated with birth weights, we'll need to use different methods discussed in the next section.

## 5.3 Correlation Analysis

It's useful to know if certain features are predictive of other values in a data set. For example, if we know that users that use a certain product feature are retained longer than other users, it provides useful insight into product development. Correlation analysis helps to understand which features are correlated within a data set. Each feature is compared with all other features in a data set. However, often the goal is to understand only the correlation between a single feature, rather than a complete matrix of comparisons.

It's also important to avoid coming to strong conclusions based on only correlation analysis, because it's not possible to establish causality. Usually the more dedicated users will explore more of the options provided by an application, so trying to drive users to perform a specific action is not going to necessarily increase retention. For example, it's common to see improved retention for users that add social connections within an application, but driving users to add a friend in an application might not result in the desired outcome of longer user retention. It's best to experiment using controlled methodologies, which I'll discuss in a later chapter.

R has a built in function for performing correlation analysis. The `cor` function computes the correlation between all of the different columns in a data frame, using a specified methodology. The pearson method is useful when dealing with continuous values, and the spearman method is useful when using discrete values, such as survey data. An example of computing a correlation matrix is shown in the R snippet below.

```
res <- cor(df, method = "pearson", use = "complete.obs")
res <- ifelse(res > 0, res.5, -abs(res).5)
library(corrplot)
corrplot(res, type = "upper", order = "hclust")
```

To visualize the results, I've included the `corrplot` library, which creates a plot of the matrix. I noticed that the correlations between different features in the data set were weak, so I applied a square root transformation to make the results more readable. The resulting plot is shown in the figure below. As noted earlier, the gestation period is not the strongest factor in determining birth weight. The plurality is actually the strongest predictor of weight.

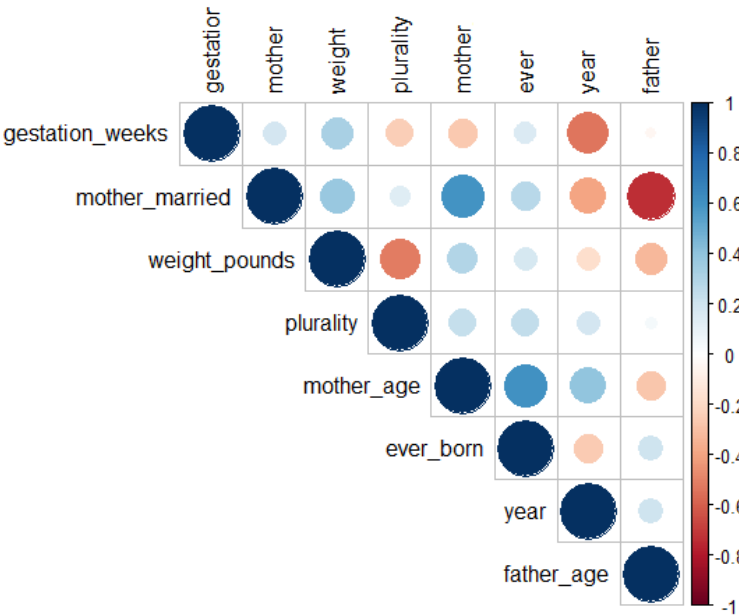


Figure 5.8: Correlation Matrix for the Natality Data Set (R-values have been square-rooted for visibility).

Similar to plotting histograms, it's useful to try different transformations of features when calculating correlations. For example, when predicting housing prices, the log transform of the square footage of a home is often a stronger indicator of the home value than the raw square footage value.

## 5.4 Feature Importance

Often the goal in exploring a data set is to determine which features are useful for predicting an outcome. This is similar to correlation analysis, but instead of evaluating the impact of each feature in isolation, the goal is to determine the significance of a single feature when including all other features. Correlation analysis is a way of understanding the marginal effect of a feature, and feature importance is a way of understanding the unique effect of a feature. This is useful for dealing with heavily correlated features, such as number of sessions versus total session length. Both are likely correlated with user retention, but one feature will be a stronger indicator than the other. LIME is a generalization of this type of analysis worth exploring.

While the next chapter will cover predictive modeling in more detail, we're going to use a simple linear model here to show how to perform feature analysis. Our goal for this example is to predict birth weight, which is a continuous feature. Given this problem specification, regression is a good fit for predicting outcomes. We can build a linear regression model, and then evaluate the importance of different features in the model to determine which factors are strong predictors of the outcome. The following code example shows how to build a linear regression model in R, and to measure the importance of features using different methods.

```
library(relaimpo)
fit <- lm(weight_pounds ~ . , data = df)
boot <- boot.relimp(fit, b = 10,
  type = c("lmg", "first", "last", "pratt"),
  rank = TRUE, diff = TRUE, rela = TRUE)
booteval.relimp(boot)
plot(booteval.relimp(boot, sort=TRUE))
```

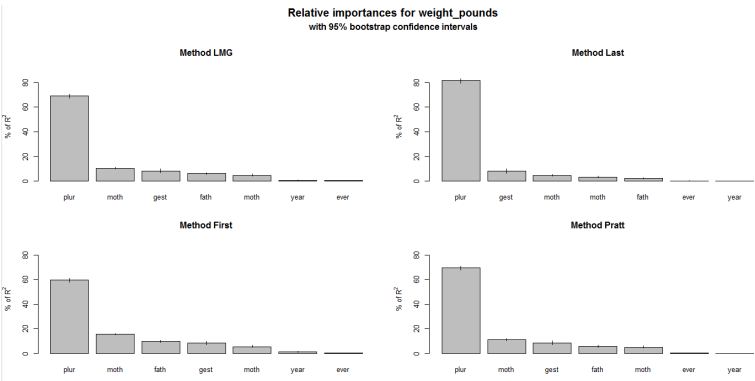


Figure 5.9: Feature Weighting for the Natality data set.

The results of this script are shown in the plots below. Different approaches are used to weight the features, and all of these methods show that plurality is the strongest feature in determining birth weight. The script first fits a linear regression model by specifying weight\_points as the target value, and then uses the relative importance library to determine which features are significant in determining the outcome.

5.5 Conclusion

Exploratory data analysis is one of the key competencies of a data scientist at a startup. You should be able to dig into a new data set and determine how to improve your product based on the results. EDA is a way of understanding the shape of a data set, exploring correlations within the data, and determining if there’s a signal for modeling an outcome based on the different features.

Exploratory analysis often involves some lightweight modeling to determine the importance of different features within a data set. In the next chapter, we’ll dig into predictive modeling, which focuses on estimating outcomes based on a number of different features.





## Chapter 6

# Predictive Modeling

Machine learning can be used to make predictions about the future. You provide a model with a collection of training instances, fit the model on this data set, and then apply the model to new instances to make predictions. Predictive modeling is useful for startups, because you can make products that adapt based on expected user behavior. For example, if a viewer consistently watches the same broadcaster on a streaming service, the application can load that channel on application startup. Predictive models can also be used to build data products, such as a recommendation system that could recommend new broadcasters to the viewer.

This chapter provides a light introduction to predictive modeling with machine learning. I'll discuss the different types of prediction problems and introduce some of the commonly used approaches, present approaches for building models using open tools and scripting languages, and provide an applied example of clustering. The goal for this chapter isn't to provide an in-depth understanding of specific methods, but to show how a variety of tools can be used to quickly prototype different types of models.

### 6.1 Types of Predictive Models

Machine learning models typically fall into two categories: supervised learning and unsupervised learning. For supervised problems,

the data being used to fit a model has specified labels, or target variables. For example, if the goal is to identify which users in a mobile game are likely to become purchasers, we can use transaction data from past users as labels, where 1 means a paid user and 0 means a free user. The label is used as input to the supervised algorithm to provide feedback when fitting the model to a training data set. Classification and regression algorithms are two types of supervised learning. In a classification task, the goal is to predict the likelihood of an outcome, such as whether or not a mobile game user will make a purchase. For regression, the goal is to predict a continuous variable, such as the price of a home given a description of different features.

For unsupervised problems, no explicit labels are provided for training a model. The most common type of unsupervised learning method is clustering, which infers labels by forming groups of different instances in a data set. Clustering is useful for answering segmentation questions, such as what are the different archetypes of users that a product should support.

