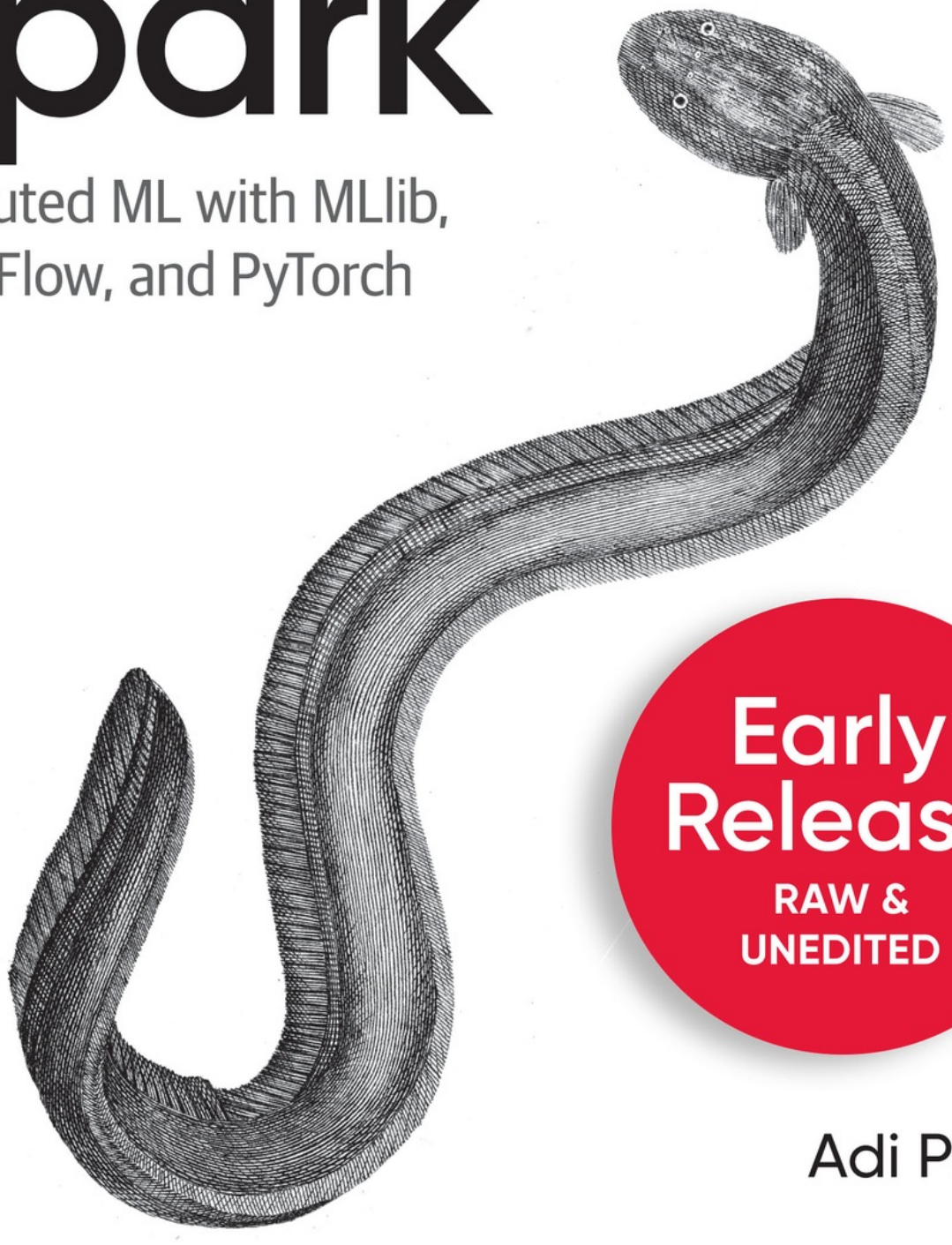
O'REILLY®

Scaling Machine Learning with Spark

Distributed ML with MLlib, TensorFlow, and PyTorch



Early Release

RAW & UNEDITED

Adi Polak

Scaling Machine Learning with Spark

Designing Distributed ML Platforms with PyTorch, TensorFlow, and MLLib

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Adi Polak

Machine Learning with Apache Spark

by Adi Polak

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Rebecca Novack and Jill Leonard

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

February 2023: First Edition

Revision History for the Early Release

- 2021-12-20: First Release
- 2022-04-15: Second Release
- 2022-07-27: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098106829 for release details.

978-1-098-10675-1

Chapter 1. Managing the ML Experiments Lifecycle with MLFlow

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. The GitHub repo is available now at https://github.com/adipolak/ml-with-apache-spark.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jleonard@oreilly.com.

In the field of machine learning development and data science, it is complex to record, integrate and build models collaboratively while experimenting with a combination of features, standardization techniques, and hyperparameters. In addition, it's a complete task to track experiments, reproduce results, package models for deployment, store and manage models without distorting the company's core goals

Due to the above-given reasons, it is obvious there is a need to evolve ML development and make it a more robust, predictable, standardized, and filtered software development. To go further with this, many organizations have started to build internal *machine learning platforms* to manage the ML lifecycle, and yet they still face challenges like most of the ML platforms often support only a small set of built-in algorithms of ML libraries which

are bound to each company's infrastructure. Moreover, these platforms are usually not open source, and users cannot easily leverage new ML libraries or share their work with others in the community.

Managing ML Experiment lifecycle can be also referred to as MLOps, which is a combination of machine learning, development, and operations. Machine learning is all about the experiment itself, training, tuning, and finding the optimal model. Development is about developing the pipelines and tools to integrate and take the machine learning model from the development / experimental stage to staging and production. And lastly, operations is about the CI/CD tools, monitoring, and managing the models at scale. Take a look at Figure 1-1, each step of the model lifecycle needs to be supported by the team in charge of the overall MLOps.

Figure 1-1. Machine Learning Model life cycle, from development to archiving

ML Experiments are ML pipeline code living in a repository, for example, Git branch. They contain *Code* + *Data* + *Model* and are an integral part of the R&D software lifecycle. For this book, we chose MLFlow because it is open-source, natively integrated with Apache Spark, and allows us to

manage the ML experiment by abstracting away complex functionality while allowing flexibility for collaboration and expanding to other tools.

What Is MLflow

MLflow is a platform that makes it simpler to manage the ML lifecycle. It allows the user and its team to have a standardized structure to manage data, including its experimentation, reproducibility, deployment, and a central model registry.

MLflow redefines feature organization and integrates the entire ML workflow. From overarching experiments to single-run trials to individual members of the team, MLflow allows you to track your process efficiently. Every hyperparameter tweak, every feature change, every possible metric can be recorded in one organized location using MLflow. It is the tool that keeps your team in sync and interconnected.

From a high-level approach, you can split it into two main components, the tracking server and the model registry as shown in Figure 1-2, the rest are supporting components of the flow. After the model is registered, the team can build the automated jobs and REST serving to move it downstream. Notice that the platform itself does not handle the model move from staging all the way to archive, which requires dedicated engineering work.

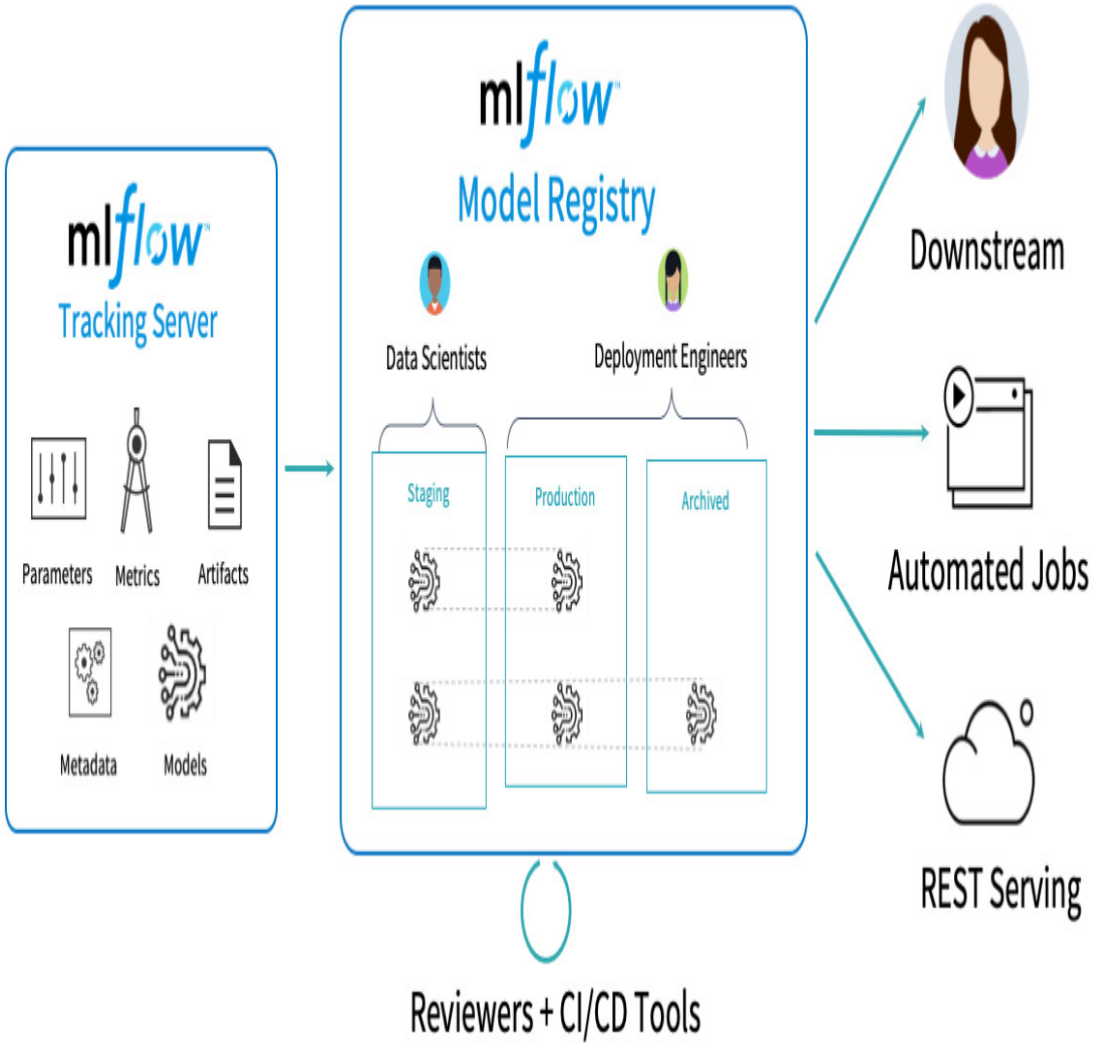


Figure 1-2. Databricks diagram of MLflow

Software Components of MLFlow Platform

To better understand how it works, let's take a look at its software components which consist of storage, a backend server, a front-end/UI component, API, and CLI.

Storage

MLflow provides support for connecting to multiple storage types and leveraging them to track the ml workflow. Storage can be a directory of files, databases, or any other notion of storage. It contains all the information about the experiments, multiple runs, parameters, etc.

Backend server

Responsible for communicating information between the database/storage, UI, SDK, and CLI to the rest of the components. Capture logs from experiments, etc.

Front-end

This is the UI side, where we can interact/track *experiments* and runs in a visual manner, as shown in figure 3-2.

API and CLI

The code we write and the command line we use with MLFlow.

We can interact with the MLFlow platform from the API, CLI, or its UI. Behind the scenes, it tracks all the information we provide it with. The API/CLI also generates dedicated directories that can be pushed to a git repository for better collaboration. In Figure 1-3 you can see a UI example of managing multiple runs within an experiment.

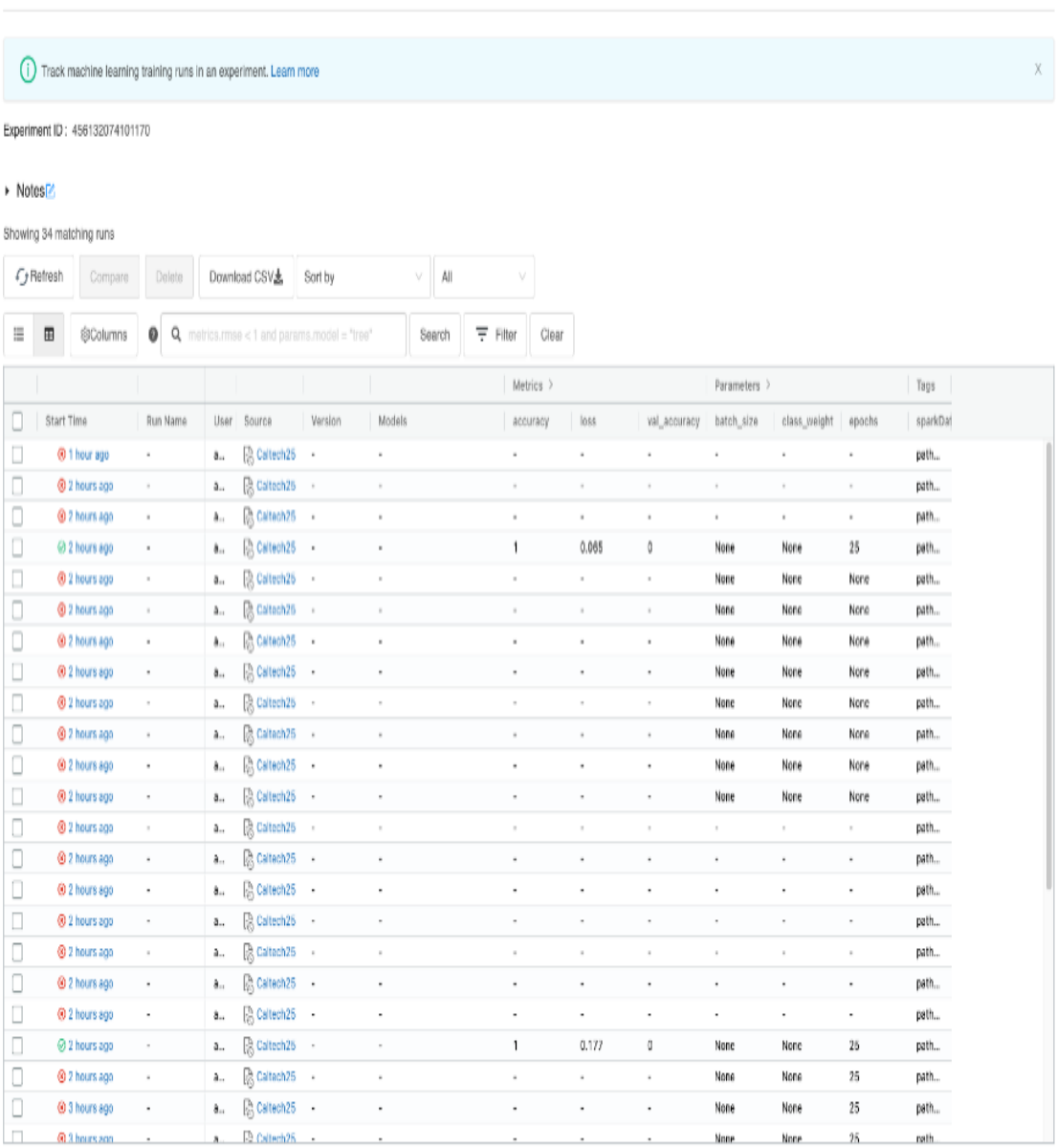


Figure 1-3. shows multiple runs and their tracking as part of an experiment

Use of MLflow in Different Organizations

As you can imagine, many parts come together to complete the puzzle of productionizing machine learning and building the lifecycle end-to-end. Because of this, MLFlow is usable in different organizations, and many technological personas are going to work with it. Some of these personas include:

Individual Users

As an individual without organizational responsibility, you can use MLFlow to track experiments locally on your machine, organize code in projects for future use, and output models that you can later test on fresh data. You can also use it for organizing your research work.

Data Science Teams

As a team, data scientists working on the same problem and experimenting with different models can compare their results by deploying MLflow Tracking. Anyone with access to the git repository can download and run another team member's model. Access MLflow UI to track the various parameters and get a better understanding of the experiment stage.

Organizations

From an organizational point of view, you can package training and data preparation steps for team collaboration and compare results from various teams working on the same task. For example, engineering teams can easily move workflows from R&D to staging to production. They can share projects, models, and results and run another team's code using MLflow Projects.

ML Engineers/ Developers

Often, data scientists will work together with ML/AI engineers. Using MLflow, data scientists and engineers can publish code to GitHub in the MLflow project format, making it easy for anyone to run their code. In addition, ML engineers can output models in the MLflow Model format to automatically support deployment using MLflow's built-in tools. ML

Engineering will also work together with the DevOps team to define the webhooks around MLFlow databases for moving the model between development stages, from development, validating, staging, production, and retiring.

Logic Components of MLflow

MLflow currently offers four logic components, *tracking*, for capturing the parameters and information related to the experiment, a *project* component that captures the whole project under a file system, *models, a* logic component that tracks the models and is saved within the project, and lastly, the *registry* abstract the storage that captures information related to the experiment and state of a model. Let's discuss them in detail.