There are two other types of machine learning models that I won't discuss here: semi-supervised learning and reinforcement learning. Semi-supervised learning is a process that identifies target labels as part of the training process, and is often implemented with autoencoders in deep learning. Reinforcement learning is a model that is updated based on a reward policy, where the actions taken by a model provide positive and negative feedback signals and are used to update the model.

For a startup, you're likely going to get started with classification and regression models, which are often referred to as classic, or shallow machine learning problems. There's a wide variety of different approaches that can be used. Some common approaches for classification are logistic regression, naive bayes, decision trees, and ensemble methods such as random forests and XGBoost. Common approaches for regression include many of the same approaches as classification, but linear regression is used in place of logistic regression. Support vector machines were popular back when I was in grad school a decade ago, but now XGBoost seems to be the king of shallow learning problems.

It's important to know how different algorithms are implemented, because if you want to ship a predictive model as part of a product, it needs to be reliable and scalable. Generally, eager models are

preferred over lazy models when shipping products. Eager models are approaches that generate a ruleset as part of the training process, such as the coefficients in a linear regression model, while a lazy model generates the rule set at run time. For example, a nearest neighbor (k-NN) model is a lazy approach. Lazy methods are often useful for building online learning systems, where the model is frequently updated with new data while deployed, but may have scalability issues.

How the performance of a predictive model is evaluated depends on the type of problem being performed. For example, metrics such as mean absolute error (MAE), root-mean squared error (RMSE), and correlation coefficients are useful for evaluate regression models, while ROC area under the curve (AUC), precision, recall, and lift are useful for classification problems.

## 6.2 Training a Classification Model

This section presents a few different approaches that can be used to build a classification model. We'll use the same data set as the past chapter on EDA, but instead of predicting birth weights in the Natality data set, we'll attempt to predict which pregnancies will result in twins instead of singletons.

To start, we'll need to pull a data set locally that we can use as input to different tools. The R code below shows how to sample 100k pregnancies and save the data frame to a CSV. This query is similar to the one from the past chapter, but I've included additional constraints in the where clause to avoid pulling records with missing (NA) values.

```
library(bigrquery)
project <- "your_project_id"
options(stringsAsFactors = FALSE)

sql <- "SELECT year, mother_age
      ,father_age, gestation_weeks
      ,case when ever_born > 0 then ever_born
        else 0 end as ever_born
      ,case when mother_married then 1
```

```
        else 0 end as mother_married
    ,weight_pounds
    ,case when plurality = 2 then 1 else 0 end as label
FROM `bigquery-public-data.samples.natality`
where plurality in (1, 2)
    and gestation_weeks between 1 and 90
    and weight_pounds between 1 and 20
order by rand()
LIMIT 100000"

df <- query_exec(sql, project = project,
                  use_legacy_sql = FALSE)
write.csv(df, "natality.csv", row.names = FALSE)
```

One of the challenges with this data set is that there are way more negative examples in this data set than there are positive examples. Only 2.4% of the pregnancies in the sampled data set have a label of '1', indicating twins. This means we'll need to use metrics other than accuracy in order to gauge the performance of different approaches. Accuracy is not a good metric for problems with a large class imbalance such as this one, because predicting a label of 0 for every record results in an accuracy of 97.6%. Instead, we'll use the AUC curve metric for evaluating different models, since it's useful for handling problems with imbalanced classes.

Another consideration when evaluating different models is using different training, test, and holdout data sets. The holdout data set is withheld until the end of the model training process, and used only once for evaluation. Training and test data sets can be used as frequently as necessary when building and tuning a model. Methods such as 10-fold cross validation are useful for building robust estimates of model performance. This is typically the approach I take when building models, but for the sake of brevity is not covered in all of the different examples below.

### 6.2.1 Weka

One of the tools that I like to use for exploratory analysis and evaluating different modeling algorithms is Weka<sup>1</sup>, which is implemented in Java and provides a GUI for exploring different models.

---

<sup>1</sup><https://www.cs.waikato.ac.nz/ml/weka/>

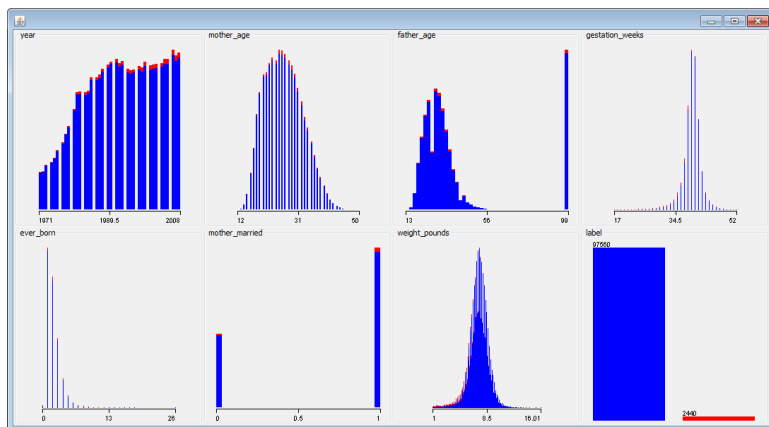


Figure 6.1: Visualizing different features in the data set with Weka.

It's a bit dated now, but I still find it quite useful for quickly digging into a data set and determining if there's much of a signal available for predicting an outcome.

The chart above shows visualizations of different features in the data set. The red data points represent the positive examples (twins), and the blue data points represent negative examples (singletons). For features with a strong signal, it's often possible to draw a vertical line that separates most of the red and blue data points. This isn't the case with this data set, and we'll need to combine different features to build a good classifier.

I used Weka to explore the following algorithms and to compute AUC metrics when using 10-fold cross validation:

- **Logistic:** 0.892
- **LogitBoost:** 0.908

The best performing algorithm out of the ones I explored was LogitBoost. This algorithm has a number of hyperparameters, such as number of iterations, that be be tuned to further improve the performance of the model. There may be other algorithms in Weka that work even better on this data set, but our initial exploration has resulted in promising results.

A visualization of the ROC curve for the logistic regression model is shown in the figure above. It's also possible to explore the im-

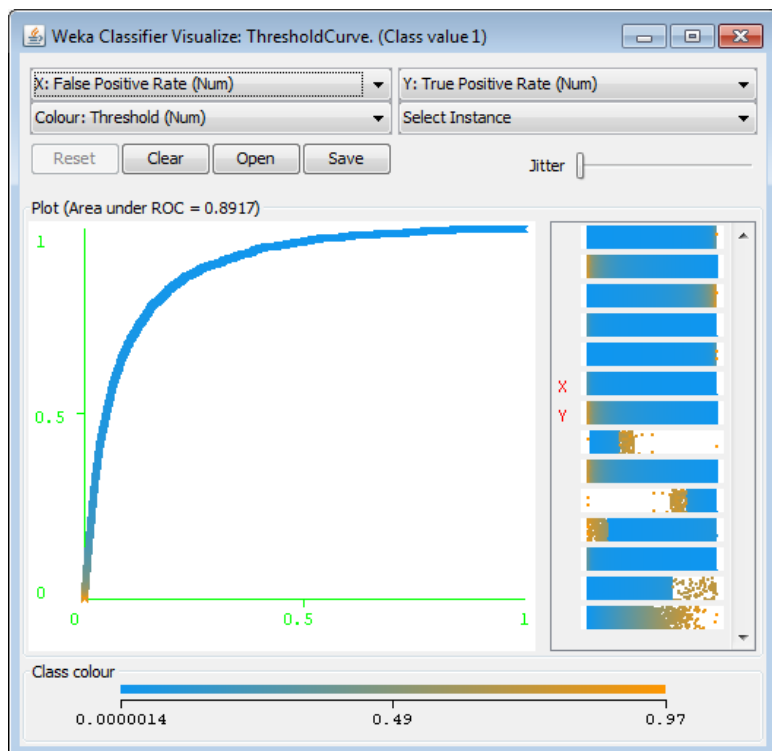


Figure 6.2: Visualizing the ROC Curve with Weka.

portance of different features in a logistic regression model with Weka. You can inspect the coefficients of the model directly. For example, `weight_pounds` has the highest coefficient value of 0.93. It's also possible to use the InfoGain attribute ranker to determine which features are most important for this classification task. Weka found that `weight_pounds` (0.0415) was the most influential feature, followed by `gestation_weeks` (0.0243).