MLflow Tracking

MLflow Tracking can be used in a standalone script (not bound to any specific framework) or notebook. It provides an API, UI, and a CLI for logging experiment parameters, code itself and its versions, ML metrics, and output files when running your machine learning code to later visualize them. It also enables you to log and query experiments using Python and some other APIs.

MLflow Tracking is based on the concept of runs end experiments, which are nothing but executions of some data science code. You can define how to record MLflow runs, it can be to local files, databases, or remotely to a tracking server. By default, the MLflow Python API logs run locally to files in `*mlruns*` directory.

Runs

Generally speaking, run is an execution of some piece of data science code that is logged and packaged as part of the experiment.

Experiments

An experiment can have many runs. This is the primary access control for the runs.

Runs are recorded for further use and tracked using MLflow Python, R, Java, and REST APIs from anywhere you run your code. You can use tracking capabilities in a standalone program, remote cloud machine, or a notebook. It tracks the project URI and source version in recorded runs as part of the MLflow Project. Which later allows you to query all the recorded runs using Tracking UI or the MLflow API.

How to use MLflow Tracking to record Runs

Let's assume we have a TensorFlow experiment that we want to run and track with MLFlow. Our first step is to import the *mlflow.tensorflow* library. After that, we can leverage auto logging capabilities or log the params and metrics programmatically.

To start the run, in Python we can use the `with` operator - with `mlflow.start_run()` – This API call singles a new run within an experiment, if one exists. If no experiment exists, it automatically creates a new one. You can create your own experiment with `mlflow.create_experiment()` API or the UI. Within the run, you develop your training code and leverage log_params and log_metric for tracking important information. At the end, you log your model and all necessary artifacts (more on that later). Check out the code snippet in Example 3-1 to better understand how the flow works:

Example 1-1. Example 3-1

```
[source,python]
import mlflow
import mlflow.tensorflow
# Enable mlflow autolog to log the metrics and model
mlflow.tensorflow.autolog() ❶
with mlflow.start_run(): ❷
# Log parameters (key-value pairs)
mlflow.log_param("num_dimensions", 8)❸
...
# Log any float metric; metrics can be updated throughout the run
mlflow.log_metric("alpha", 0.1)❹
```

```
... some machine learning training code
...
# Log artifacts (output files)
mlflow.log_artifact("model.pkl")❺
mlflow.log_model("ml_model_path") ❻
```

❶ Auto logs metrics, parameters, and artifacts
❷ Launch a new run under this experiment or launch an existing experiment if one is given to it.
❸ Log the partners we use in our ml algorithm.
❹ Log the metrics that can be updated throughout the run.
❺ Log the artifact of the project.
❻ Log the machine learning model itself.

As of writing this book, *mlflow.tensorflow.autolog()* is an experimental method in version 1.20.2. It uses the TensorFlow Callbacks mechanism to hook various functionality into the training, evaluating, and predicting phase.

Callbacks

> Callbacks can be passed to TensorFlow methods such as fit, evaluate, and predict in order to hook into the various stages of the model training and inference lifecycle. For example, you can leverage them to stop the training phase at the end of an *epoch* and cut costs on compute when training has reached the desired accuracy by configuring the parameters during the run. Callbacks in TensorFlow are part of Keras library `tf.keras.callbacks.Callback`, whereas, in PyTorch, they are part of `pytorch_lightning.callbacks'. PyTorch behaves differently as callbacks are often done with extension to the PyTorch library, where the most used one is the open-source PyTorch Lightning library backed by Grid.AI company.

Epoch

> Epoch indicated the number of passes machine learning training algorithms do on the entire training dataset. One epoch is one pass over the whole dataset. In each cycle of an epoch, you can access the logs and programmatically make decisions using Callbacks.

At the beginning of the training, MLFlow AutoLog tries to log all the configurations that are relevant for the training. Later, on each epoch cycle, it captures the logs metrics, including updating the overall training t. At the end of the training, it logs the model using `mlflow.keras.log_model` functionality. Hence it covers logging the whole lifecycle, where you can add additional parameters and artifacts you wish to log with it using the rich functionality like `mlflow.log_param’,`mlflow.log_metric`, `mlflow.log_artifact`, and more.

Autolog is also available for PyTorch, as demonstrated in Example 3-2.

Example 1-2. Example 3-2

```
[source, python]import mlflow.pytorch
# Auto log all MLflow entities
mlflow.pytorch.autolog()
```

However, it is recommended that you programmatically log the params, metrics, models, and artifacts as a general role. The same recommendation applies to working with Spark MLlib.

Log your dataset path and version

For experiment tracking, reproducibility, and collaboration, I advise logging the dataset path and version together with the model name and path, during the training phase. In the future, it will allow you to reproduce the model given the exact dataset when necessary and also differentiate between models that were trained with the same algorithm but different versions of the input.

For that, I recommend using the `mlflow.log_param()` functionality.

Example 1-3. Example 3-3

```
[source, python]
dataset_params = {"dataset_name": "twitter-accounts",
"dataset_version": 2.1}
# Log a batch of parameters
with mlflow.start_run():
  mlflow.log_params(dataset_params)
```

Another recommended option is using the run tags as part of the start_run.

```
[source, python]
with mlflow.start_run(tags=dataset_params):
```

MLFlow provides a flexible API. It is up to you to decide how to structure your experiment logs within the two recommended options. Figure 1-4 shows you how the two options look in the UI.

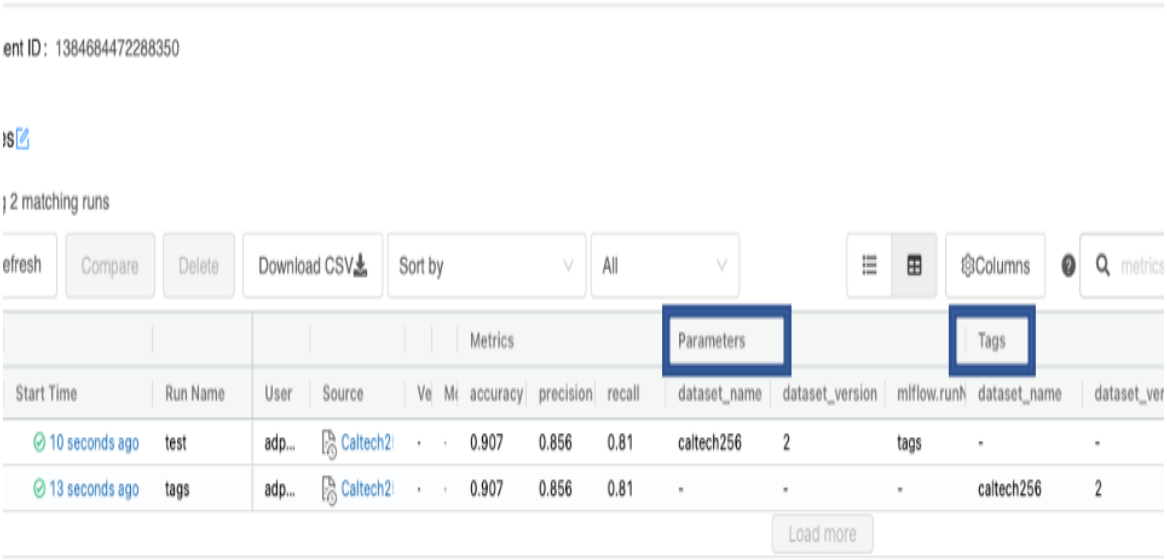


Figure 1-4. Parameters vs. Tags

> **WARNING**
>
> You may ask yourself, why not use the *mlflow.log_artifacts(local_dir, artifact_path=None)*. I advise against it since it copies everything in the given `local_dir` and writes it in the `artifact_path'. When I work with a large scale of data for training, I would like to avoid copying data without real necessity.

By now, you understand that the MLFlow Tracking component's responsibility is to log all the information for the project packaging.

MLflow Projects

MLflow Projects is a standard format for reusable and reproducible packaging, based primarily on conventions of data science code. It includes an API and command-line tools for executing projects and making it possible to chain together projects into workflows.

Every project is nothing but a directory with code or a Git repository and uses a descriptor file to specify its dependencies and how to run the code. MLflow Project is defined by a simple YAML file called `MLproject`.

From the MLFlow docs: MLflow currently supports the following project environments: Conda environment, Docker container environment, and system environment. By default, MLflow uses the system path to find and run the conda binary.

You can decide to use a different Conda installation by changing the MLFLOW_CONDA_HOME environment variable. On top of that, Docker containers allow you to capture non-Python dependencies such as Java libraries. The system environment is supplied at runtime and all the project's dependencies must be installed before project execution. The MLflow project format gives structure to share reproducible data science code as it is an open-source interface. MLflow Tracking is an essential component of MLFlow that enables you to track your work and supports reproducibility, extensibility, filtered and experimentation.

Running the experiment from the previous section will record the run in a folder with the following outline:

```
--- 58dc6db17fb5471a9a46d87506da983f
------- artifacts
------------ model
------------  MLmodel
------------ conda.yaml
------------ input_example.json
------------  model.pkl
------- meta.yaml
```

```
------- metrics
------------  training_score
------- params
------------  A
------------ …..
-------  tags
------------ mlflow.source.type
------------ mlflow.user
------- meta.yaml
```

The first line of output, `58dc6db17fb5471a9a46d87506da983f`, is the experiment identification also known as experiment `UUID`, 128-bit label universal unique identifier.

At the root of the directory, there is a YAML file named `meta.yaml`, which contains metadata about the experiment. With that, we already have extensive traceability information available for us and a solid foundation to continue the experiments. This folder contains all the information you need to reproduce the experiment.

Your project may have multiple entry points for invoking runs with named parameters. Since mlflow projects support Conda, you can specify code dependencies through a Conda environment leveraging MLFlow CLI:

```
[source,bash]mlflow run example/project -P alpha=0.5
```

Running the experiment using the `-P` param overwrites the parameter we described in the application code. `mlflow.log_metric("alpha", 0.1)<4>`. Which gives us the flexibility of changing parameters using the CLI tool.

MLflow Models

MLflow Models component is used for packaging ML models in multiple formats called *flavors*. Flavors deployment tools can be used to understand the model and write tools that work with models from any ML library without integrating each tool with each library. MLflow Model has several standard flavors that are built-in deployment tools support, such as a Python function flavor that describes how to run the model as a Python function.

`python_function` flavor can be deployed to a Docker-based REST server, cloud, and as a user-defined function. MLflow will also automatically remember which Project you output MLflow Models as artifacts using the Tracking API and run they came from.

As discussed above, every MLflow Model is a directory containing arbitrary files with an *MLmodel* file that can define multiple flavors a model can be used in.

Let's take a look at a `MLmodel` descriptor file that lists the flavors it can be used in.

```
[source, yaml]
{todo:}
```

In this example, the model can be used with tools that support either the `` `tesnorflow` `` or `python_function` model flavors.

MLflow Registry

The MLflow Registry abstracts a centralized storage and model state by providing a set of APIs and dedicated UI. It also allows us to manage the full lifecycle of an MLflow Model through CLI. You can think about it as a storage, for storing the model, an API for interacting with the model stage/status, querying, etc, a version for model lineage (keeping the history of a given model), and a UI for visualization. The registry is an essential part of MLflow components that are all connected and works together to provide a filtered platform for ML data regulation and structure. During experiments, we generate multiple models, each model receives a version, stage, status, and annotations. All of them can be handled by MLflow Model Registry.

After logging the model using the *log_model* API of the corresponding model flavors (as shown in code Example 3-4), it receives a status - *None*. Once a model has been logged, you can change the stage from "Tracking" state, with status None to "Register" state which has multiple statuses. Model Registry is also in charge of changing model status between *Staging, Production, and Archived*.

Registering Models during Runs

During the training cycle, we produce multiple models to pick the most suitable one.

Like you already read in the Tracking section, the models can be logged implicitly with autolog or explicitly with *log_model* model function.

Example 1-8. Example 3-4

```
[code, python]
mlflow.keras.log_model(keras_model=model,
registered_model_name='tfmodel', artifact_path="path",)
```

This code snippet provides the model with *STAGE_NONE ("None"),* which means the model is logged in the development stage.

Transitioning Model Stages

At a high-level machine learning development has 4 stages: development, staging, production, and archive. Since our work is tracked and packaged with MLFlow, it allows us to connect to CI/CD (continuous integration, continuous deployment) for transitioning the model between the stages. For that, we need to write dedicated scripts that listen on the model status and within an update, trigger a desired script. You can also leverage webhooks or other mechanisms of your flavor.

To promote the model from the development stage to staging in the database, use the MLFlow Client API:

```
[code,python]
client = MlflowClient()
Model_version = client.transition_model_version_stage(
    name='tfmodel',,
    version=3,
    stage="Staging"
)
```

MlflowClient is the python client to interact with the tracking server. *transition_model_version_stage* functionality allows us to update the model stage and invoke the desired CI/CD scripts. *stage* parameter accepted the following values: *Staging|Archived|Production|None*.

At this point, moving from *None* to staging, given a configured environment that tracks model status, will open a request to transition the model from *`None`* status to *`Staging`* status for integration testing on a Staging environment. We will discuss it in more detail in Chapter 10.

Understanding MLFlow Tracking Servers for Large Scale

From MLflow configurations, we can choose from multiple storages for different tasks. With MLFlow projects itself, you can decide where to save the experiment artifacts to. If you work locally on your machine, you can save it locally, or if you work from a notebook on the cloud or other remote server, you can save the project there. The MLflow Tracking Server (that helps us track the experiment information) has two different components for storage that we need to be familiar with: a backend store and an artifact store. See Figure 1-5.

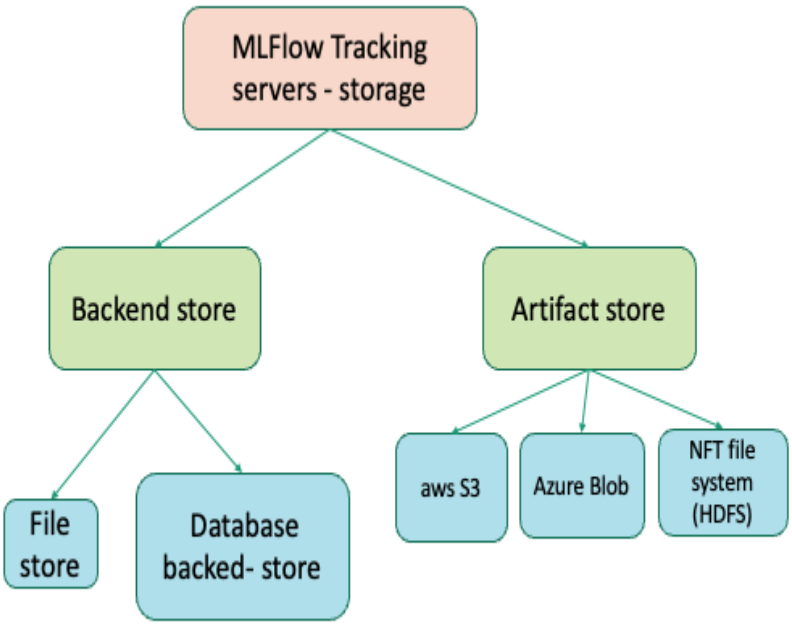


Figure 1-5. Storage servers options

The backend store is where MLflow Tracking Server stores *experiments* - *runs* metadata and params, metrics, and tags tables, see Figure 1-6 to understand the tables columns and relationship.