Weka is usually not the best choice for productizing models, but it does provide a useful tool for exploring a wide variety of different algorithms.

### 6.2.2 BigML

Another tool that I've used in my startup experience is BigML<sup>2</sup>. This tool is similar to Weka in that it provides a GUI (web-based) for exploring different types of models without requiring any coding. The tool has fewer options than Weka, but has more recent models such as DeepNets.

The image above shows one of the feature importance tools provided by BigML. These tools are useful for understanding which features are useful in predicting an outcome. I explored two different models with BigML, resulting in the following AUC metrics:

- **Logistic:** 0.890
- **DeepNet:** 0.902

Instead of using 10-fold cross validation, I used a single 80/20 split of the data to evaluate the different models. The performance of the models in BigML was similar to Weka, but did not quite match the performance of LogitBoost.

In addition to plotting ROC curves, as shown above, BigML can plot other useful visualizations such as lift charts. BigML also provides useful classification metrics such as precision, recall, and F1 score.

---

<sup>2</sup><https://bigml.com/>



Figure 6.3: Evaluating Feature Importance in a Logistic Model with BigML.

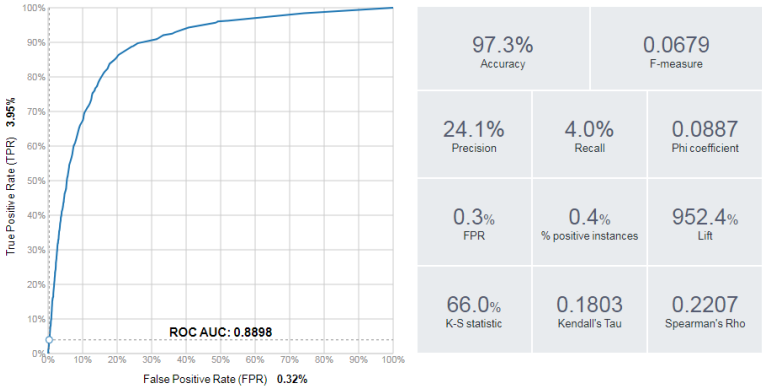


Figure 6.4: Evaluation Metrics provided by BigML.



### 6.2.3 R — Glmnet

We can implement the logistic regression model that we've already evaluated using the `glm` library in R. The generalized linear models function can be applied to logistic regression by specifying the binomial family as input. R code that loads the CSV and trains a logistic regression model is shown below.

```
df <- read.csv("Natality.csv")
fit <- glm(label ~ ., family=binomial(), data=df)
fit

library(Deducer)
rocplot(fit)
```

After fitting the model, the `fit` statement outputs the coefficients of the model. To evaluate the performance of the model, I used the `Deducer` library, which includes an `rocplot` function. For this basic model fitting approach, I did not perform any cross validation. The result was an AUC of 0.890 on the training data set.

To use regularization when fitting a logistic regression model in R, we can use the `glmnet` library, which provides lasso and ridge regression. An example of using this package to evaluate feature importance is shown in the code below:

```
library(glmnet)
x <- sparse.model.matrix(label ~ ., data = df)
y <- as.factor(df$label)

fit = glmnet(x, y, family = "binomial")
plot(fit, xvar = "dev", label = TRUE)
```

By default, the “least squares” model is used to fit the training data. The chart below shows how the coefficients of the model vary as additional factors are used as input to the model. Initially, only the `weight_pounds` features is used as input. Once this term begins getting penalized, around the value of -0.6, additional features are considered for the model.

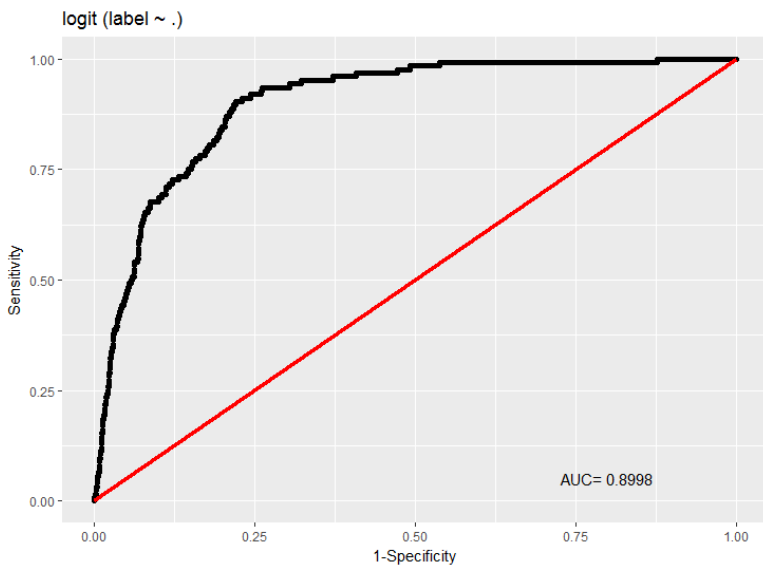


Figure 6.5: ROC Curve for the logistic regression model in R.

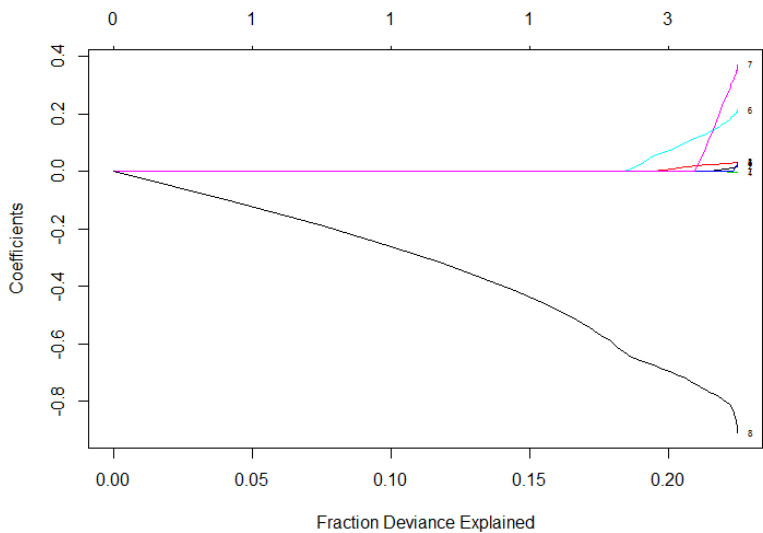


Figure 6.6: Feature weights based on different lambda values for glmnet.

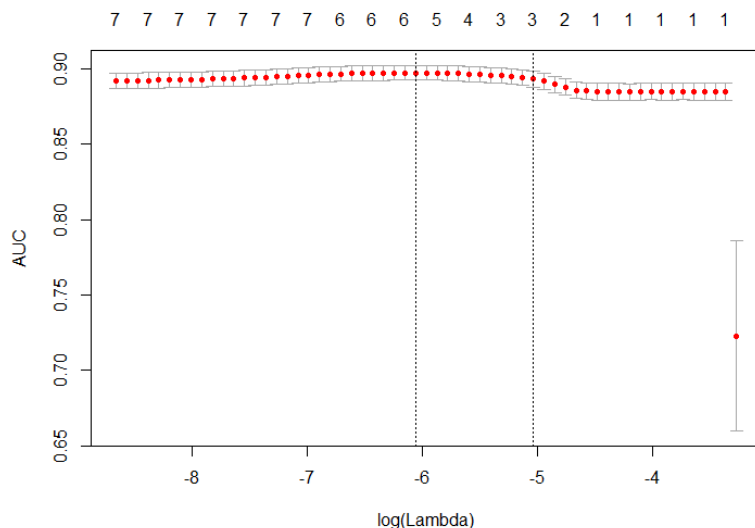


Figure 6.7: Performance (AUC) of the glmnet model based on different lambda values.

```
cvfit = cv.glmnet(x, y, family = "binomial",
                  type.measure = "auc")
cat(paste("ROC:", max(cvfit$cvlo)))
plot(cvfit)
```

The glmnet package provides a built-in cross validation feature that can be used to optimize for different metrics, such as AUC. The R code above shows how to train a logistic regression model using this feature, and plots the outcome in the figure shown below. The AUC metric for the regularized logistic regression model was 0.893.

### 6.2.4 Python — scikit-learn

Another tool that I wanted to cover in this section is scikit-learn, because it provides a standardized way of exploring the accuracy of different types of models. I've been focused on R for model fitting and EDA so far, but the Python tooling available through scikit-learn is pretty useful.

```
# load the data set
import pandas as pd
df = pd.read_csv('./Natality.csv')

# build a random forest classifier
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
x = df.drop('label', axis=1)
y = df['label']
rf.fit(x, y)

# evaluate the results
from sklearn.metrics import roc_curve, auc
false_positive_rate, true_positive_rate, _ =
    roc_curve(y, rf.predict(x))
roc_auc = auc(false_positive_rate, true_positive_rate)

# plot the curve
import matplotlib.pyplot as plt
plt.plot(false_positive_rate, true_positive_rate,
         'b', label='AUC = %0.2f'% roc_auc)
plt.legend(loc='lower right')
plt.show()
```

The Python code above shows how to read in a data frame using pandas, fit a random forest model using sklearn, evaluate the performance of the model, and plot the results, as shown in the figure below.

## 6.3 Clustering

One of the types of analysis that is useful for startups is understanding if there's different segments, or clusters of users. The general approach to this type of work is to first identify clusters in the data, assign labels to these clusters, and then assign labels to new records based on the labeled clusters. This section shows how to perform this type of process using data from the 2016 Federal Reserve Survey of Consumer Finances<sup>3</sup>.

---

<sup>3</sup><https://www.federalreserve.gov/econres/scfindex.htm>

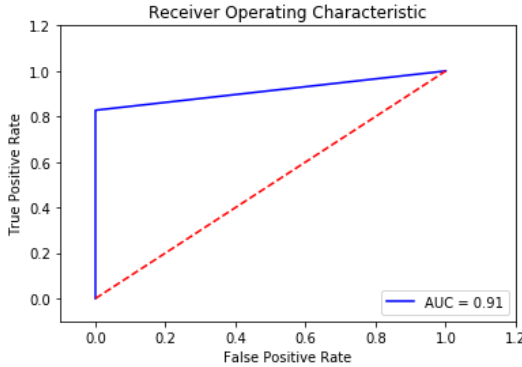


Figure 6.8: ROC Curve for the RF model in scikit-learn.

The survey data set provides a breakdown of assets for thousands of households in the US. The goal of this clustering exercise is to identify if there are different types of affluent households, with a net worth of \$1M+ USD. The complete code to load the data and perform the analysis is provided in the Jupyter notebook online<sup>4</sup>. Prior analysis with this data set is presented my prior blog post<sup>5</sup>.

For each of the surveyed households, we have a number of columns that specify how assets are allocated for the household, including residential and commercial real estate, business equity, retirement, and many other assets. The first thing we want to do is determine which assets have strong signals for clustering users. We can use PCA, and a factor map to accomplish this goal:

```
# filter on affluent households
affluent <- households[households$netWorth >= 1000000,]
cat(paste("Affluent: ", floor(sum(affluent$weight))))

# plot a Factor Map of assets
fviz_pca_var(PCA(affluent, graph = FALSE),
  col.var="contrib", gradient.cols = c("#00AFBB",
    "#E7B800", "#FC4E07"), repel = TRUE)+
  labs(title="Affluent Households - Assets Factor Map")
```

<sup>4</sup><https://bit.ly/2kp5ANb>

<sup>5</sup><https://bit.ly/2koWEHi>

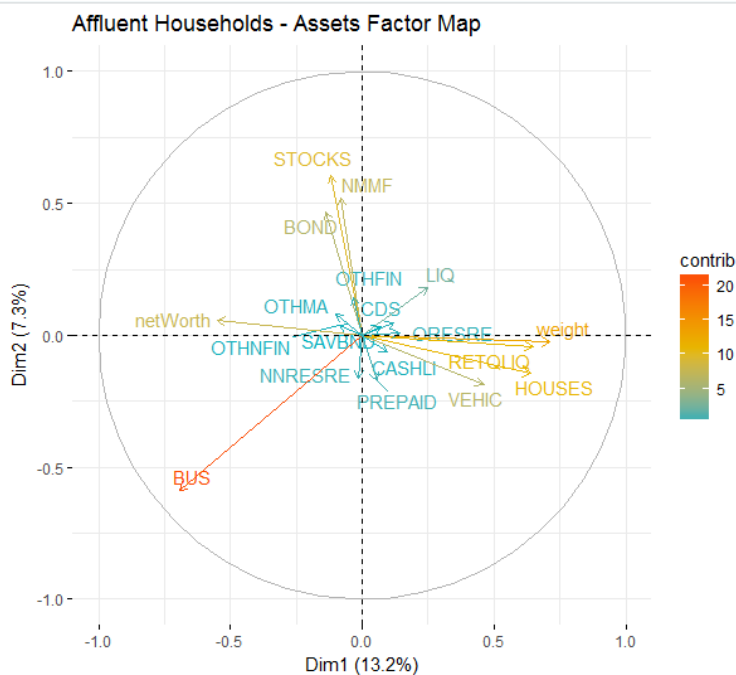


Figure 6.9: A factor map showing differences in asset allocations.

The results plotted below show that there are a few different assets groups that vary across affluent households. The most significant factor is business equity. Some other groupings of factors include investment assets (STOCKS, BONDS) and real estate assets/retirement funds.

### 6.3.1 How many clusters to use?

We've now shown signs that there are different types of millionaires, and that assets vary based on net worth segments. To understand how asset allocation differs by net worth segment, we can use cluster analysis. We first identify clusters in the affluent survey respondents, and then apply these labels to the overall population of survey respondents.

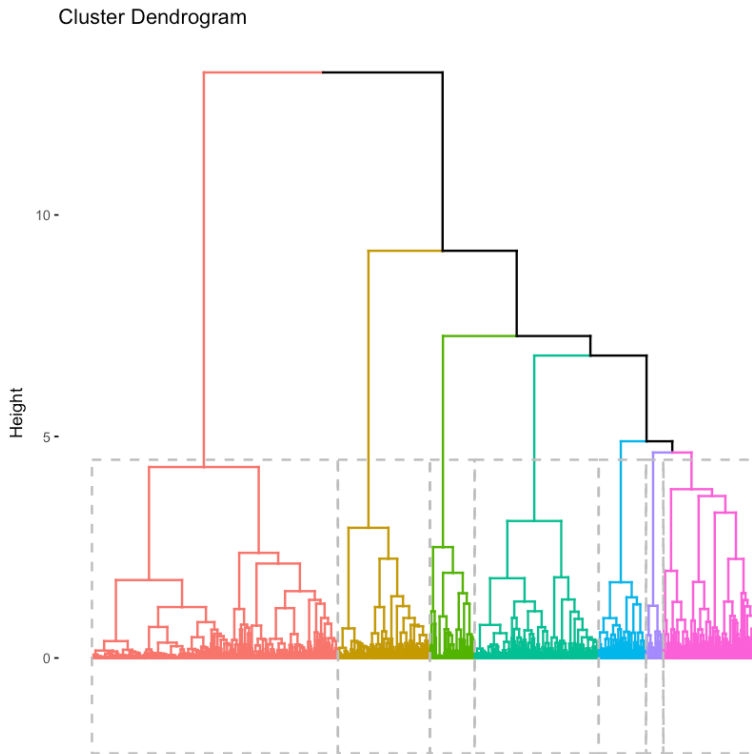


Figure 6.10: A hierarchical visualization of the cluster assignments.

```
k <- 7
res.hc <- eclust(
  households[sample(nrow(households), 1000), ],
  "hclust", k = k, graph = FALSE)
fviz_dend(res.hc, rect = TRUE, show_labels = FALSE)
```

To determine how many clusters to use, I created a cluster dendrogram using the code snippet above. The result is the figure shown below. I also varied the number of clusters,  $k$ , until we had the largest number of distinctly identifiable clusters.