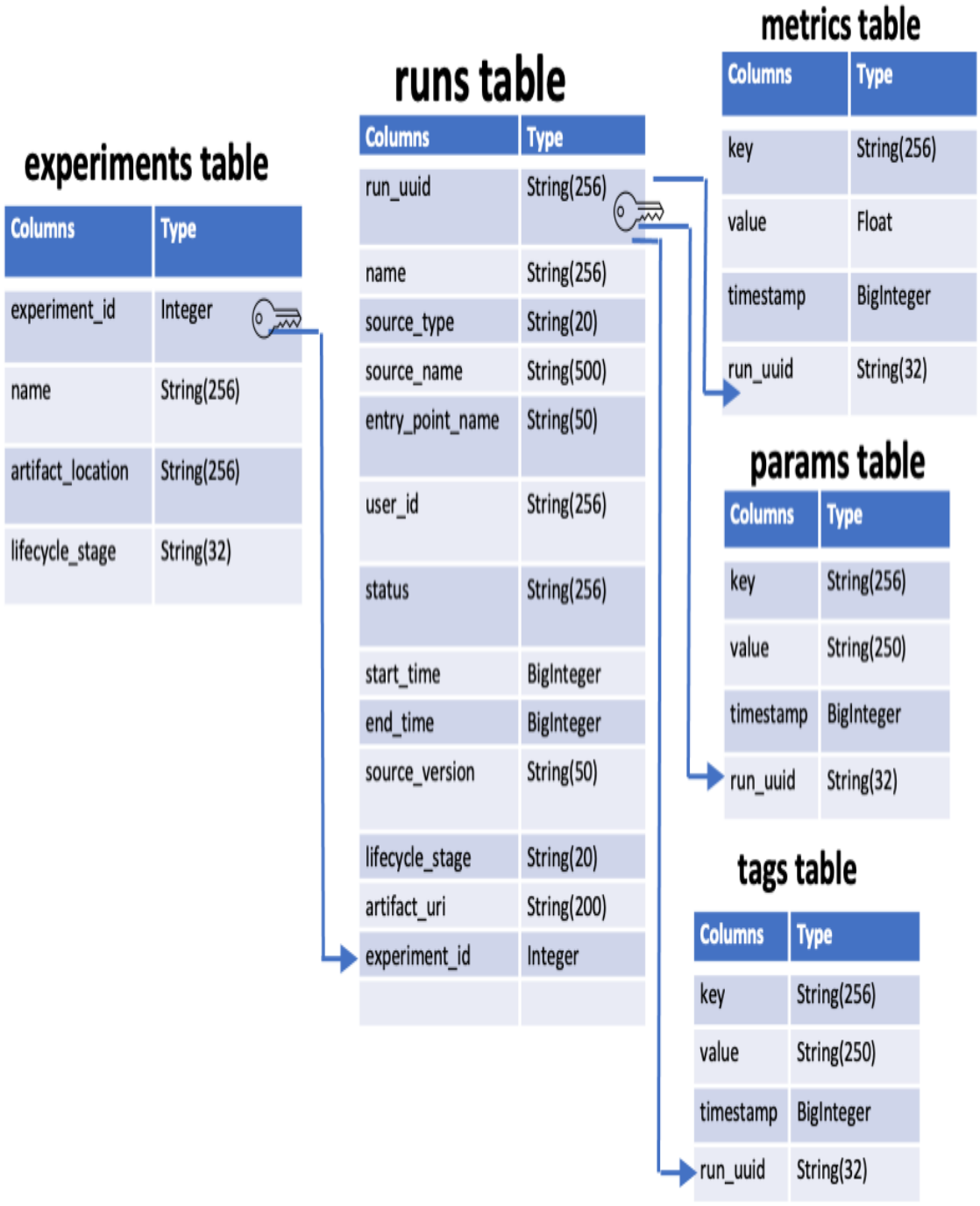
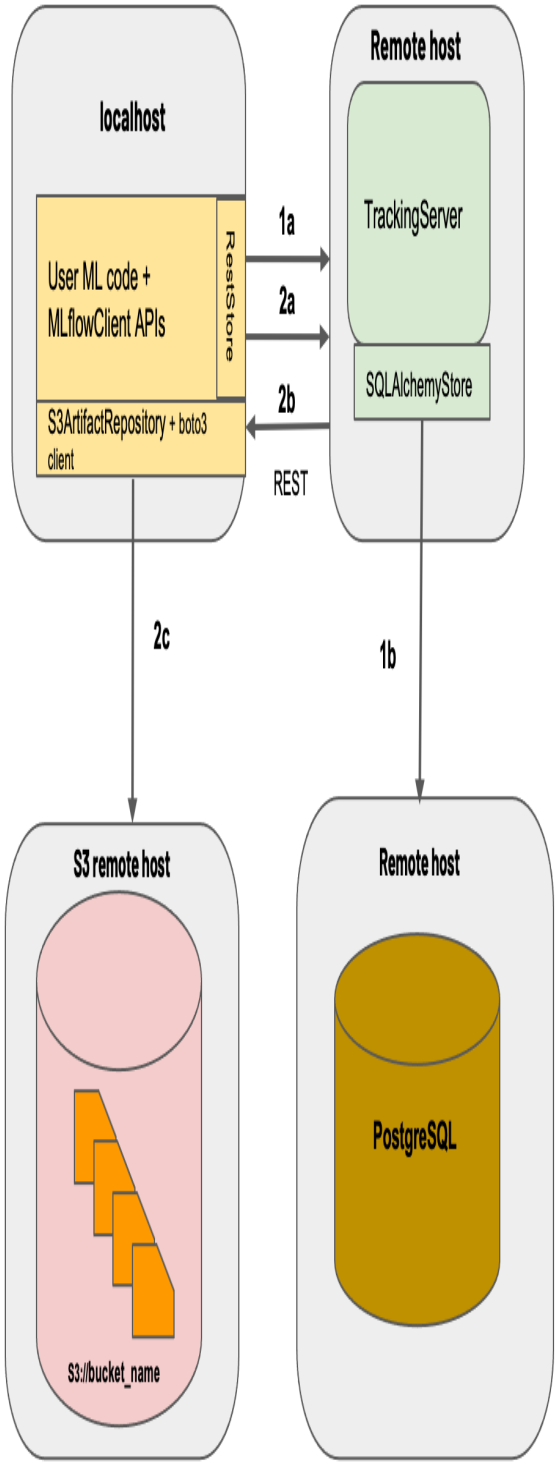


Figure 1-6. MLFlow tracking server SQL table hierarchy

The Backend store supports two types: file store and database-backed store. Let's discuss a distributed architectures option where the tracking server, backend store, and artifact store reside on remote hosts since distributed architectures often have better support for scale and team collaboration than running on *localhost*.



Scenario 4: `mlflow server --backend-store-uri postgresql://URI --default-artifact-root S3:/bucket_name --host remote_host`

Figure 1-7. Distributed Tracking servers example from docs

As you can see from Figure 1-7 , there could be multiple hosts. Where the main component is in a *localhost* and *Tracking Server itself*, the *Storage*, and the Tracking Server storage resides remotely.

Since the backend store can be a file store or a database backed-store, for scalable approach example, we would use PostgreSQL. PostgreSQL is an open source data-bases that has great support by the community, To connect with itm using python, MLflow uses SQLAlchemayStore tool.*SQLAlchemayStore* is the Python SQL toolkit and Object Relational Mapper that enables developers to interact directly with PostgreSQL using python.

For our scalable solution, the Artifact store location should be one that is suitable for large data since this is where we store our models that can be big. For that, we need to configure it using the artifact-root parameter.

ML Experiments Anti-Pattern

There is often a huge knowledge gap between developers and data scientists; while the first are experts in building software, the others are experts in building machine learning models and solving a business problem. That sometimes creates a dissonance between coding best practices and data science best practices, which leads to general software anti-pattens, especially when dealing with a large scale of data or a complex system. For example, if we were given a relatively small dataset - 100MB we can save the dataset together with the ML experiment code and output itself in a Git repository if we wish to do so. However, often within organization compliance and when we want to move through the stages of productionizing machine learning, we need to use scalable storage that holds to one or more requirements such as privacy, GDPR, access control, and others. For that, it’s often best to use scalable storage services. Those are often named object-store services such as Azure Blob, AWS S3, and others in the public cloud.

Summary

At this point, you have a better understanding of what MLFlow is and know how to manage experiments. Managing ML Pipelines lifecycle with data, code, and models is an important task and the differentiator between learning machine learning, to actually using it as part of an organization's R&D lifecycle. While MLFlow allows us to manage the whole experiment, we will discuss it again in chapter 10 about deployment. In the next chapter, we dive into the work with the data, and the machine learning pipeline itself, ingesting, preprocessing, engineering and so forth.

Chapter 2. Data Ingestion, Preprocessing, and Data Statistics

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. The GitHub repo is available now at https://github.com/adipolak/ml-with-apache-spark.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jleonard@oreilly.com.

You are most likely familiar with the phrase *Garbage in - Garbage out*, it captures well the notion that flawed, incorrect or nonesens data input will always produce faulty output. It emphasizes that data ingestion, preprocessing, and statistically understanding the *data* by exploring and preparing are going to impact over whole processes. Faulty data ingestion has a direct impact on the quality of the data, same goes to preprocessing. To get a feel of the data in hand and potentially the correctness of it, we are leveraging statistics.

Those are crucial steps, where data science, ML engineers and data engineers often spend significant time working on, researching and improving. Data statistics or getting a feel for the data are also known as descriptive statistics.

Before we start, let's understand the flow. Let's assume that at the start: Our Data resides on disk, database, or object store. We will follow the next steps to understand it:

1. Ingest it. Moving the data from the state it is in right now, and ingesting it into a DataFrame instance. This is also named deserialization of the data. More accurately, in Spark, this step defines the plan of how to deserialize the data. Transferring it from one mode to the other. This step provides us with a basic schema that we infer from the existing data.
2. Next step is: Preprocessing, marshaling the data to fit into a desired schema. If we load the data in String and we need it in a Float we will cast, change and tweek it to fit the desired schema. Think about the schema structure and number of data sources. Syncronizing data from multiple sources at a multi-terabyte scale is often an error-prone process and requires planning ahead.
3. Qualifying the data, using statistics to understand the data and how to work with it.

Step 2 and 3 can overlap, as we can decide to do more preprocessing to the data, depending on the statistics calculation from stage c.

Now that we have a better understanding of the steps, let dig into each one of them.

Data Ingestion with Spark

Apache Spark is generic enough to allow us to extend its API and develop dedicated connectors to any type of store for ingesting (and persisting/sinking/saving) data using the connector mechanism. Out of the box, it supports file formats such as *parquet, csv, binary, json, orc, image, etc*.

Spark also enables us to work with Batch and Streaming. For streaming it has an old API - named DStream or simply Streaming.The improved API is Structured Streaming.

Batch

> Batch data is an offline data, residing in a storage or a database, when we work with a batch, we will not get any fresh data to process. It is a fixed number that doesn’t change.

Structured Streaming

> Structured Streaming provides an api for distributed continuous processing of endless streams of data. Which allows us to leverage micro-batches of data processing and process multiple data records at a time.

In this chapter, we focus on batch processing with offline data, since it is the most common approach across varied machine learning use cases such as video production, financial modeling, drug discovery, genomic research, recommendation engines, and others. In chapter 10, we will demonstrate how to use Structured Streaming where we discuss serving the models with both batch and structure streaming.

Specifying Batch reading with a defined data format is done using the format functionality:

```
[source,python]
df = spark.read.format("image")
```

The class that enables this is the *DataFrameReader* that you can configure through its *options* API to define how to load the data and infer the schema if provided by your file format.

Working with images

An image file format is a file that can store data in an uncompressed, compressed, or a vector format. For example, JPEG is a compressed format and TIFF is an uncompressed format.

We save the Digital data in those formats in order to easily convert them for a computer display or a printer. This is the result of *Rasterization*. Rasterization[1] main task is converting the image data into a grid of pixels where each pixel has a number of bits that defines its color and transparency. Rasterizing an image file for a specific device takes into account the number of bits per pixel (the color depth) that the device is designed to handle. When we work with images, we need to attend the file format and understand if it's compressed or uncompressed. More on that in image compression and parquet.

In this chapter we use a Kaggle image dataset named Caltech256. Caltech256 dataset contains image files with JPEG compression formats. Our first step is to load them into a Spark DataFrame instance, for doing that, we can choose between two load format options: image or binary.

> **NOTE**
>
> During the load to DataFrame, Spark engine does not load the images into memory yet, it operates on a lazy mechanism. Lazy mechanism means that it creates a plan of how to load them. The plan contains information about the actual data such as table fields/columns, format, files addresses, etc.

Image format

Image format is MLlib dedicated functionality based on OpenCV types. *OpenCV* is a C/C++ based tool for computer vision workloads. MLlib functionality allows you to convert compressed images (jpeg,png,etc) into the OpenCV unique data format. Later, you can store them as a row in a DataFrame.

The supported decompress types are:

- CV_8U
- CV_8UC1
- CV_8UC3
- CV_8UC4

Where openCV propose these principals: *CV_<bit-depth>{U|S|F}C(<number_of_channels>)* where `U: unsigned`, `S:signed` and `F:floating point`

> **WARNING**
>
> As of Spark 3.1.1, the limit on image size is 1GB.

Since Spark is open source you can follow the image size support updates in *ImageSchema.scala* class which is a Scala code. In there you can notice the `decode` functionality of `assert(imageSize < 1e9, “image is too large”)` this tells us that the limit is 1GB.

For the exploration of relatively small files, the image format Spark provides is a fantastic way to start working with images and actually see the rendered output. However, for larger files, and general rhythm of work where there is no actual need of looking at the images themselves during the process, I advise you to use the Binary format, as it is more efficient and you will be able to process larger image files faster. On top of it, in the case of large images files (>= 1GB), binary format as a data source is the only way to process them.

Binary Format Data Source

Spark began supporting binary file data source with Spark 3.0.[2] With this change, it began reading binary files and converting them into a single record in a table. The record contains the raw content as BinaryType and a couple of metadata columns. Using Binary format to read the data, produces a DataFrame with the following columns:

- path: StringType
- modificationTime: TimestampType
- length: LongType
- content: BinaryType

For working with Caltech256 we load the data with the binary format as shown in the next code snippet:

```
[code,python]
from pyspark.sql.types import BinaryType
spark.sql("set spark.sql.files.ignoreCorruptFiles=true")
df = spark.read.format("binaryFile") \
.option("pathGlobFilter", "*.jpg") \
.option("recursiveFileLookup", "true") \
.load(files_path)
```

Since the data within our dataset is in a nested folders hierarchy as discussed in Chapter 2, *recursiveFileLookup* enables us to read the nested folders, while

pathGlobFilter option allows us to filter the files and read only the ones with *.jpg* extension.

Since Spark is based on a lazy evaluation mechanism, it allows us to save on computation cost, optimize queries and increase overall manageability. Just imagine that if Spark didn't have a lazy mechanism, every transformation would trigger a complete run over large datasets before it can ever be optimized with the next transformations and requests.

Lazy evaluation in Spark means that the execution will not start until an action is triggered. In Spark, the driver accumulates the various transformation requests and queries, optimizes them and only acts when there is a specific action request. This is why our reader from the previous code snippet, didn't yet load the data into the executors for computing.

Working with Tabular Data

Since Spark provides out-of-the-box connectors to various file format types, it makes working with it on tabular data pretty straightforward. For example, in our GitHub repository, under dataset you will find BotOrNot - Twitter accounts dataset, where the files are of a CSV format. With the connector functionality of - `.format("csv")`, or directly `.csv(file_path)` we can easily load it into a DataFrame instance. Do pay attention to the schema though. Even with the InferSchema option, CSV formats tend to define column format as String, often missing out on Integers, Boolean, etc. Hence in the beginning, our main job is to correct the columns data types, and adjust any string or nested string. For example, having a json string as part of the column in a CSV format file, would require writing dedicated code for handling. Notice that each Spark Data Sources connectors have unique properties that provide you with a set of options into how to deal with corrupted data. For example you can control the column pruning behavior by setting `spark.sql.csv.parser.columnPruning.enabled` to false if you don't wish to ignore columns or use true for the opposite. Besides that, you can leverage `mode` parameter to make pruning or not even more specific, with approaches such as `PERMISSIVE` which sets the field to `null`, `DROPMALFORMED` to ignore the whole record or `FAILFAST`, to throw an

exception upon processing a corrupted record. See code snippet below for example:

```
[source,python]
df = spark.read\
        .option("mode","FAILFAST")\
        .option("delimiter","\t")\
        .csv(some_file_path)
```

After loading the data and successfully passing the stage of deserializing the data into a DataFrame, is it time to preprocess it, I advise on saving it in a format that has a typed schema, with well defined column names and types, such as Parquet or Avro.

Preprocessing Data

Preprocessing the data is the art of fitting the data into the desired state, may it be a strongly typed schema, or a specific data type required by the algorithm.

Preprocessing vs Processing data

Those two can be extremely confusing to differentiate and we will try to explain how to refer to them. When discussing preprocessing, we refer to all the work we do before validating the dataset itself. This is done before getting a feel for the data and feature engineering. Whereas processing data is what we do during Feature Engineering and Descriptive Statistics. Those are interlocked procedures (See Figure 2-1) that we would likely repeat again and again until we reach the desired state. Spark provides us with all the tools to work with that data, either through the MLlib library or SQL APIs.

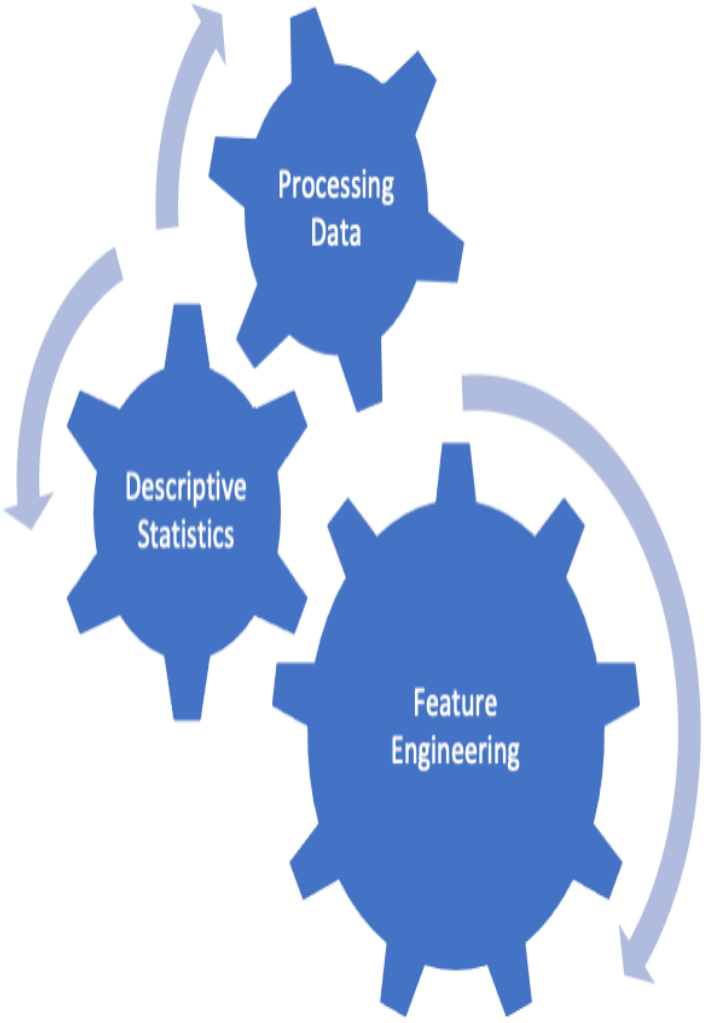


Figure 2-1. An interlocked procedure during machine learning

Why Preprocess the Data?

Preprocessing the data, is the part where we wrangle the data into the schema. It is a crucial step before we can even start exploring the data, let alone engineer new features. The reason we need to preprocess the data often is because ML algorithms have a dedicated input requirement such as specific

data structures and / or data types. For example, with Spark, there are dedicated MLlib types that the MLlib algorithms require as input. In some academic research papers, you might find this process named data marshaling.

For deciding about the types and how to process the data, let's breakdown the high-level requirements of the MLlib algorithms that will provide necessary requirements to determine which featurization techniques to use:

- *classification and/or regression algorithms*: here, you would want to process your data into one column of type vector (dense or sparse) with double or float values, often named features, but you have the flexibility to set the input column name later using the APIs input.
- *Recommendation algorithms:* for recommendation, you would want to have a userCol column with an integer representing the user id, itemCol with an integer representing the item id, and ratingCol column with double or float representing the item's rating by the user.
- *Unsupervised Approach:* When dealing with data that calls for using the unsupervised approach, you would often need one column of type vector to represent your features.

Data Structures

Data structures, such as *unstructured* or *semi-structured*, often require processing before it can be fully utilized. Looking at data structure, we use three definitions:

Structured

> Structured data has a high degree of organization, think of a table in a spreadsheet, It can be a comma-separated value file (.csv) or a table in a database. We can refer to it as *tabular* data.

Semi-structured

> Semi-structured data has some degree of organization, there might be some tags that separate elements and enforce hierarchies. Think of a JSON file where there are object notations. It might require preprocessing before we can use it for machine learning algorithms. It's a TXT file that has some

structure to it. Queries Twitter for extracting tweets returns a response in JSON format which is semi-structured data.

Unstructured

Unstructured data has no defined organization and no specific format. Think of .jpeg images, .mp4 video files, sounds files, etc. This data often requires major preprocessing before we can use it to build models. Unstructured data is most of the data that we create today.

MLlib Data Types

MLlib has its own dedicated data types input for machine learning algorithms. To work with Spark MLlib you would need to transform your columns to fit into the dedicated types of `Vector` and `Matrix`, this is why preprocessing and transforming data are processes that you will perform in an interlocked manner.

Under the hood, Spark uses the private objects *VectorUDT* and *MatrixUDF* which abstracts multiple sparse, dense vectors and Matrices types. Those objects allow easy interaction with *spark.sql.Dataset* functionality.

Vector

A vector object represents a numeric vector, you can think about it as an array, just like in Python, only here the index is of type *Integer* and value of type *Double*.

Matrix

Matrix object represents a numeric matrix; Matrix can be local to one machine or distributed across multiple machines. In its local version, indices type are of *Integer* and value is of type *Double*. In its distributed version, indices are of type *Long*, and values are of type *Double*. All Matrix types are represented by vectors.

> **NOTE**
>
> With Python tools, MLlib recognizes NumPy arrays and Python lists as dense vectors and SciPy's csc_matrix with a single column as a sparse vector.

Example: SparseVector vs. DenseVector

It's good to understand how those two are being represented, as you will encounter them in the documentation and your experiments.

How does a *DenseVector* look? Let's take a look at it:

```
[python, code]
Row(features=DenseVector([1.2, 543.5, 0.0, 0.0, 0.0, 1.0, 0.0]))]
```

A dense vector has an array of values and is considered less efficient in terms of memory consumption. In the above example, there are 7 values in the DenseVector. Behind the scenes, Spark decides on which vector to create for a given task.

What about a SparseVector?

SparseVector is an optimization of a DenseVector with empty/default values. Let's take a look at how the vector from the previous example, it translated into a SparseVector:

```
[python, code]
Row(features=SparseVector(7,{0:1.2, 1: 543.5,  5:1.0}))
```

The first number represents the size of the vector - 7, and the map - *{ }* represents the indices and their values. In this vector, there are only 3 values, in index 0 the value is 1.2, in index 2 the value is 543.5 and index 5 the value is 1. The rest of the indices are of value 0.0 as this is the default value.

Let's take a look at a bigger vector example:

```
[python, code]
[Row(features=SparseVector(50, {48: 9.9, 49: 6.7}))]
```

Like before, in this case, the vector size is 50 and we have only two values, 9.9 for index 48 and 6.7 for index 49.

SparseVecor can also look like this: *(50,[48,49],[9.9,6.7])* where the first number - 50 represents the size, the first array [48,49] represents the indices,

and the second array [9.9,6.7] represents the values for the indices accordingly. So the value for index 48 is 9.9 and for 49 is 6.7 . The rest of the vector indices have a value of 0.0.

Why do we need both?

In machine learning, some algorithms such as the *NaiveBayes classifier* work better on dense vectorized features and consequently might perform poorly given sparse vectorized features.

What can I do if that's the data I have?

First of all, this process will help you estimate the data, get a feel for it and start developing the code to work with it. Also, you can try collecting more data. You can also choose algorithms that perform better on sparse vectors. After all, this is part of the process of building machine learning models!

> **NOTE**
>
> There is a dedicated community project to improve PySpark and MLlib documentation which evolves and gets better every day. Make sure to bookmark it and use it. PySpark MLlib Doc.

Now you understand that there are dedicated data types for all sorts of algorithmic requests. Another example is a data type for supervised machine learning algorithms: *LabeledPoint* that represents features and labels of a data point. This is why preprocessing the data is a crucial step in the machine learning workflow.

Preprocessing with MLlib Transformers

Transformers are part of the Apache Spark MLlib library named `pyspark.ml.feature`. Its capabilities include Transformers, Extractors, and Selectors. Many of them are based on machine learning algorithms and statistical or mathematical computations. For preprocessing we will leverage the Transformers API. But you might find other APIs helpful as well in different situations.

They are algorithms that take a DataFrame as an input and output a new DataFrame with the desired columns. As a general approach, they are defined as translators of a given input to a corresponding output. They enable us to scale, convert or modify existing columns. We can split Transformers functionality into *text data transformers*, *categorical features transformers*, *continuous numerical transformers*, and *others*.

The tables would guide you on when to use each transformer. You should be aware that given the statistical nature of the transformers there might be APIs that would take longer to finish.

Working with Text Data

Text data often consists of documents that represent words, sentences or any form of free flowing text. This is inherently unstructured data which often has a noisy nature. *Noisy data* in Machine learning is irrelevant or meaningless data that might affect the machine learning and significantly impact the model performance. Examples for that would be stop words such as: *"a", "the", "is", "are"*. In MLlib, you can find a dedicated functionality just for extracting stop words and so much more! MLlib provides a rich functionality for manipulating text data input. With text, we want to ingest it and transform it into a format that can be easily fitted into machine learning algorithms. Most of the machine learning algorithms in MLlib, expect structured data as an input in tabular format with rows and columns etc. On top of that, for efficiency in data representations of memory consumption, we would hash the strings since `string` is considered more memory space costly than `int`, `float` or `boolean`. Before adding text data to your ML project, you first need to clean it up by using text data transformers. To learn about the common APIs and their usage, take a look at Table 4:1.

Table 2-1. Text Data transformers

API	Usage
Tokenizer	Maps a column of text into a list of words based on the regular expression \\s. This removes one whitespace or comma. Under the hood, tokenizer API is using java.lang split functionality.
RegexTokenizer	Splits the text based on your regex input with default regex \\ s + that removes multiple whitespaces and/or commas and other supported delimiters. It is more computationally heavy than Tokenizer since it uses scala.util.matching regex functionality. Regex should conform to Java Regular Expression syntax.
n-gram	Extract a sequence of n tokens given an integer n. Notice that the input column can only be an Array of Strings. To convert one text String into an Array of Strings, use the Tokenizer functionality first.
StopWordsRemover	Take a sequence of text and drop the default stop words. You can specify languages, case sensitivity and provide your own list of stop words.
HashingTF	One of the most used transformers, takes an array of string and generates hash instead. In many free-text scenarios, you will need to run the Tokenizer first.

Before we proceed, let's generate a synthetic dataset to better understand the functionality.

```
[source,python]
sentence_data_frame = spark.createDataFrame([
    (0, "Hi I think pyspark is cool ","happy"),
    (1, "All I want is a pyspark cluster","indifferent"),
    (2, "I finally understand how ML works","content"),
    (3, "Yet another sentence about pyspark and ML","indifferent"),
    (4, "Why didn't I know about mllib before","sad"),
```

```
    (5, "Yes, I can","happy")
], ["id", "sentence", "sentiment"])
```

Our dataset has 3 columns: row `id` of type `int`, `sentence` and `sentiment` of type `string`.

Transformation will include the following steps: Free Text → List of words. List of just words → List of meaningful words. Select meaningful values.

Ready? Set? Transform! Our first step is to transform the free text into a list of words, for that, we can use either `Tokenizer` or `RegexTokenizer` API.

```
[source,python]
from pyspark.ml.feature import Tokenizer

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
tokenized = tokenizer.transform(sentence_data_frame)
```

The `Tokenizer` is set to take the `sentence` column as an input and generate a new DataFrame, adding an output column named `words`. Notice that we used the functionality of `transform`. *Transformers* always have a `transform` function. Figure 2-2 shows our new DataFrame with the added words column:

```
+---+-----------------------------------+---------+-----------------------------------------+
|id |sentence                           |sentiment|words                                    |
+---+-----------------------------------+---------+-----------------------------------------+
|0  |Hi I think pyspark is cool         |happy    |[hi, i, think, pyspark, is, cool]        |
|1  |All I want is a pyspark cluster    |indifferent|[all, i, want, is, a, pyspark, cluster]|
|2  |I finally understand how ML works  |fulfill  |[i, finally, understand, how, ml, works] |
|3  |Yet another sentence about pyspark and ML|indifferent|[yet, another, sentence, about, pyspark, and, ml]|
|4  |Why didn't I know about mllib before|sad     |[why, didn't, i, know, about, mllib, before]|
|5  |Yes, I can                         |happy    |[yes,, i, can]                           |
+---+-----------------------------------+---------+-----------------------------------------+
```

Figure 2-2. New DataFrame with array of words column

The next step would be to remove stop words, words that are less likely to provide value in our machine learning process. For that we will use `StopWordsRemover`.

```
[source,python]from pyspark.ml.feature import StopWordsRemover

remover = StopWordsRemover(inputCol="words",
outputCol="meaningful_words")
meaningful_data_frame = remover.transform(tokenized)
# we use show function for educational purposes only, with large set
of data, we should avoid it.
meaningful_data_frame.select("words","meaningful_words").show(5,trunca
te=False)
```

```
+----------------------------------------+-------------------------------+
|words                                   |meaningful_words               |
+----------------------------------------+-------------------------------+
|[hi, i, think, pyspark, is, cool]       |[hi, think, pyspark, cool]     |
|[all, i, want, is, a, pyspark, cluster] |[want, pyspark, cluster]       |
|[i, finally, understand, how, ml, works]|[finally, understand, ml, works]|
|[yet, another, sentence, about, pyspark, and, ml]|[yet, another, sentence, pyspark, ml]|
|[why, didn't, i, know, about, mllib, before]|[know, mllib]              |
|[yes,, i, can]                          |[yes,]                         |
+----------------------------------------+-------------------------------+
```

Figure 2-3. New DataFrame with an array of meaningful words column

> **NOTE**
>
> Notice that the process of removing stopwords can be seen as part of the feature engineering section as it removes the noise from the words array. We decided to keep it as part of the preprocessing since often with NLP you will want to remove stop words at the beginning as it is known to be noisy data.

From Nominal categorical features to indexes

One of the strategies to speed up our machine learning process is to turn discrete categorical values presented in `string` format into a numerical form using indexes. It can be discrete or continuous numeric and depends on the machine learning models we plan to use. See Table 4-1 lists to better understand the most common APIs and their usage.

Table 2-2. Working with nominal categories

g
o
r
i
c
a
l
f
e
a
t
u
r
e
s

API	Usage
StringIndexer	Encode String columns into indexes, where the first one (starting at 0) is the most frequent value in the column and so on. Used for faster training with supervised data where the columns are the categories/labels.
IndexToString	Symmetrical to `StringIndexer`, often used to retrieve the label categories after the training process.
OneHotEncoder	Maps categorical features into a continuous feature that is often used in machine learning algorithms such as Logistic Regression.
VectorIndexer	Similar to StringIndexer. Takes a whole vector as an input and converts it into category indices.

The DataFrame we have generated also has a column representing the data `sentiment` category. Let's take a look at the Sentiment Categorical features in our DataFrame.

Our sentiment categories: Happy, Fulfilled, Sad and Indifferent. Let's turn them into indexes using `StringIndexer`.

```
[source, python]from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol="sentiment",
outputCol="categoryIndex")
indexed =
indexer.fit(meaningful_data_frame).transform(meaningful_data_frame)
indexed.show(5)
```

In this code snippet, we created a new `StringIndexer` instance that takes the `sentiment` column as an input and creates a new DataFrame with `categoryIndex` column. We call the `fit` functionality first, providing it with our DataFrame named `meaningful_data_frame`. This step is necessary for training the indexer before the transformation. At this step, it builds a map between indexes and categories by scanning the category column. This functionality is another preprocessing tool named *Estimator*. For more examples, refer to Chapter 6 where we discuss this in detail. After fitting the estimator, we can call `transform` to calculate the new indexes. Take a look at Figure 2-4 to understand the new dataframe with cateogyIndex:

id	sentence	sentiment	words	meaningful_words	categoryIndex
0	Hi I think pyspar...	happy	[hi, i, think, py...	[hi, think, pyspa...	0.0
1	All I want is a p...	indifferent	[all, i, want, is...	[want, pyspark, c...	1.0
2	I finally underst...	Fulfilled	[i, finally, unde...	[finally, underst...	2.0
3	Yet another sente...	indifferent	[yet, another, se...	[yet, another, se...	1.0
4	Why didn't I know...	sad	[why, didn't, i, ...	[know, mllib]	3.0
5	Yes, I can	happy	[yes,, i, can]	[yes,]	0.0

Figure 2-4. DataFrame with categoryIndex

The outcome of executing `StringIndexer` is a new column named - `categoryIndex` of type `double`.

Structuring Continuous Numeric Data

There are cases where we would want to structure our continuous numeric data. Structuring continuous data is providing a threshold, or multiple thresholds for taking an action or deciding on a classification. For example, as shown in Figure 4-5, when we have scores for a specific sentiment, and given the score is in between a defined range, we would take an action. Think about a customer satisfaction system, we would like our machine learning model to recommend an action that is based on customer sentiment score. Let's say our biggest customer has an "anger" score of 0.75, and our threshold for calling the customer is anger score above 0.7. In this instance, we would want to call them and discuss how we can improve their experience. The thresholds themselves can also be defined by using machine learning algorithms, or plain statistics. Going forward let's assume we have a DataFrame with a dedicated score for every sentiment. That score is a continuous number between [0,1] specifying the relevancy of the sentiment category. The business goal we want to achieve is going to determine if we need to provide a threshold/specific structure for the data for future recommendation.

sentence_id	happy	indifferent	Fulfilled	sad
0	0.01	0.43	0.3	0.5
1	0.097	0.21	0.2	0.9
2	0.4	0.329	0.97	0.4
3	0.7	0.4	0.3	0.87
4	0.34	0.4	0.3	0.78
5	0.1	0.3	0.31	0.29

Figure 2-5. :DataFrame with Sentiment score for each category

Continuous numeric values

Continued numeric values are often represented in a vector where `integer`, `float` or `double` are widely used.

Take into consideration the type of data you are working with and cast it when necessary. You can leverage Spark SQL API as shown here:

```
[source,python]casted_data_frame =
sentiment_data_frame.selectExpr("cast(happy as double)")
```

Some of the most common strategies for handling continuous numeric data are as follows.

Fixed Bucketing/Binning

> This is done manually by either binarizing data by providing a specific threshold or a range of buckets. This process is similar to what we discussed earlier about structuring the continuous data.