If you'd prefer to take a quantitative approach, you can use the `fviz_nbclust` function, which computes the optimal number of clusters using a silhouette metric. For our analysis, I decided on 7.

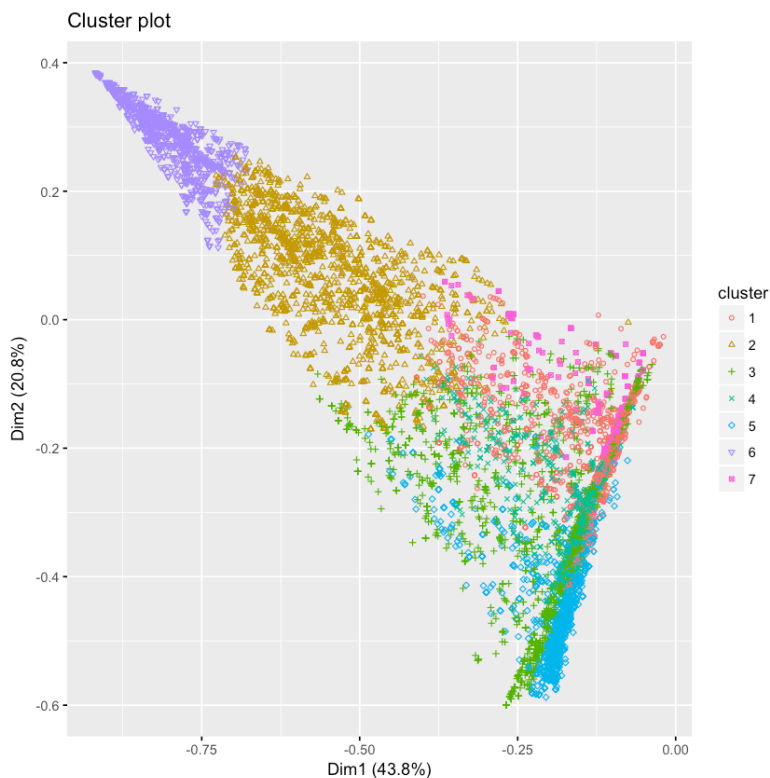


Figure 6.11: A visualization of the two-principal components identified by PCA.

```
clarax <- clara(affluent, k)
fviz_cluster(clarax, stand = FALSE,
              geom = "point", ellipse = F)
```

To cluster the affluent households into unique groupings, I used the CLARA algorithm. A visualization of the different clusters is shown below. The results are similar to PCA and the factor map approach discussed above.



### 6.3.2 Cluster Descriptions

Now that we've determined how many clusters to use, it's useful to inspect the clusters and assign qualitative labels based on the feature sets. The code snippet below shows how to compute the average feature values for the 7 different clusters.

```
groups <- clarax$clustering
results <- as.data.frame(
  t(aggregate(affluent, list(groups), mean)))
results[2:18,]
```

The results of this code block are shown below. Based on these results, we came up with the following cluster descriptions:

- **V1:** Stocks/Bonds — 31% of assets, followed by home and mutual funds
- **V2:** Diversified — 53% busequity, 10% home and 9% in other real estate
- **V3:** Residential Real Estate — 48% of assets
- **V4:** Mutual Funds — 50% of assets
- **V5:** Retirement — 48% of assets
- **V6:** Business Equity — 85% of assets
- **V7:** Commercial Real Estate — 59% of assets

With the exception of cluster V7, containing only 3% of the population, most of the clusters are relatively even in size. The second smallest cluster represents 12% of the population while the largest cluster represents 20%. You can use `table(groups)` to show the unweighted cluster population sizes.

### 6.3.3 Cluster Populations by Net Worth

The last step in this analysis is to apply the different cluster assignments to the overall population, and to group the populations by net worth segments. Since we trained the clusters on only affluent households, we need to use a classification algorithm to label the non-affluent households in the population. The code snippet below uses `knn` to accomplish this task. The remaining code blocks compute the number of households that are classified as each cluster, for each of the net worth segments.

	V1	V2	V3	V4	V5	V6	V7
LIQ	0.066	0.036	0.053	0.043	0.068	0.017	0.041
CDS	0.010	0.004	0.008	0.005	0.006	0.001	0.005
NMMF	0.118	0.063	0.027	0.500	0.031	0.013	0.021
STOCKS	0.261	0.036	0.034	0.038	0.026	0.010	0.024
BOND	0.046	0.010	0.007	0.015	0.003	0.002	0.003
RETQLIQ	0.097	0.061	0.102	0.109	0.480	0.016	0.045
SAVBND	0.001	0.001	0.001	0.001	0.002	0.000	0.000
CASHLI	0.011	0.009	0.010	0.010	0.012	0.006	0.004
OTHMA	0.043	0.039	0.119	0.023	0.020	0.006	0.021
OTHFIN	0.011	0.006	0.008	0.005	0.005	0.001	0.008
PREPAID	0.000	0.000	0.000	0.000	0.000	0.000	0.000
VEHIC	0.011	0.011	0.016	0.010	0.021	0.005	0.013
HOUSES	0.102	0.098	0.388	0.135	0.222	0.047	0.109
ORESRE	0.120	0.060	0.091	0.044	0.058	0.016	0.049
NNRESRE	0.022	0.033	0.037	0.010	0.010	0.008	0.590
BUS	0.070	0.526	0.092	0.048	0.029	0.848	0.064
OTHNFIN	0.011	0.007	0.008	0.003	0.006	0.004	0.004

Figure 6.12: Asset allocation amounts for the 7-identified clusters.

```

# assign all of the households to a cluster
groups <- knn(train = affluent, test = households,
  c1 = clarax$clustering, k = k, prob = T, use.all = T)

# figure out how many households are in each cluster
clusters <- data.frame(
  c1 = ifelse(groups == 1, weights, 0),
  ...
  c7 = ifelse(groups == 7, weights, 0)
)

# assign each household to a net worth cluster
nw <- floor(2*log10(nwHouseholds))/2
results <- as.data.frame(t(aggregate(clusters,
                                     list(nw),sum)))

# compute the number of households for each segment
results$V1 <- results$V1/sum(ifelse(nw==4,weights,0))
...
results$V11 <- results$V11/sum(ifelse(nw==9,weights,0))

# plot the results
plot <- plot_ly(results, x = ~10^Group.1, y = ~100*c1,
  type='scatter', mode = 'lines', name = "Stocks") %>%
  add_trace(y = ~100*c2, name = "Diversified") %>%
  ...
  add_trace(y = ~100*c7, name = "Commercial R.E.") %>%
  layout(yaxis = list(title = '% of Households'),
    xaxis=list(title = "Net Worth ($)", type = "log"),
    title="Cluster Populations by Net Worth")

```

The results of this process are shown in the figure below. The chart shows some obvious and some novel results: home ownership and retirement funds make up the majority of assets for non-affluent households, there is a relatively even mix of clusters around \$2M (excluding commercial real estate and business equity), and business equity dominates net worth for the ultra-wealthy households, followed by other investment assets.

For this clustering example, I explored survey data and identified seven different types of affluent households. I then used these clus-

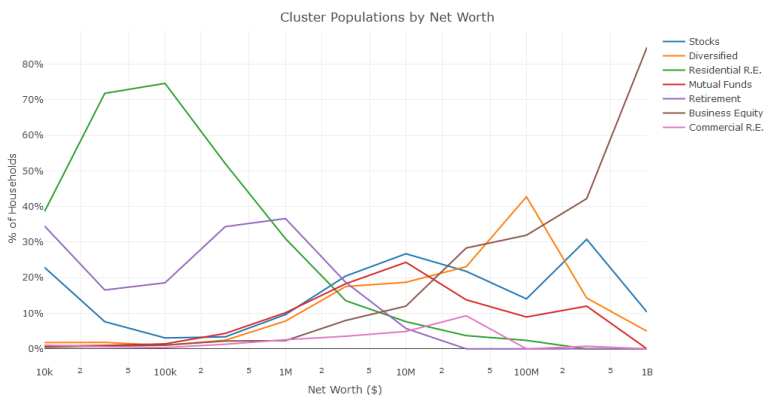


Figure 6.13: How the distribution of clusters varies based on Household Net Worth.

ters to assign labels to the remaining households. A similar approach could be used at a startup to assign segmentation labels to the user base.