Adaptive Bucketing/Binning

> The overall data can be skewed. Some values will frequently occur while some will be rare. This might make it hard to manually specify a range for each bucket. This is a more advanced technique where the transformer calculates the distribution of the data and depicts the bucket's range according to it.

Table 4-2 lists the most common continuous numerical transformers available in MLlib. Remember to pick the ones that fit your project based on the usage.

Table 2-3. Commonly used numerical

transformers

API	Usage
Binarizer	Turn a numerical feature into a binary given a threshold. For example, 5.1 with threshold 0.7 would turn into 1 and 0.6 into 0.
Normalizer	Convert a vector of double values into normalized values that are nonnegative real numbers between 0 and 1. The default p-norm is 2 which implements the Euclidean norm for calculating a distance and reducing `float` range to [0,1].
StandardScaler	Estimator that takes a vector of `float` and aims to center the data given the input standard deviation and or mean value.
RobustScaler	Similar to StandardScaler, it takes in a vector of `float` and produces a vector of scaled features given the input data quantile desired range.
MinMaxScaler	Estimator that takes `min` and `max` values where the default range is [0,1].

MaxAbsScaler	Similar to the other scalers' behavior, it takes a vector of `float` and divides each value by the maximum absolute value in the input columns.

Bucketizer	Takes a fixed array of buckets and continues a numerical column. Outputs a bucketed column with a bucket for every numerical point.

QuantileDiscretizer	Take a column of continuous numerical values and split it given maximum buckets that determine the approximate quantiles values.

In addition to those discussed up until now, MLlib offers even more functionalities! These use statistics or abstract other spark functionality that are available using other APIs. Checkout Table 4-4 to learn more about them and their usage. Bear in mind that more are being added every day with code examples available at Apache Spark Github repository under `examples/src/main/python/ml/` directory.

Table 2-4. Other transformers

API	Usage

Interaction	Takes distinct columns vectors and outputs a vector with all the vector combinations.
ElementwiseProduct	Takes a column with vectors of data and a transforming vector of the same size and outputs a multiplication of them that is associative, distributive and commutative. This is based on the Hadamard product. It is used to scale the existing vectores.
SQLTransformer	Takes a SQL statement that is based on Spark's SQL available operations and transforms input of data given the statement.
VectorAssembler	Takes a list of columns and concatenates it into one column in the dataset. This is highly useful for various estimators that take only one column.
Imputer	Takes a column of numeric type value and completes missing values in the dataset using the column mean or median value. Useful when using estimators of ML models that can't handle missing values.
PCA	Principal component analytics is a statistical method that turns a vector of potential correlated values into non-correlated ones, by outputting the most important components of the data(principal component). It is useful in predictive models and dimensionality reduction with a potential cost of interpretability. It is part of the dimensionality reduction set of algorithms.
PolynomialExpansion	Take a vector of features and expand it into a polynomial space given N-degrees, value of 1 means no expansion.
Discrete Cosine Transform (DCT)	Used in signal processing or data compression such as images, audio, radio, and digital television, it takes a vector of data points in the time domain and translates them to the frequency domain.

Preprocessing Images Data

Another common example of data is image data. That too needs to go through preprocessing to move forward in the machine learning workflow. But images are different from what we've seen before! They required another kind of procedure; let's look at the more common path and its 3 steps. Of course, there might be more or fewer steps, depending on the actual data:

- Extracting labels
- Transform labels to index
- Extracting image size

Extracting labels

Starting with labels: our images dataset has a nested structure where the directory name indicates the classification of the image, hence the image whole path on the file system contains the label. We need to extract it in order to use it later on. This is an essential part of the preprocessing with images, and most raw image datasets follow this pattern. After loading the images as BinaryType, we get a table with a column *path* of type String. This contains our label. Now, it's time to leverage String manipulation to extract it. Let's take a look at one path example:
`".../256_ObjectCategories/198.spider/198_0089.jpg"`

We understand that the label is actually an index of the label and a name: `"198.spider",` this is the part we need to extract from the string. Fortunately, Spark SQL functions provide us with the `regexp_extract` API that enables us to easily manipulate strings according to our needs.

Here is how we defined the function with the regex: `"256_ObjectCategories/([^/]+)".`

```
[source, python]
from pyspark.sql.functions import col, regexp_extract
def extract_label(path_col):
  """Extract label category number from file path using built-in sql
function"""
  return regexp_extract(path_col,"256_ObjectCategories/([^/]+)",1)
```

Now we have a python function that takes a *path_col* and uses Spark SQL API to extract a label. Let's create a new DataFrame with labels by calling this function from Spark SQL query:

```
[source, python]
images_with_label = df_result.select(
    col("path"),
    extract_label(col("path")).alias("label"),
    col("content"))
```

Images_with_label DataFrame now consists out of 3 columns:

- `path:string`
- `label:string`
- `content:binary`

Now that we have a label, it's time to transform it into an index:

Transform labels to index

As shown previously, our label is of type *String.* This can pose a challenge for our machine learning model, as strings tend to be heavy on memory footprint. Actually, every string in our table should be transformed before being ingested into a machine learning algorithm, unless it is a true necessity not to. Since our labels are of format `{index.name}` we have three options:

1. Extracting the index from the string itself, leveraging String manipulation.
2. Providing a new index using Spark StringIndexer like we've discussed previously in *From Nominal categorical features to Indexes*.
3. Using python to define an index, there are only 257 indexes in the range of [1,257].

The cleanest way to handle this is actually using the existing index and extracting it. This approach will allow us to track back the experiment and link to the original dataset without maintaining yet another unnecessary index mapping.

Extracting Images size

We choose to extract image size as part of preprocessing the images since we know for sure that the data contains different sizes. This is something we might consider doing to get a feel for the data as well as moving forward to our algorithms. Some machine learning algorithms will require us to have a unified size for images and having that knowledge in advance can help us make better optimization decisions.

Since Spark doesn't yet provide that kind of functionality out of the box, we use Pillow, (aka PIL). A friendly python library for working with Images. For running it distributedly in Spark executors, we leverage *pandas_udf.* Pandas UDF allows us to define python friendly user-defined-functions(UDF) on top of Apache Arrow optimization. Where the UDF takes a series of rows and operates on them, this makes it much faster than the one row at a time approach with Spark traditional UDFs.

```
[source,python]
from pyspark.sql.functions import col, pandas_udf,
from PIL import Image
import pandas as pd
@pandas_udf("width: int, height: int")
def extract_size_udf(content_series):
        sizes = content_series.apply(extract_size)
return pd.DataFrame(list(sizes))
```

Now that we have the function, we can use Spark Select functionally to tie the two:

```
[source, python]
images_df = images_with_label.select(
    col("path"),
    col("label"),
    extract_size_udf(col("content")).alias("size"),
    col("content"))
```

Those code snippets created a Spark transformer request for extracting the size from images into a new column of type struct with width and height:

```
[python,source]
Size:struct
width:integer
height:integer
```

Save the data and avoid small files problem

At the end of an extensive data marshaling, it can be a good idea to save the data to cold or hot storage before continuing for the next step. Sometimes, saving the data between steps is referred to as checkpoints; checkpoints are points in time where we save a version of the data we are happy with. One reason to save the data is fast recovery; if our Spark cluster breaks completely, instead of calculating everything from scratch, we can recover from the last point. The second reason is for collaborating. If data is persistent to storage and available for your colleagues, they can leverage it and develop their flow using preprocessed data. This is especially useful when working with large data sizes and where compute can take extensive resources and time. As Spark provides us with functionality to ingest numerous data formats, it also enables us to save the data in the same manner.

Avoiding Small files

A small file is significantly smaller than the storage block size. Yes, there is a minimum block size even with object stores such as Amazon S3, Azure Blob, etc. Having a significantly smaller file can result in wasted space on the disk since the storage is going to use the whole block size to save that small file, this is an overhead that we should avoid. On top of that, storage is being optimized to support fast read and write by block size. Don't worry! Spark API to the rescue! Instead of wasting precious space, and paying high price for storing small files overhead, we can easily avoid it with Spark *repartition* or *coalescing* functionality. We have more flexibility of which one to choose from since we are operating on offline data. Since we want to be in control of the exact number of partitions, it's ok to allow for longer *repartition* compute by running the *repartition* operation (as shown in the code snippet below) on the data instead of *coalescing* that is known to run faster. *Coalescing* functionality itself is known to run faster than *repetition* as it uses existing partitions to minimize the amount of data shuffled over the network. It first detects the existing partitions, and then shuffles them. On the contrary, *repartition* method creates new partitions completely and shuffles the data over the network to reach a final state of evenly distributed data over the partitions. Do know that it's not always perfect, but this is the functionality goal and execution. For the long term, repetition has a price of shuffling all the data over the network in the beginning, but later down the road, Spark functionality will execute faster as

Spark is a distributed compute engine over distributed data. We can address this functionality again in any step of the machine learning workflow when we notice a relatively slow compute and want to speed it up.

```
[source, python]
output_df = output_df.repartition(NUM_EXECUTERS)
```

Image Compression and Parquet

Let's take a look at our images dataset as an example, we would like to save it in Parquet file format. When saving from Spark into Parquet files, there is a default parquet compression named Snappy. However, since images already possess codec compression (JPEG, PNG, etc.) it wouldn't make sense to compress them again. How do we do it? We save the existing configured compression in an instance, configure spark with parquet to *uncompressed* codec, save the data with parquet format and assign coalescing back to spark configuration for future work. Here is a code snippet:

```
[source, python]
# Images data is already compressed so we turn off parquet compression
compression = spark.conf.get("spark.sql.parquet.compression.codec")
spark.conf.set("spark.sql.parquet.compression.codec", "uncompressed")
# save the data stored in binary format as parquet
output_df.write.mode("overwrite").parquet(save_path)
spark.conf.set("spark.sql.parquet.compression.codec", compression)
```

Descriptive Statistics: Getting a Feel for the Data

Machine learning is not magic, you will need to understand the data. Understanding the data beforehand will save you much time and effort. MLlib provides a dedicated library named *pyspark.ml.stat* that contains all the functionality for extracting basic statistics out of the data.

Don't worry! You don't need to fully understand statistics to use the MLlib, but it can definitely help you in your machine learning journey. Understanding the data using statistics enables us to better decide on which machine learning algorithm to use, identify biases and estimate the data quality. Like mentioned earlier, it's all about *Garbage in Garbage Out*. Ingesting low-quality data into a

machine learning algorithm will result in a low-performing model, hence, this part is a must!

PANDAS, KOALAS AND DASK

For a deeper, statistical analysis of a given data, many data scientists are using Pandas library. Pandas is a python analysis library for working with relatively small data that can fit in one machine memory - RAM. It's counterpart in the Apache Spark ecosystem is Koalas. Koalas is an open-source project for running pandas API on top of Apache Spark DataFrames. If you wish to leverage pandas API over a large set of data independently of Spark engine, check out Dask.

In this section we shift gears from straightforward processes and will focus on getting a feel for the data with the Spark MLlib functionality for computing statistics.

Calculating Statistics

Welcome to the Machine Learning Zoo Project!

For learning about MLlib statistics functionalities, we'll use the Zoo Animal Classification dataset from the Kaggle repository. This dataset was created in 1990 and has 16 Boolean-valued attributes with various traits to describe animals and 7 class types: Mammal, Bird, Reptile, Fish, Amphibian, Bug, and Invertebrate.

The first thing you need to do to get a feel of the data to better plan your machine learning journey is calculating the features statistics. Knowing how the data itself is distributed will provide you with valuable insights to determine which algorithms to select, how to evaluate the model, and overall how much effort you need to invest in it.

Descriptive Statistics with Spark Summarizer

From wikipedia: "A descriptive statistic is a summary statistic that quantitatively describes or summarizes features from a collection of information"

MLlib provides us with a dedicated summarizer object for computing statistical metrics from a specific column. This functionality is part of the MLlib LinearRegression for building the `LinearRegressionSummary`. When building the Summarizer, we need to specify the desired metrics. *Table 4-1* lists the functionality available in Spark API.

Table 2-5. Summarizer metrics options

n
s

Metric	Functionality description
mean	calculate the center of a given numerical column
sum	summary of the numerical column
variance	how far the set of numbers in the column are spread out from its mean value on average
std	Standard deviation - The square root of the variance value to provide more weight to outliers in the column
count	Counts the size of the dataset
numNonZeros	calculated the amount of non zeros in each columns
max	calculates the maximum value in the column
min	calculates the minimum value in the column
normL1	calculates the L1 norm - similarity between the numeric values

normL2	calculates the Euclidean norm

> **NOTE**
>
> Euclidean norm (aka Norm 2) and Norm 1 are tools for calculating the distance between numeric points in an N-dimensional space. It is used as a common metric to measure the similarity between data points. It is often used in fields such as geometry, data mining, and deep learning.

Here is a code snippet on how to create an instance of Summarizer object with metrics:

```
[source,python]from pyspark.ml.stat import Summarizer
summarizer = Summarizer.metrics("mean","sum","variance","std")
```

Just like the rest of the MLlib functionality, the summarizer expects a vector of numeric features as an input. For assembling the vector, you can use MLlib `VectorAssembler` functionality.

Although there are many features in the Zoo datasets, we are going to examine the following columns:

- Feathers
- Milk
- Fins
- Domestic

We are loading the data into a DataFrame instance named `zoo_data_for_statistics`.

In the next code sample, you can see how to build the vector. Notice how we set the output column name to be `features` as expected by the summarizer.

```
[source, python]from pyspark.ml.feature import VectorAssembler
# set the output col to features as expected as input for the
summarizer
vecAssembler = VectorAssembler(outputCol="features")
# assemble only part of the columns for the example
```

```
vecAssembler.setInputCols(["feathers","milk","fins","domestic"])
vector_df = vecAssembler.transform(zoo_data_for_statistics)
```

Our vector is leveraging Apache Spark's Dataset functionality, which is a strongly typed objects collection that encapsulates the DataFrame. You can still call the DataFrame from a Dataset if needed. Dataset enables you to call on a specific column without the dedicated column functionality:

```
[source, python]
Vector_df.features
```

Now there is a dedicated vector column, and summarizer, let's extract statistics. We can call `summarizer.summary` functionality to plot all the metrics or call on a specific metric:

```
[source, python]
# compute statistics for multiple metrics
statistics_df =
vector_df.select(summarizer.summary(vector_df.features))
# statistics_df will plot all the metrics.
statistics_df.show(truncate=False)

# compute statistics for single metric "std" without the rest
vector_df.select(Summarizer.std(vector_df.features)).show(truncate=Fal
se)
```

Figure 2-6 shows the output of calculating `std on the vector of features.`

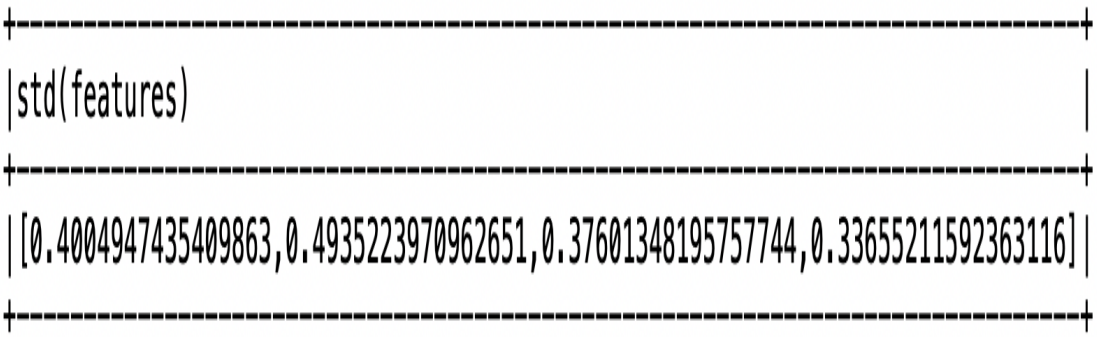


Figure 2-6. STD of the Features column

Standard deviation (STD) is an indicator of the variation in a set of values. A low standard deviation indicates that the values tend to be close to the mean

(also called the expected value) of the set, while a high standard deviation indicates that the values are spread out over a wider range.

Since the features feathers”,"milk”,"fins”,"domestic” are inherently of type boolean: milk can be 1 for true or 0 for false false, same for fins and so on, calculating STD doesn’t provide us with much insights, since the result will always be a decimal number between 0 and 1. That misses the value of STD in calculating how “spread out” the data is. Instead, let’s tryout the sum functionality. The sum functionality will tell us how many animals in the dataset have feathers, milk, fins or are domestic animals.

```
[source,python]
# compute statistics for single metric "sum" without the rest
vector_df.select(Summarizer.sum(vector_df.features)).show(truncate=Fal
se)
```

Take a look at the output of `std`:

```
+--------------------+
|sum(features)       |
+--------------------+
|[20.0,41.0,17.0,13.0]|
+--------------------+
```

Figure 2-7. Sum of the Features column

Figure 2-7 and the sum of feature data, tells us that there are 20 animals with feathers (vector first value), 41 animals (vector second value) that provide milk,17 animals with fins and overall 13 domestic animals. The `sum` functionality provides us with more insights about the data itself than the `std` due to the boolean nature of the data. However, the more complicated/diverse the dataset is, regardless of which function is used, looking at the various metrics will only help.