## 6.4 Conclusion

Predictive modeling is an application of machine learning with a wide variety of tools that can be used to get started. One of the first things to consider when building a predictive model is determining the outcome that you're trying to predict, and establishing metrics that you'll use to measure success.

In this chapter, I showed four different approaches for building classification models for predicting twins during pregnancy. I showed how the GUI based tools Weka and BigML can be used to evaluate logistic regression models, ensemble models, and deep nets. I also scripting examples for performing logistic regression with regularization in R, and random forests in Python. I concluded the chapter with an example of clustering, which may be useful for performing segmentation tasks for a startup.

Independent of the approach being used to build a predictive model, it's important to be able to output a model specification as a result of your training process. This can be a list of coefficient weights for

a linear regression model, a list of nodes and weights for a random forest model, or a list of neuron weights and activations for a deep learning network. In the next chapter, I'll discuss how to scale predictive models to millions of users, and being able to represent a trained model as a specification is a prerequisite to production.



## Chapter 7

# Productizing Models

One the key ways that a data scientist can provide value to a startup is by building data products that can be used to improve products. Making the shift from model training to model deployment means learning a whole new set of tools for building production systems. Instead of just outputting a report or a specification of a model, productizing a model means that a data science team needs to support operational issues for maintaining a live system.

An approach I've used to ease this transition is managed tools, such as Google Dataflow, which provides a managed and scalable solution for putting models into production. Most of the approaches discussed in this chapter are using a serverless approach, because it's usually a better fit for startups than manually provisioning servers. Using tools like Dataflow also enables data scientists to work much closer with engineering teams, since it's possible to set up staging environments where portions of a data pipeline can be testing prior to deployment. Most early data scientists at a startup will likely be playing an ML engineer role as well, by building data products.

Rather than relying on an engineering team to translate a model specification to a production system, data scientists should have the tools needed to scale models. One of the ways I've accomplished this in the past is by using predictive model markup language (PMML) and Google's Cloud Dataflow. Here is the workflow I recommend for building and deploying models:

- Train offline models in R or Python
- Translate the models to PMML
- Use Dataflow jobs to ingest PMML models for production

This approach enables data scientists to work locally with sampled data sets for training models, and then use the resulting model specifications on the complete data set. The third step may take some initial support from an engineering team, but only needs to be set up once. Using this approach means that data scientists can use any predictive model supported by PMML, and leveraging the managed Dataflow service means that the team does not need to worry about maintaining infrastructure.

In this chapter, I'll discuss a few different ways of productizing models. First, I discuss how to train a model in R and output the specification to PMML. Next, I provide examples of two types of model deployments: batch and live. And finally, I'll discuss some custom approaches that I've seen teams use to productize models.

## 7.1 Building a Model Specification

To build a predictive model, we'll again use the Natality public data set. For this chapter, we'll build a linear regression model for predicting birth weights. The complete notebook for performing the model building and exporting process is available online<sup>1</sup>. The first part of the script downloads data from BigQuery and stores the results in a data frame.

```
library(bigrquery)
project <- "gcp_project_id"

sql <- "
  SELECT year, plurality, apgar_5min, mother_age
         ,father_age,  gestation_weeks, ever_born
         ,mother_married, weight_pounds
  FROM `bigquery-public-data.samples.natality`
  order by rand()
  LIMIT 10000"
```

---

<sup>1</sup><https://github.com/bgweber/WindfallData/blob/master/natality>



```
df <- query_exec(sql, project = project
                 , use_legacy_sql = FALSE)
```

Next, we train a linear regression model to predict the birth weight, and compute error metrics:

```
lm <- lm(weight_pounds ~ ., data = df)
summary(lm)
cor(df$weight_pounds, predict(lm, df))
mean(abs(df$weight_pounds - predict(lm, df)))
sqrt(mean(abs(df$weight_pounds - predict(lm, df))^2))
```

Which produces the following results:

- Correlation Coefficient: 0.335
- Mean Error: 0.928
- RMSE: 6.825

The model performance is quite weak, and other algorithms and features could be explored to improve it. Since the goal of this chapter is to focus on productizing a model, the trained model is sufficient.

The next step is to translate the trained model into PMML. The `r2pmml` R package and the `jpmml-r` tool make this process easy and support a wide range of different algorithms. The first library does a direct translation of a R model object to a PMML file, while the second library requires saving the model object to an RDS file and then running a command line tool. We used the first library to do the translation directly:

```
library(r2pmml)
r2pmml(lm, "natality.pmml")
```

This code generates the following pmml file. The PMML file format specifies the data fields to use for the model, the type of calculation to perform (regression), and the structure of the model. In this case, the structure of the model is a set of coefficients, which is defined as follows:

```

<RegressionTable intercept="7.5619">
  <NumericPredictor name="year"
                    coefficient="3.6683E-4"/>
  <NumericPredictor name="plurality"
                    coefficient="-2.0459"/>
  ...
  <NumericPredictor name="mother_married"
                    coefficient="0.2784"/>
</RegressionTable>

```

We now have a model specification that we are ready to productize and apply to our entire data set.

## 7.2 Batch Deployments

In a batch deployment, a model is applied to a large collection of records, and the results are saved for later use. This is different from live approaches which apply models to individual records in near real-time. A batch approach can be set up to run on a regular schedule, such as daily, or ad-hoc as needed.

### 7.2.1 SQL Query

The first approach I'll use to perform batch model deployment is one of the easiest approaches to take, because it uses BigQuery directly and does not require spinning up additional servers. This approach applies a model by encoding the model logic directly in a query. For example, we can apply the linear regression model specified in the PMML file as follows:

```

select weight_pounds as actual,
  + 11.82825946749738
  + year * -0.0015478882184680862
  + plurality * -2.1703912756511254
  + apgar_5min * -7.204416271249425E-4
  + mother_age * 0.011490472355621577
  + father_age * -0.0024906543152388157
  + gestation_weeks * 0.010845982465606988

```

```
+ ever_born * 0.010980856659668442
+ case when mother_married then 1 else 0 end*0.264942
  as predicted
from records
```

The result is that each record in the data set now has a predicted value, calculated based on the model specification. For this example, I manually translated the PMML file to a SQL query, but you could build a tool to perform this function. Since all of the data is already in BigQuery, this operation runs relatively quickly and is inexpensive to perform. It's also possible to validate a model against records with existing labels in SQL:

```
select sum(1) as records
,corr(actual, predicted) as Correlation
,avg(abs(actual - predicted)) as MAE
,avg(abs( (predicted - actual)/actual )) as Relative
from predictions
```

The results of this query show that our model has a mean-absolute error of 0.92 lbs, a correlation coefficient of 0.33, and a relative error of 15.8%. Using SQL is not limited to linear regression models, and can be applied to a wide range of different types of models, even Deep Nets. Here's how to modify the prior query to compute a logistic rather than linear regression:

```
1/(1 + exp(-1*(
  --regression calculation
))) as predicted
```

I've also used this approach in the past to deploy boosted models, such as AdaBoost. It's useful when the structure of the model is relatively simple, and you need the results of the model in a database.

### 7.2.2 DataFlow — BigQuery

If your model is more complex, Dataflow provides a great solution for deploying models. When using the Dataflow Java SDK, you

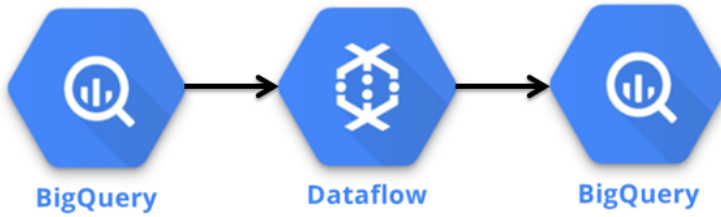


Figure 7.1: Components in the BigQuery Batch Deployment.

define an graph of operations to perform on a collection of objects, and the service will automatically provision hardware to scale up as necessary. In this case, our graph is a set of three operations: read the data from BigQuery, calculate the model prediction for every record, and write the results back to BigQuery. This pipeline generates the following Dataflow DAG:

I use IntelliJ IDEA for authoring and deploying Dataflow jobs. While setting up the Java environment is outside of the scope of this book, the pom file used for building the project is available on Github. It includes the following dependencies for the Dataflow sdk and the JPMML library:

```
<dependency>
  <groupId>com.google.cloud.dataflow</groupId>
  <artifactId>google-cloud-dataflow-java-sdk-all
    </artifactId>
  <version>2.2.0</version>
</dependency>
<dependency>
  <groupId>org.jpmml</groupId>
  <artifactId>pmml-evaluator</artifactId>
  <version>1.3.9</version>
</dependency>
```

As shown in the figure above, our data flow job consists of three steps that we'll cover in more detail. Before discussing these steps, we need to create the pipeline object:

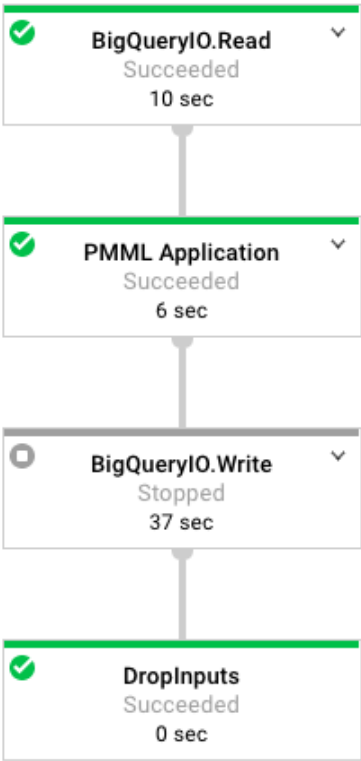


Figure 7.2: The Dataflow graph of operations used in this tutorial.

```
PmmlPipeline.Options options = PipelineOptionsFactory
    .fromArgs(args).withValidation()
    .as(PmmlPipeline.Options.class);
Pipeline pipeline = Pipeline.create(options);
```

We create a pipeline object, which defines the set of operations to apply to a collection of objects. In our case, the pipeline is operating on a collection of TableRow objects. We pass an options class as input to the pipeline class, which defines a set of runtime arguments for the dataflow job, such as the GCP temporary location to use for running the job.

The first step in the pipeline is reading data from the public BigQuery data set. The object returned from this step is a PCollection of TableRow objects. The feature query String defines the query to run, and we specify that we want to use standard SQL when running the query.

```
private static final String query =
    "SELECT year, plurality, ... weight_pounds\n" +
    "FROM `bigquery-public-data.samples.natality`";

pipeline.apply(BigQueryIO.read().fromQuery(query)
    .usingStandardSql().withoutResultFlattening())
```

The next step in the pipeline is to apply the model prediction to every record in the data set. We define a PTransform that loads the model specification and then applies a DoFn that performs the model calculation on each TableRow.

```
.apply("PMML Application", new PTransform
    <PCollection<TableRow>, PCollection<TableRow>>() {

model=new RegressionModelEvaluator(PMMLUtil.unmarshal(
    Resources.getResource("natality.pmml").openStream()));

return input.apply("To Predictions", ParDo.of(
    new DoFn<TableRow, TableRow>() {
        @ProcessElement
```

```

        public void processElement(ProcessContext c) {
            /* Apply Model */
        }
    }
}

```

The apply model code segment is shown below. It retrieves the TableRow to create an estimate for, creates a map of input fields for the pmml object, uses the model to estimate the birth weight, creates a new TableRow that stores the actual and predicted weights for the birth, and then adds this object to the output of this DoFn. To summarize, this apply step loads the model, defines a function to transform each of the records in the input collection, and creates an output collection of prediction objects.

```

TableRow row = c.element();
HashMap<FieldName, Double> inputs = new HashMap<>();
for (String key : row.keySet()) {
    if (!key.equals("weight_pounds")) {
        inputs.put(FieldName.create(key), Double
            .parseDouble(row.get(key).toString()));
    }
}

Double estimate =(Double)model.evaluate(inputs)
    .get(FieldName.create("weight_pounds"));

TableRow prediction = new TableRow();
prediction.set("actual_weight", Double.parseDouble(
    row.get("weight_pounds").toString()));
prediction.set("predicted_weight", estimate);
c.output(prediction);

```

The final step is to write the results back to BigQuery. Earlier in the class, we defined the schema to use when writing records back to BigQuery.

```

List<TableFieldSchema> fields = new ArrayList<>();
fields.add(new TableFieldSchema()
    .setName("actual_weight").setType("FLOAT64"));
fields.add(new TableFieldSchema()
    .setName("predicted_weight").setType("FLOAT64"));

```

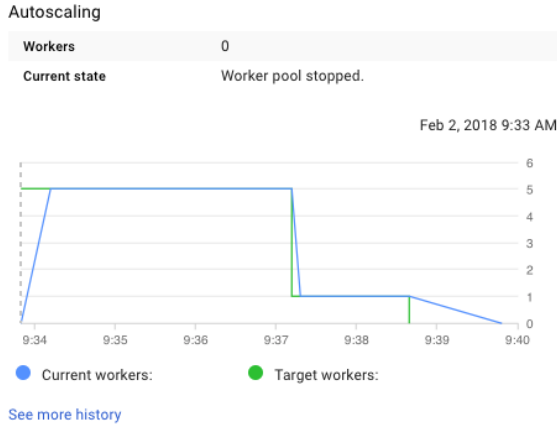


Figure 7.3: Autoscaling the Model Predicting Task.

```
TableSchema schema=new TableSchema().setFields(fields);

.apply(BigQueryIO.writeTableRows()
  .to(String.format("%s:%s.%s", proj, dataset, table))
  .withCreateDisposition(Write.CreateDisposition
    .CREATE_IF_NEEDED).withSchema(schema));

pipeline.run();
```

We now have a pipeline defined that we can run to create predictions for the entire data set. The full code listing for this class is available online<sup>2</sup>. Running this class spins up a Dataflow job that generates the DAG shown above, and will provision a number of GCE instances to complete the job. Here's an example of the autoscaling used to run this pipeline:

When the job completes, the output is a new table in our BigQuery project that stores the predicted and actual weights for all of the records in the natality data set. If we want to run a new model, we simply need to point to a new PMML file in the data flow job. All of the files needed to run the offline analysis and data flow project are available on Github.

<sup>2</sup><https://github.com/bgweber/StartupDataScience/tree/master/Productizing>



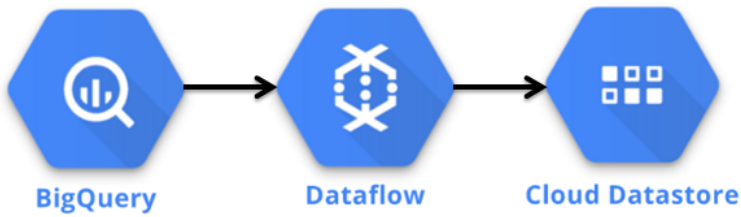


Figure 7.4: Components in the Datastore Batch Deployment.

### 7.2.3 DataFlow — DataStore

Usually the goal of deploying a model is making the results available to an endpoint, so that they can be used by an application. The past two approaches write the results to BigQuery, which isn't the best place to store data that needs to be used transactionally. Instead of writing the results to BigQuery, the data pipeline discussed in this section writes the results to Datastore, which can be used directly by a web service or application.

The pipeline for performing this task reuses the first two parts of the previous pipeline, which reads records from BigQuery and creates `TableRow` objects with model predictions. I've added two steps to the pipeline, which translate the `TableRow` objects to Datastore entities and write the results to Datastore:

```
.apply("To DS", ParDo.of(new DoFn<TableRow, Entity>() {  
  
    @ProcessElement  
    public void processElement(ProcessContext c) {  
        TableRow row = c.element();  
  
        // Create a lookup key for the record  
        String keyName = row.get("recordID").toString();  
        Key.Builder keyBuilder = Key.newBuilder();  
        Key.PathElement.Builder path = Key.PathElement.  
            newBuilder().setKind("Profile").setName(keyName);  
        keyBuilder.addPath(path);  
        Key key = keyBuilder.build();
```

```

// set the experiment group
String expGroup = Double.parseDouble(
    row.get("predicted_weight")
    .toString()) >= 8 ? "Control" : "Treatment";
Value value = Value.newBuilder()
    .setStringValue(expGroup).build();

// create an entity to save to Datastore
Entity.Builder builder = Entity
    .newBuilder().setKey(key);
builder.putProperties("Experiment", value);
c.output(builder.build());
}
}))
.apply(DatastoreIO.v1().write().withProjectId(proj));

```

Datastore is a NoSQL database that can be used directly by applications. To create entities for entry into Datastore, you need to create a key. In this example, I used a recordID which is a unique identifier generate for the birth record using the `row_number()` function in BigQuery. The key is used to store data about a profile, that can be retrieved using this key as a lookup. The second step in this operation is assigning the record to an control or treatment group, based on the predicted birth weight. This type of approach could be useful for a mobile game, where users with a high likelihood of making a purchase can be placed into an experiment group that provides a nudge to make a purchase. The last part of this snippet builds the entity object and then passes the results to Datastore.