Correlation

Correlation between two features means that if feature A increases or decreases, feature B would act accordingly (positive Correlation) or the exact opposite (negative Correlation). Since machine learning algorithms' goal is to learn from data, perfect correlated features are less likely to provide insights to improve model accuracy. This is why filtering them out can significantly improve our algorithm performance while maintaining the quality of the results. ChiSquareTest in MLlib is a statistical test that helps us assess categorical data and labels by running Pearson correlation on all pairs and outputting a matrix with a correlation score. Be mindful that Correlation doesn't necessarily imply causation. In this section, you will learn about Pearson and Spearman's Correlation in Spark MLlib.

Pearson Correlation

Pearson correlation measures the strength of linear association between two variables. It produces a coefficient r that indicates how far away these data points are from the line. The range of r is [-1,1]:

- `r=1` is a perfect positive correlate. Both variables act in the same way.
- `r=-1` is perfect negative/inverse correlation which means that when one variable increases, the other decreases.
- `r=0` means no correlation.

Pearson is the default correlation test with MLlib `Correlation` object. Let's take a look at the code and the results:

```
[source, python]from pyspark.ml.stat import Correlation
from pyspark.ml.stat import KolmogorovSmirnovTest

# compute r1 0 pearson correlation
r1 = Correlation.corr(vector_df, "features").head()
print("Pearson correlation matrix:\n" + str(r1[0])+ "\n")
```

The output is a Row where the first value is a `DenseMatrix`:

```
Pearson correlation matrix:
DenseMatrix([[ 1.        , -0.41076061, -0.22354106,  0.03158624],
             [-0.41076061,  1.        , -0.15632771,  0.16392762],
             [-0.22354106, -0.15632771,  1.        , -0.09388671],
             [ 0.03158624,  0.16392762, -0.09388671,  1.        ]])
```

Figure 2-8. Pearson correlation matrix

Each line represents the correlation of a feature with all the other features in a pairwise way : `r1[0][0,1]` represents the correlation of feathers with milk, which is a negative value -0.4107 that indicates negative correlation between animals that produce milk and animals with feathers.

Check out Table 4-6 understand the matrix correlation outcome.

Table 2-6. Pearson correlation table

	feathers	milk	fins	domestic
feathers	1	-4.107	-0.2235	0.0315
milk	-0.4107	1	-0.1563	0.1639
fins	-0.2235	-0.1563	1	-0.0938
domestic	0.0315	0.1639	-0.0938	1

From Table 4-6 you can spot a negative and positive correlation, for example, fins and milk has a negative correlation versus domestic and milk has a positive correlation.

Spearman Correlation

Spearman correlation measures the strength and direction of association between two variables. Also called monotonic, it is a curvilinear relationship whereas Pearson measures linear association. Spearman should be used when the data is discrete and doesn't necessarily follow a linear line as shown in Figure 2-9. For testing correlation and deciding which approach fits the data better, you would need to understand the nature of the data itself: if it's from an ordinal scale - use spearman, or of an interval scale - use pearson. To learn more about it, I recommend reading the Practical Statistics for Data Scientists book.

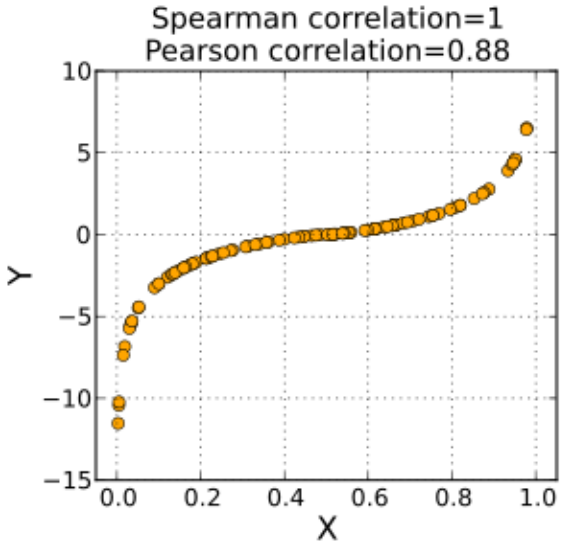


Figure 2-9. You can see how spearman correlation is not a linear line in the image from wikipedia

Notice that for using spearman you have to specify it:

```
[source,python]from pyspark.ml.stat import Correlation
from pyspark.ml.stat import KolmogorovSmirnovTest

# compute r1 0 pearson correlation
r2 = Correlation.corr(vector_df, "features", "spearman").head()
print("Spearman correlation matrix:\n" + str(r2[0]))
```

The output is a `Row` with `DenseMatrix` similar to the one before and follows the same rules and order as we've seen previous in Figure 4-10.

```
Spearman correlation matrix:
DenseMatrix([[ 1.        , -0.41076061, -0.22354106,  0.03158624],
             [-0.41076061,  1.        , -0.15632771,  0.16392762],
             [-0.22354106, -0.15632771,  1.        , -0.09388671],
             [ 0.03158624,  0.16392762, -0.09388671,  1.        ]])
```

Figure 2-10. Spearman correlation matrix

AUTOMATED FEATURE SELECTOR

Spark has an automation for feature selectors based on correlation and hypothesis tests, such as chi-squared, ANOVATTest and F-value (discussed in Chapter 5 when we cover optimizing features during feature engineering). So to speed up the process it is best to use those instead of calculating every hypothesis by yourself. If after the features selectors process you identify an insufficient set of features, it's best to use hypothesis tests to evaluate if you need to enrich your data or find a larger set of data. For that, you can use the ChiSquareTest Statistical hypothesis. We provided you with a code example in the book's GitHub repository demonstrating how to use it. Statistical hypothesis tests have a null hypothesis (H0) and alternative hypothesis (H1).

- *H0:* The sample data follow the hypothesized distribution.
- *H1:* The sample data does not follow the hypothesized distribution. The outcome of the test is the pValue which demonstrates the likelihood of the column to be related to H0.

TUNING LINEAR ALGEBRA OPERATIONS

Under the hood, Spark has a private object named `BLAS.BLAS` stands for Basic Linear Algebra Subprograms. This is a routine that is used in `MLlib vectors` and `matrices.` For advanced optimizations, BLAS uses a java library named `netlib-java` that optimizes native linear algebra operations based on the operating system and hardware.

Read here to learn how to accelerate linear algebra to find the correct library for your operation system. Notice that Spark MLlib would still work without you defining it specifically, but for optimization purposes, it is a good idea to leverage that capability.

Summary

In this chapter, we discussed ingesting data, preprocessing text and images, and descriptive statistics. These are crucial steps in any machine learning workflow

where people spend a significant portion of their time on. Executing on those steps mindfully sets us up for greater success and a better machine learning model that can answer the business goal in a much more profound way. As a rule of thumb, it's best to collaborate with your peers to validate and engineer these steps to ensure the insights, and data can be reused in multiple experimentations. The tools in this chapter will accompany us again and again throughout the process. In the next chapter, we will dive into feature engineering and build upon the outcomes from this chapter.

---

1 Image file format wikipedia(https://en.wikipedia.org/wiki/Image_file_formats)

2 Spark binary format docs(https://spark.apache.org/docs/latest/sql-data-sources-binaryFile.html) - notice that binary data source schema can change with new releases of Apache Spark, or when using Spark in managed environments such as Databricks.

Chapter 3. Feature Engineering

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. The GitHub repo is available now at https://github.com/adipolak/ml-with-apache-spark.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jleonard@oreilly.com.

In machine learning, Feature Engineering is the process of using *domain knowledge* to select and transform the most relevant variables from the data to reach the machine learning process goal. Feature Engineering is also referred to as Featurization. It is a crucial part of processing the data in your machine learning pipeline.

In data science, domain knowledge is the general background of a specific field or vertical, it is less about the tools, and more about the data and problem itself.

Using domain knowledge can help you simplify the machine learning problem (aka. better the chances of reaching your business goal). To be successful, you must know the problem background and understand the data you are working with. For example, let's assume you are in charge of detecting the vehicle model and year from a set of images. That can be a tough problem to solve! After all, the car has many angles and you will need to differentiate and narrow down the options based on very specific model features. Rather than asking your ML model to assess many

individual features, instead you can detect the vehicle registration plate and translate the pixels into numbers. Once you have the vehicle plate numbers, you can match them with other datasets to extract the vehicle model and year. Since extracting numbers from images is a solved ML problem, we leveraged our domain knowledge together with feature engineering to reach the business goal.

As you saw in the vehicle example, your goal and data will determine which features you will need for training your models. Interestingly enough, given that machine learning algorithms can be the same for different scenarios, like the classification of emails into Spam and Non-Spam or the classification of Twitter accounts into Bots or Real Users, *Good quality* features might be the main driver for your model performance. Being a data scientist with specific expertise in the business domain can be an advantage when seeking data scientist jobs. For example in Cyber Security there are explicit requirements to understand the data-domain. In healthcare tech, you might be working with a medical doctor to design the features and knowledge of anatomy, biological systems, and medical conditions may be required.

On top of that, most machine learning algorithms are not intelligent enough to automatically extract meaningful features from raw data. As a result, stacking multiple algorithms, *extractors*, and *transformers* with smart *selectors* can help us achieve noteworthy features.

Before we proceed, here are a couple of terms you should know:

Estimator

An algorithm that can be fit on a DataFrame to produce a Transformer.

Hashing function

A function that is used to map data of arbitrary size to fixed-size values.

Derived Feature

Feature obtained from feature engineering.

Raw Feature

> Feature obtained directly from the dataset with no extra data manipulation or engineering.

In this chapter, we will explore some of those featurization techniques using Spark tools.

Features and Their Impact on Models

Since our models are a mathematical and statistical representation of the data we built them with, they might be impacted by outliers, have internal bias installed in them, or other culprits that we would want to avoid. Also, some algorithms are more sensitive to the magnitude of features. Think of a missing values scenario where you had to decide what to do given your algorithms libraries didn't support such inputs. Did you fill those columns? Did you drop it altogether? Which algorithm did you use? Those decisions impact the center of gravity in algorithms such as linear models, SVMs, neural networks, PCA, and nearest neighbors since each data point changes the outcome. Stated plainly: *If the missing values are larger than the real value, filling them with a default value can completely tilt the equation in favor of the default value.* To better understand this, let's take a look at an SVM(Support Vector Machine) algorithm, as shown in Figure 3-1, where we need to develop a linear vector/function (H1, H2, H3) to distinguish between two categories: an empty circle or a full circle given x1 and X2 as an input. If we would have missing values for most of the X2 in the empty circle category and we'd provide them with a default value, that will completely change our linear vector direction and degree. So we have to think carefully about how we are filling in missing data and how our features impact the models.

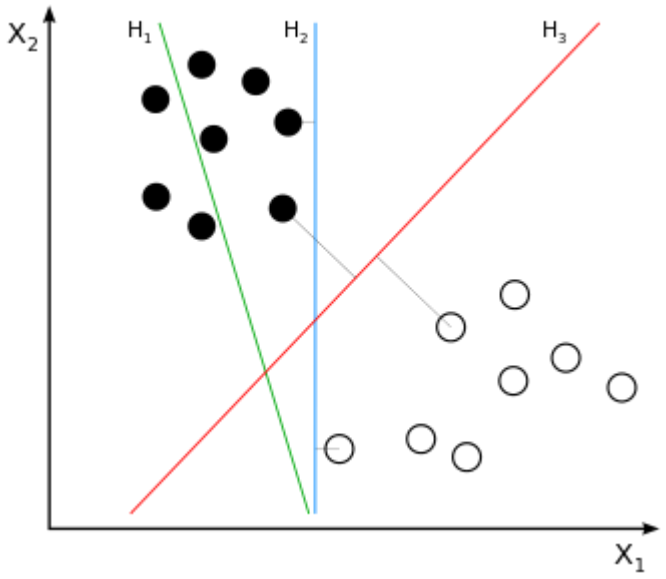


Figure 3-1. SVM with linear hard margin, image from Wikipedia

Another important aspect of featurization is the ability to identify and reduce *noise*, especially when using automated machine learning models that have built-in feature extraction mechanisms. Noisy data in algorithms provide a risk of producing a wrong pattern that the algorithm starts to generalize from, which in turn creates an undesired model outcome that doesn't represent well the business domain. Understanding the data together with featurization is key here.

As shown in Figure 3-2, feature engineering is a rich world of techniques, tools, and requirements.

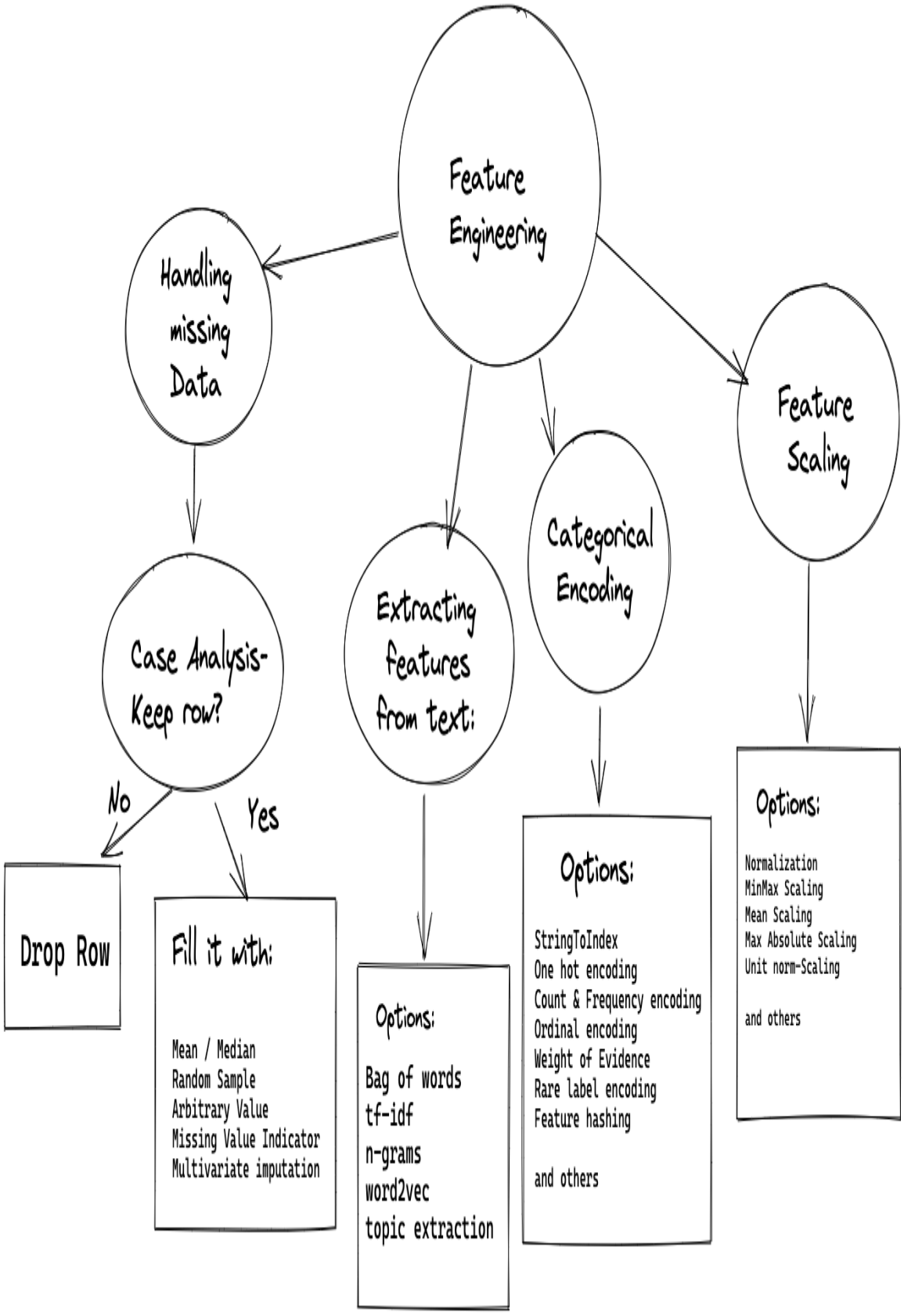

Feature Engineering
Handling missing Data
Case Analysis- Keep row?
No
Yes
Drop Row
Fill it with:
Mean / Median
Random Sample
Arbitrary Value
Missing Value Indicator
Multivariate imputation
Extracting features from text:
Options:
Bag of words
tf-idf
n-grams
word2vec
topic extraction
Categorical Encoding
Options:
StringToIndex
One hot encoding
Count & Frequency encoding
Ordinal encoding
Weight of Evidence
Rare label encoding
Feature hashing
and others
Feature Scaling
Options:
Normalization
MinMax Scaling
Mean Scaling
Max Absolute Scaling
Unit norm-Scaling
and others

Figure 3-2. High level of feature engineering requirements and techniques

Each technique is directly related to the type of data and requirement, there are no hard rules or guidance, but a set of questions you will need to answer:

Handling missing data

Do I use this data for a specific case analysis? What is the "price" or dropping the whole row versus filling it out with default data? What is the percentage of the missing data in my dataset? Is that a whole row, or specific columns? Answer those questions and you've got yourself a plan!