The resulting entities stored in Datastore can be used in client applications or web services. The Java code snippet below shows how to retrieve a value from Datastore. For example, a web service could be used to check for offers to provide to a user at login, based on the output of a predictive model.

```

public static void main(String[] args) {
    Datastore datastore = DatastoreOptions
        .getDefaultInstance().getService();
    KeyFactory keyFactory = datastore
        .newKeyFactory().setKind("Profile");
}

```

```
// get a user profile
Entity profile = datastore.get(
    keyFactory.newKey("User101"));
System.out.println(profile.getString("Experiment"));
}
```

## 7.3 Live Deployments

The approaches we've covered so far have significant latency, and are not useful for creating real-time predictions, such as recommending new content for a user based on their current session activity. To build predictions in near real-time, we'll need to use different types of model deployments.

### 7.3.1 Web Service

One of the ways that you can provide real-time predictions for a model is by setting up a web service that calculates the output. Since we already discussed Jetty in the chapter on tracking data, we'll reuse it here to accomplish this task. In order for this approach to work, you need to specify all of the model inputs as part of the web request. Alternatively, you could retrieve values from a system like Datastore. An example of using Jetty and JPMML to implement a real-time model service is shown below:

```
public void handle(...) throws IOException{

    // load the PMML model
    final ModelEvaluator<RegressionModel> evaluator;
    try {
        evaluator = new RegressionModelEvaluator(
            PMMLUtil.unmarshal(Resources.getResource(
                "natality.pmml").openStream()));
    }
    catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

```
// create a map of inputs for the pmml model
HashMap<FieldName, Double> inputs = new HashMap<>();
for (String attribute : modelFeatures) {
    String value = baseRequest.getParameter(attribute);
    inputs.put(FieldName.create(attribute),
        Double.parseDouble(value));
}

// output the estimate
Double estimate =(Double)evaluator.evaluate(inputs)
    .get(FieldName.create("weight_pounds"));
response.setStatus(HttpServletResponse.SC_OK);
response.getWriter().println(
    "Prediction: " + estimate);
baseRequest.setHandled(true);
}
```

The code processes a message request, which contains parameters with the values to use as model inputs. The snippet first loads the PMML specification, creates an input map of features using the web request parameters, applies the evaluator to get a predicted value, and writes the result as output. The `modelFeatures` array contains the list of features specified in the PMML file. A request to the service can be made as follows:

```
http://localhost:8080/&year=2000&plurality=1
    &apgar_5min=0&mother_age=30&father_age=28
    &gestation_weeks=40&ever_born=1&married=1
```

The result of entering this URL in your browser is a web page which lists the predicted weight of 7.547 lbs. In practice, you'd probably want to use a message encoding, such as JSON, rather than passing raw parameters. This approach is simple and has relatively low latency, but does require managing services. Scaling up is easy, because each model application can be performed in isolation, and no state is updated after providing a prediction.

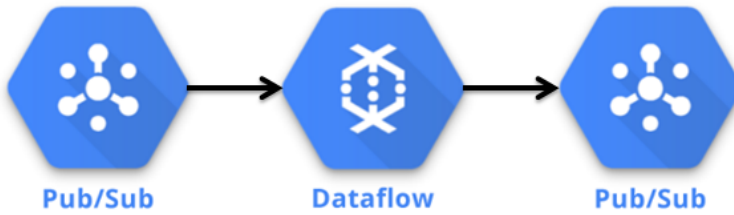


Figure 7.5: Components in the PubSub Live Model Deployment.

### 7.3.2 DataFlow - PubSub

It's also possible to use Dataflow in a streaming mode to provide live predictions. We used streaming Dataflow in the chapter on data pipelines in order to stream events to BigQuery and a downstream PubSub topic. We can use a similar approach for model predictions, using PubSub as a source and a destination for a DataFlow job. The Java code below shows how to set up a data pipeline that consumes messages from a PubSub topic, applies a model prediction, and passing the result as a message to an output PubSub topic. The code snippet excludes the process of loading the PMML file, which we already covered earlier in this chapter.

```
// Read messages from PubSub
PCollection<PubsubMessage> events = pipeline.apply(
    PubsubIO.readMessages().fromTopic(inboundTopic));

// create a DoFn for applying the PMML model
events.apply("To Predictions", ParDo.of(
    new DoFn<PubsubMessage, PubsubMessage>() {

@ProcessElement
public void processElement(ProcessContext c) {
    PubsubMessage row = c.element();
    // create a map of inputs for the pmml model
    HashMap<FieldName, Double> inputs = new HashMap<>();
    for (String key : row.getAttributeMap().keySet()) {
        if (!key.equals("weight_pounds")) {
            inputs.put(FieldName.create(key),
                Double.parseDouble(row.getAttribute(key)));
        }
    }
}
```

```
    }  
  }  
  
  // get the estimate  
  Double estimate = (Double)evaluator.evaluate(inputs)  
    .get(FieldName.create("weight_pounds"));  
  
  // create a message with the prediction  
  String message = "Prediction:" + estimate;  
  PubsubMessage msg = new PubsubMessage(  
    message.getBytes(), new HashMap());  
  c.output(msg);  
}}  
.apply(PubsubIO.writeMessages().to(outboundTopic));
```

The code reads a message from an incoming topic and then parses the different attributes from the message to use as feature inputs for the model. The result is saved in a new message that is passed to an outbound topic. Since the Dataflow job is set to run in streaming mode, the pipeline will process the messages in near real-time as they are received. The full code listing for this pipeline is available on Github<sup>3</sup>.

In practice, PubSub may have too much latency for this approach to be useful for directly handling web requests in an application. This type of approach is useful when a prediction needs to be passed to other components in a system. For example, it could be used to implement a user retention system, where mobile game users with a high churn likelihood are sent targeted emails.

### 7.3.3 Custom Engineering

Other approaches that are viable for providing live model deployments are Spark streaming, AWS Lambda, and Kinesis analytics.

Sometimes it's not possible for a data science team to build data products directly, because the system that needs to apply the model is owned by a different engineering team. For example, Electronic Arts used predictive models to improve matchmaking balance, and

---

<sup>3</sup><https://github.com/bgweber/StartupDataScience/tree/master/Productizing>

the team that built the model likely did not have direct access to the game servers executing the model. In a scenario like this, it's necessary to have a model specification that can be passed between the two teams. While PMML is an option here, custom model specifications and encodings are common in industry.

I've also seen this process breakdown in the past, when a data science team needs to work with a remote engineering team. If a model needs to be translated, say from a Python notebook to Go code, it's possible for mistakes to occur during translation, the process can be slow, and it may not be possible to make model changes once deployed.

## 7.4 Conclusion

In order to provide value to a startup, data scientists should be capable of building data products that enable new features. This can be done in combination with an engineering team, or be a responsibility that lives solely with data science. My recommendation for startups is to use serverless technologies when building data products in order to reduce operational costs, and enable quicker product iteration.

This chapter presented different approaches for productizing models, ranging from encoding logic directly in a SQL query to building a web service to using different output components in a Dataflow task. The result of productizing a model is that predictions can now be used in products to build new features.

I also introduced the idea of segmenting users into different experiment groups based on a predicted outcome. In the next chapter, I'll discuss different experimentation methodologies that can be used including A/B testing and staged rollouts.





# Bibliography

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.7.8.