Extracting features from text

Do I have only unstructured text data? How would I make sense of it? what is the nature of my data? How long is the text? A tweet with a limited number of characters? Or a Feature Engineering chapter in a book? More about it in the Text Featurization process.

Categorical Encoding

This is the process of transforming categorical variables into a set of binary variables. The goal here is to boost the performance of the model. Example of categorical data is the city where a person lives: San Francisco, New York, Berlin, Tel Aviv, etc. Those are nominal categories, as there isn't a specific order to them. Categories can be of inherent order as well, for example a student's grades: A, A+,B+, etc. We will showcase them at chapter 6, many algorithms take encoding categories as input.

Feature Scaling

We use this technique to standardize the independent features in the dataset. We attend to it during the preprocess phase or as part of the feature engineering when new highly varying values show up. It is also referred to as data normalization. We would want to use it when

working with gradient descent algorithms, as the difference in ranges of a specific feature will create too big of a step to reach overall optimization. See Figure 3-3 that demonstrates how X moves into an inner circle in each gradient descent algorithm iteration until it hits an optimum phase or maximum allowed iterations. You can think about it as a person stuck in the woods and trying to find a parking place; that person would wander around, collect information and try to optimize their route. But what if most of the trees were transparent and the person could see through them? This is feature scaling. It is also useful for distance based algorithms and regression. For feature scaling we can leverage a machine learning algorithm named Principal Component Analysis, it's an unsupervised technique that enables us to filter noisy dataset and reduce data dimensions.

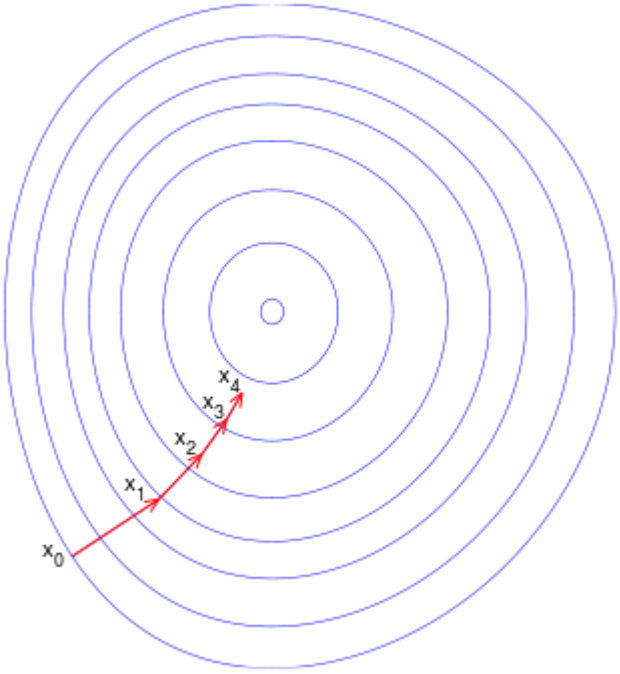


Figure 3-3. how the center of gravity moves in gradient descent in every iteration, from x1 to x2 and so on, figure by wikipedia

The MLlib Tools

PySpark MLlib provides many featurization functionalities in it's `pyspark.ml.features` package (notice that the Scala-based APIs are at `spark.ml.features`.). New features are being added every day. Therefore, we would cover the most common ones. The whole list of up-to-date APIs can be found at Apache Spark MLlib documentation site and code examples can be found at Apache Spark Github repository under `examples/src/main/python/ml/` directory.

In this section, we will cover MLlib Extractors and Selectors.

Extractors

Extractors are MLlib APIs for excerpting features that are not necessarily meaningful on their own, but can help us along the process. Transformers (refer to the preprocess discussion in Chapter 4 if you need a refresher) can be used as extractors and vice versa. Some of the extractors would require us to use Transformers first, like `TF-IDF` which cannot operate directly on the raw data and requires preprocessing of `Tokenizer` for extracting words and `HashingTF` or `CountVectorizer` for hashing the words.

Table 5-1 is a helpful guide on when to use each API.

Table 3-1. Spark MLlib Extractors

A
P
I
s

API	Use
TF-IDF	Term frequency-inverse document frequency, used in text mining for weighing the importance of a term to a document. Converts array of words into
Word2Vec	Convert sequences of words into a fixed-size vector.
CountVectorizer	Convert fixed size sequences of words into vectors of token counts. When sequence size varies, it will use the minimal size as the vocabulary size.
FeatureHasher	Takes a set of categorical or numerical features and hash them into one feature vector. Often used to reduce features without significantly lose their value.

Selectors

Often used last in the process of Featurization are the selectors. These are APIs for selecting a subset from a large set of features. After we use our transformers and extractors to develop a bunch of features, it's time to select the ones we would like to keep. This can be done manually or adaptively using algorithms that estimate features importance and aim to improve our machine learning model performance. Notice that too many features might lead to overfitting.

Spark provides simple options that are listed in Table 5-2.

Table 3-2. Spark MLlib Selectors A

P
I
s

API	Use
VectorSlicer	Takes a column of features vector and outputs a new column with a subset of that features given specific indices array from the user. As developers we specify the indices and VectorSlicer slices the vector for us.
RFormula	For `R` programming language users, using a dedicated formula with the basic operation for manipulating columns into one column.
ChiSqSelector	Uses ChiSquare test, which is a collection of statistical tests for evaluating the significant difference between the features and their correlation to the label itself. Takes a column with a vector of all the features, and a label column, and generates a DataFrame with a new column with a vector of selected features
UnivariateFeatureSelector	Takes as input categorical/continuous labels with categorical/continuous vector features and is based on it. Spark chooses the score function (chi2, ANOVATest or F-value)
VarianceThresholdSelector	Remove low variance features given a specific variance threshold, requires columns of feature vector, label, and variance minimum bar, all features with ⇐ variance are removed. Yields a new DataFrame with selected features column

SIMPLIFIED EXAMPLE OF EXTRACTOR AND SELECTOR AND THE IMPORTANCE OF PERSISTING DATA TO STORAGE

Many of the selectors and later machine learning models take vectors of features in the shape of one column in the DataFrame. This is why, using a columnar storage format such as Parquet is not always more efficient, as all the features are represented as one column anyway. However, it can be more efficient when given a large DataFrame with many columns, where each one represents a different transformation or set of features. This is why Spark MLlib API would generate a new DataFrame with the same columns as the previous one and a new column to represent the new feature.

Many of the transformers can be persistent to disk and reuse later again, this code snippet will show how we can stack multiple transformers, persist to disk and load from disk:

```
[source, python]from pyspark.ml.feature import Word2Vec,
Word2VecModel
from pyspark.ml.feature import Tokenizer

# Input data: Each row is a bag of words from a sentence or
document.
sentence_data_frame = spark.createDataFrame([
    (0, "Hi I think pyspark is cool ","happy"),
    (1, "All I want is a pyspark cluster","indifferent"),
    (2, "I finally understand how ML works","content"),
    (3, "Yet another sentence about pyspark and ML","indifferent"),
    (4, "Why didn't I know about mllib before","sad"),
    (5, "Yes, I can","happy")
], ["id", "sentence", "sentiment"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
tokenized = tokenizer.transform(sentence_data_frame)

word2Vec = Word2Vec(inputCol="words", outputCol="result")
model = word2Vec.fit(tokenized)

model.write().overwrite().save("some_path")
model_from_disk = Word2VecModel.load("some_path")
```

```
df_with_word_vec = model_from_disk.transform(tokenized)
df_with_word_vec.show()

selector = VarianceThresholdSelector(varianceThreshold=0.0,
outputCol="selectedFeatures")
result = selector.fit(df_with_word_vec).transform(df_with_word_vec)
```

- `Sentence_data_frame` is a Mock DataFrame for demonstrating the functionality.
- `Tokenizer -` Creating tokenizer instances, providing input column and output column names.
- Transforming the DataFrame and generating a new one named `tokenized`.
- Creating `Word2Vec` instance with input column, output column and vector size.
- Word2Vec is an Estimator chance required to run fit on the data before it is used for transformation. More on that later.
- `.save -` Persisting Word2Vec model to disk using write functionality.
- `Word2VecModel.load -` Loading Word2Vec model from disk.
- `Model_from_disk.transform -` Using the loaded model for transforming the tokenized data.
- `VarianceThresholdSelector -` Creating VarianceThresholdSelector instance for selecting features, with a threshold of 0, meaning filtering out the features with the same value in all samples. VarianceThresholdSelector expects `features` named columns as input.
- Fitting the VarianceThresholdSelector on the data and using the model for selecting features.

This is how the result DataFrame would look like at the end:

```
Output: Features with variance lower than 0.000000 are removed.
+---+----------------+---------+----------------+----------------+----------------+
| id|        sentence|sentiment|           words|        features|  selectedFeatures|
+---+----------------+---------+----------------+----------------+----------------+
|  0|Hi I think pyspar...|    happy|[hi, i, think, py...|[-6.2237260863184...|[-6.2237260863184...|
|  1|All I want is a p...|indifferent|[all, i, want, is...|[-7.1128298129354...|[-7.1128298129354...|
|  2|I finally underst...|  fulfill|[i, finally, unde...|[-8.2983014484246...|[-8.2983014484246...|
|  3|Yet another sente...|indifferent|[yet, another, se...|[0.0,0.0,0.0,0.0,...|[0.0,0.0,0.0,0.0,...|
|  4|Why didn't I know...|      sad|[why, didn't, i, ...|[-7.1128298129354...|[-7.1128298129354...|
|  5|       Yes, I can|    happy|    [yes,, i, can]|[-0.0016596602896...|[-0.0016596602896...|
+---+----------------+---------+----------------+----------------+----------------+
```

Figure 3-4. Result DataFrame with all the new columns

For simplicity, we didn't normalize our features vector to fit the range of [0,1], which could have been helpful with the selection process later. This is another route you should explore during the feature engineering process. As a side note, while in the tutorial example we save the model to disk, you are able to save it to any store where you have connectors to and can support parquet format since the model itself is saved in parquet format unless defined otherwise. We will discuss it further in Chapter 10.

Image Featurization Process

There is a misconception in the industry regarding images and feature engineering. Many assume there is no need for that, but in practice, even when we are using a neural network algorithm that extracts and maps

features such as Convolutional Neural Networks (aka. CNN, shown in Figure F:5), there is still a chance of introducing noise, computation overhead, and missed features since the algorithm itself, doesn't possess any business or data domain understanding of the given problem.

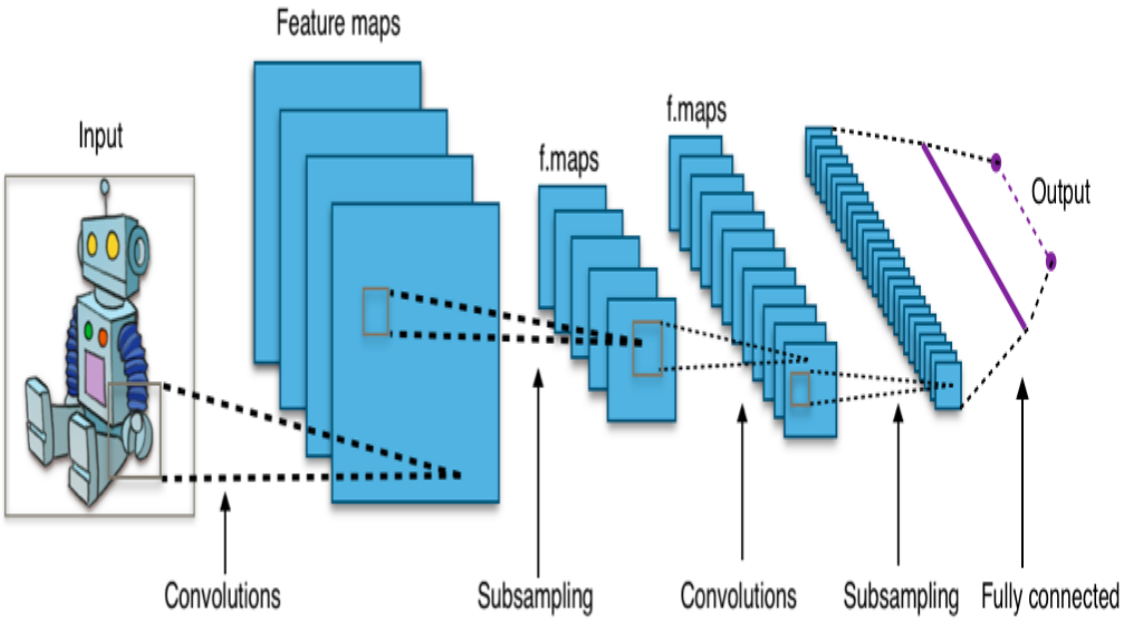


Figure 3-5. Typical CNN architecture from Wikipedia

There are many features we can extract from image data. All depend directly on the domain we work with. Our Caltech256 - image classification dataset is quite diverse and we can try out multiple techniques until we find the best features. Do bear in mind that when it comes to images and layered neural networks models, we can leverage an existing model and just add the last layer. More on that in the next chapters. We mention it here since it might require specific image features, such as specific width, height, and channels. While it seems that those characteristics are straightforward, let's refine their definition:

Image channels

> The number of conventional primary colors layers that make up an image. For example, RGB: red, green, and blue. An RGB colored image is actually composed of 3 images (matrix of pixels), one for the red

channel, second for the green, and third for the blue. GreyScale images often have only 1 channel.

Width and Height

Represents the number of pixels in an image. For example, the given dimensions of an image are 180 x 200. These dimensions are basically the number of pixels in the image (height x width).

Now that we have a better understanding of images, we know that for having a full-color image, in RGB format, we actually need 3 matrices (or channels) with a value between 0-255, where a smaller number represents black/dark color and the larger number indicates white.

> **NOTE**
>
> There are other formats of storing image data, however, RGB is the most popular one so we are addressing it here.

How should we think about and define our features? Let's get a basic feel of what we can do with image manipulation.

Understanding Image Manipulation

How best to proceed depends on the image complexity, such as how many objects are in the image, image background, a number of channels, does color provide actionable value, etc. We can think about multiple options for manipulating the existing data and extracting features.

As shown in Figure 3-6 using an RGB example, the colored spaghetti images have 3 layers of matrics that we can manipulate, add filtering, change the pixels, group them, or extract one channel only for getting a grey-scale image without the colors.

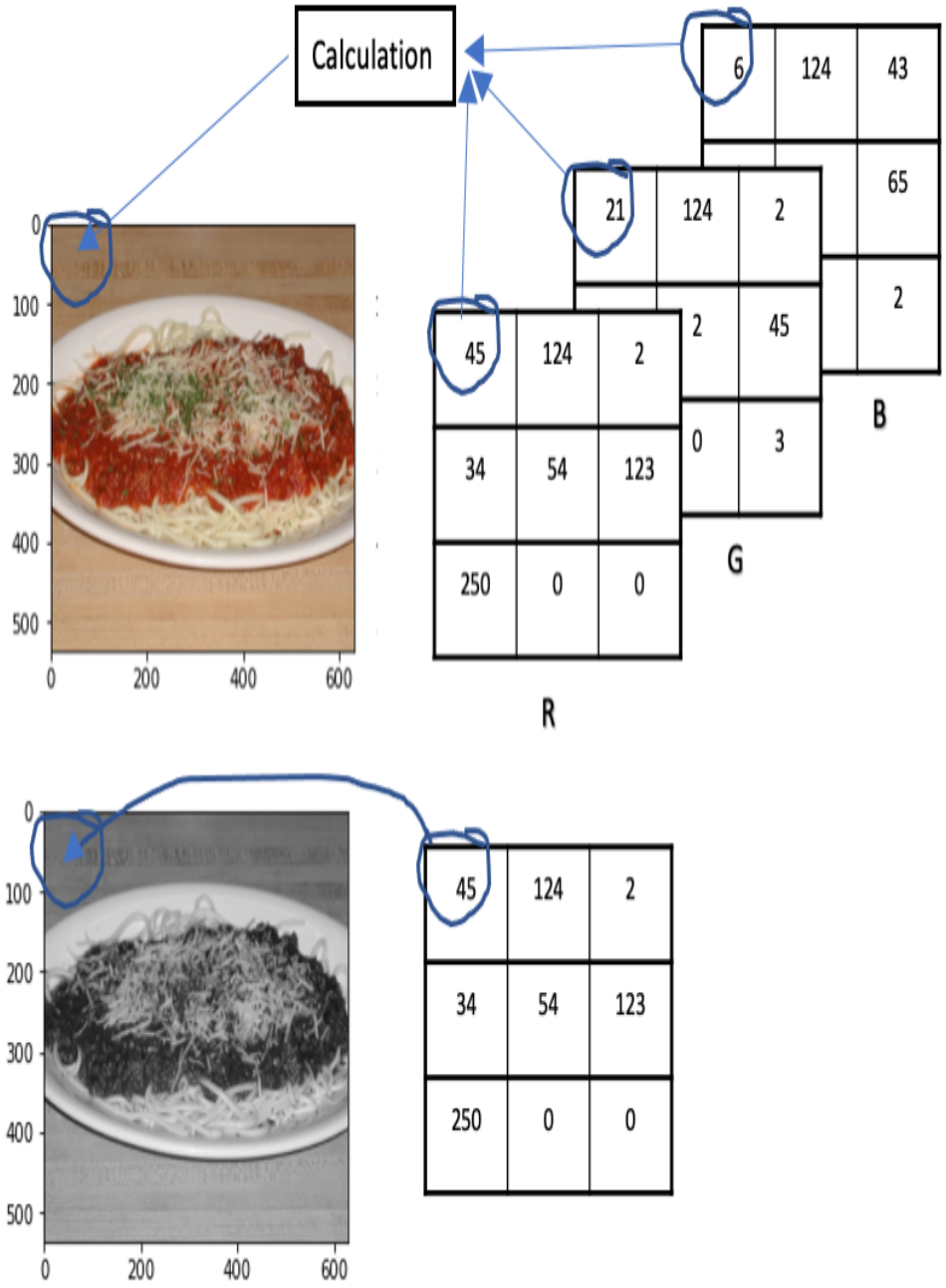


Figure 3-6. RGB image with matrices and grayscale image with matrix

GreyScale

When there is no meaning to the image colors, grey-scale can assist in eliminating noise in our data. GreyScale are Single-channeled pixel images that carry information about the intensity of light. These images are

exclusively made up of shades of gray. They are not black and white images, each pixel represents the intensity of light between 0 to 255.

Defining image boundaries using image gradients

Let’s take a look at another spaghetti image from our repository, image number 196_0070.

This image is classified as plain spaghetti, however, there are also tomatoes, some bread, and what seems like a portion of seafood. Without defining features, like image boundaries, this image can introduce a lot of noise to our algorithms and we might end up with a model that classifies images with tomatoes, seafood, and other ingredients as simply spaghetti pasta.

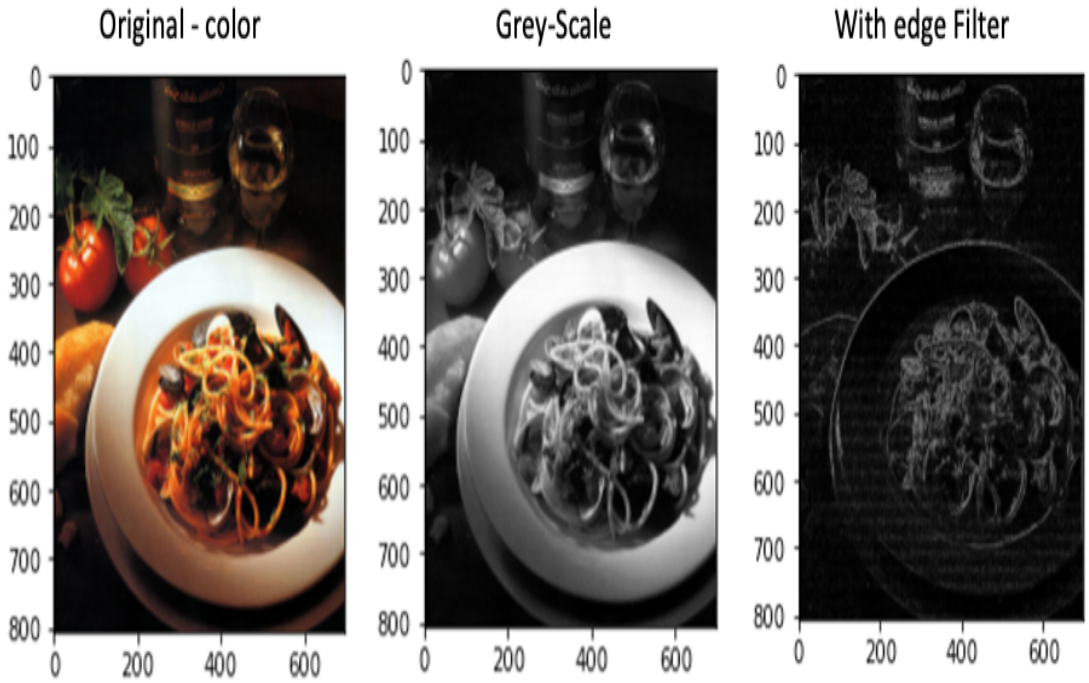


Figure 3-7. Multiple filters, classified simply as spaghetti pasta in our dataset yet it also has tomatoes, seafood and other ingredients

Extracting image edges using gradient calculations can be done by leveraging vector operators such as Laplace. The Laplace operator operates over our pixel vector and helps differentiate between the values. Using

Pillow, we can define our own convolutions methods, with the *Kernel* method. Or we can use the built-in FIND_EDGES filter that the library provides us with out-of-the-box. Take a look at the code sample as an example to learn how to use the two:

```
[python, source]from PIL import Image, ImageFilter
img = Image.open(r"sample.png")
# input image to be of mode Greyscale (L)
img = img.convert("L")
# Calculating Edges providing Kernel manualy
final = img.filter(ImageFilter.Kernel((3, 3), (-1, -1, -1, -1, 8,
-1, -1, -1, -1), 1, 0))
# Calculating Edges leveraging Pillow filter defaults
edges = img.filter(ImageFilter.FIND_EDGES)
```

There are many other filter features Pillow provides us with out-of-the-box such as BLUR, SMOOTH, EDGE_ENHANCE, and others. all of which are based on adding a filter to the image based on pixel gradient manipulation. Figure 3-7 captures how grey-scale and edge-filter features are rendered.

Extracting Features Leveraging Spark APIs

Let's examine multiple techniques using the Caltech256 dataset. In the preprocessing chapter, we started touching a bit on Python UDFs and how to use them for extracting image size. Let's dive deeper into this topic as it is our main tool for feature extraction over images.

Until 2017, PySpark supported Python UDFs that operated on one-row-at-at-a-time, those functions missed on Spark Query Engine optimization capabilities and since, behind the scenes, PySpark is being translated into Scala code, many UDFs written in PySpark had high serialization and invocation overhead. As a result, data engineers and data scientists worked together to define UDFs in Java and Scala where data scientists invoke them from Python. This made the code messy, hard to navigate and maintain. Fortunately, Spark 3.0 introduced Pandas UDFs together with Python Type Hints that allow us to run on groups of rows at a time, leverage Apache Arrow optimization for reducing deserialization operations, and use Pandas API from within the function.

APACHE ARROW OPTIMIZATION

When we transmit data over the network, or within a process running on the same machine, we need to provide a dedicated structure of how to convert the sequence of bits into objects and vice versa. This process is named *Serialization* - translating objects into bits. Another way around is *Deserialization* - translating bits into objects. With Spark translation to Java object and later on to Python Object, there was an overhead of serialization processes. As a result, Apache Arrow format was introduced into Spark. Instead of changing columnar data format, copying and converting data to different formats as shown in Figure 5-8.

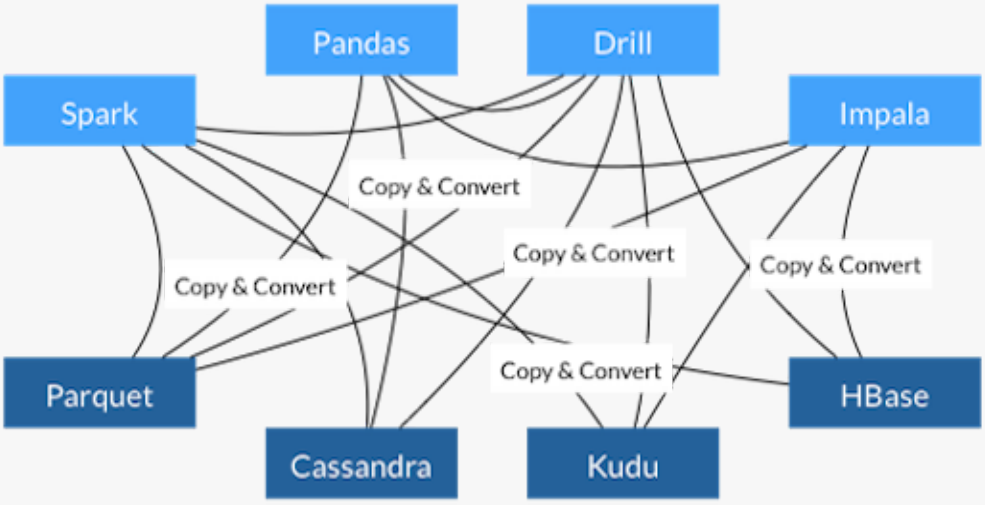


Figure 3-8. Copy and convert columnar data formats between frameworks from Arrow docs

we leverage Apache Arrow as the center of data formats, this approach of having a one data format allows us to avoid expensive translation as shown in Figure 5-9.

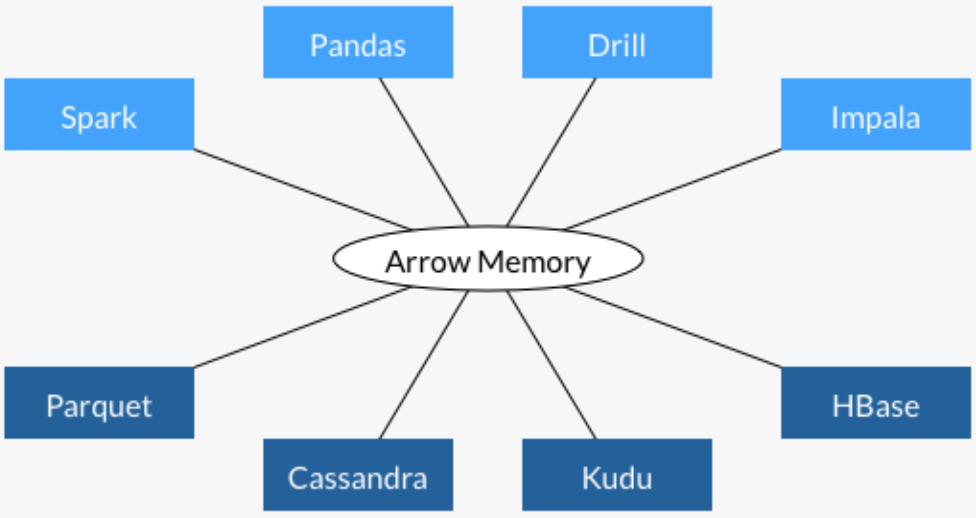


Figure 3-9. Arrow in-memory data format at the center between frameworks from Arrow docs

Arrow in-memory columnar data allows us to exchange data with less serialization-deserialization processes that are not always needed.

pyspark.sql.functions: PandasUDF and Python type Hints

`pandas_udf` are part of `pyspark.sql.functions` library that support dedicated `PandasUDFType` series types as shown in *Table 5-3*.

Table 3-3.: Breakdown of Sparkpandas U

DF types.

	Spark PandasUDFType	Python hints type	UDF Known Pandas Types - Input and output
1	SCALAR	Doesn’t need a specified pandas type, can take: Long, double, float, int, boolean	Pandas Series
2	SCALAR_ITER	Needs to specify UDF pandas type	Iterator of Pandas Series
3	GROUPED_MAP	Doesn’t need a specific pandas type. Can use mapInPandas or applyInPandas function instead	Pandas DataFrame
4	GROUPED_AGG	Doesn’t need a specific pandas type. Can use apply in Pandas function instead	Pandas DataFrame

Each Pandas UDF type (from 1 to 4) expects different input and produces other output types. Also, options 2-4 including, require us to define the type in the function, or use dedicated functions such as *apply and Map* function discussed later.

In Chapter 4, we calculated each image size using *panads_udf*. Let's calculate the average size for each category and decide if we want to resize accordingly.

1. Flatten the Size struct into two columns:

```
[python, source]
flattened = df.withColumn('width', col('size')['width'])
flattened = flattened.withColumn('height', col('size')
['height'])
```

   The flattened dataframe has two new columns, shown in figure 5-10:

```
+-----+------+
|width|height|
+-----+------+
|1500 |1500  |
|630  |537   |
|1792 |1200  |
+-----+------+
```

Figure 3-10. Flattened width and height columns

2. Extract mean width and height for each column, for that we leverage pandas_udf with hinted python type

```
[python, source]
@pandas_udf("int")
def pandas_mean(size: pd.Series) -> (int):
    return size.sum()
flattened.select(pandas_mean(flattened['width'])).show()
flattened.groupby("label").agg(pandas_mean(flattened['width']))
.show()
flattened.select(pandas_mean(flattened['width']).over(Window.pa
rtitionBy('label'))).show()
```

   The *@pandas_udf("int")* at the beginning and the -> in the python function, uses panads_udf and hints pyspark regarding the type we use.

The example above shows how to calculate *width* average, however, we can use the same function for *height* as well.

Output example in Figure 3-9:11.

label	pandas_mean(width)	pandas_mean(height)
196.spaghetti	39019	33160
249.yo-yo	40944	37326
234.tweezer	34513	27628
212.teapot	51516	45729

Figure 3-11. Average height and width using the pandas UDF function

3. Our final step would be to decide if we want to resize the images according to the results, or not. In our case, there is no point in resizing the image, since we are going to leverage PyTorch and TensorFlow ml algorithms and are going to resize according to the algorithm's requirements.

pyspark.sql.GroupedData: applyInPandas and mapInPandas

ApplyInPandas and mapInPandas are two functions that operate over a Grouped Spark DataFrame and return a new DataFrame with the results. When we define the *pandas_udf* to work with them, it receives pandas DataFrame as input and returns Pandas DataFrame as an output.

PANDAS DATAFRAME VS SPARK DATAFRAME

Those are two very different classes and you shouldn't confuse the two. While Spark DF(DataFrame) is an abstraction on top of distributed data, pandas DF isn't. Take a look at Table 5-X to understand the main difference between the two.

Table 3-4. Spark DataFrame vs Pand

a
s

D
a
t
a
F
r
a
m
e

	Spark DataFrame	Pandas DataFrame
Operation in distributed - parallel	yes	no
Lazy operation	yes	no
Immutable	yes	no

Let’s take a look at how we can extract grey_scale using applyInPands functionality:

1. Define the function - `add_grayscale_img` :

```
[python,source]
def add_grayscale_img(input_df):
  input_df['grayscale_image'] = input_df.content.apply(lambda
```

```
image:
get_image_bytes(Image.open(io.BytesIO(image)).convert('L')))
  input_df['grayscale_format'] = "png"
  return input_df
def get_image_bytes(image):
  img_bytes = io.BytesIO()
  image.save(img_bytes,format="png")  return
img_bytes.getvalue()
```

Within `add_grayscale_img,` we leverage PIL image functionality of convert to extract a grey_scale out of the image. `Get_image_bytes` is a supporting function that helps us get the right object class to use with `.convert('L').`

2. Test the function on small data

   Since this is a typical python function, it is best practice to test yourself on a small image data before invoking the functionality over the whole data set.

3. Define desired DataFrame with blank columns for extracting schema easily:

   For running the function over spark, we need to specify the output schema. For setting up the return schema in an easy manner, we leverage the existing DataFrame, adding dedicated blank(None) columns and extracting the schema calling DataFrame.schema. This overall process makes the schema definition easy in the function invocation and reduces schema mismatches.

```
[python,source]
rtn_schema = (df.select('content','label','path')
                                  .withColumn('grayscale_image',
lit(None).cast(BinaryType()))
                                  .withColumn('grayscale_format',
lit(None).cast(StringType()))                    )
```

4. Reduce DataFrame columns to the minimum required, since the grouping and apply in pandas, are relatively expensive operations, more on it in the warning later.

```
[python, source]limited_df =
df.select('label','content','path')
```

5. Run python function on Spark executors:

   Call groupby, we can use any column to group the data and call `applayInPandas` function with the pointer to the function to operate and schema. In our example `add_grayscale_img` is the function we want to execute and `rtn_schema.schema` is our schema.

```
[python, source]augmented_df =
limited_df.groupBy('label').applyInPandas(add_grayscale_img,
schema=rtn_schema.schema)
```

> **WARNING**
>
> The `groupBy` function requires a full shuffle of the data. All the data of a specific group would be loaded into one executor memory. In our example, we group the data into categories, which means that all *content* from the same category will be loaded into a single executor memory as a pandas DataFrame. That can result in out-of-memory exceptions (OOM) if the categories are too large. We used this example to simplify the solution, but we recommend using dedicated image ID and bucketing methods to make sure to avoid OOM.

6. The last step: re-join the data.

   Rejoin data from the original DataFrame with the augmented DataFrame leveraging *leftouter* in case the image transform needs to skip some rows.

```
[python, source]output_df =
df.join(augmented_df.select('path','grayscale_image'),
['path'],"leftouter")
```

   Those 6 steps provided us with two new features, one is of type Byte Array named `'grayscale_image'` and the second is `greyscale_format` of type String.

   Figure 3-12 shows output_df table structure and sample of data rows.

label	content	path	grayscale_image	grayscale_format
249.yo-yo	[255, 216, 255, 224, 0, 16, 74, 70, 73, 70, 0,...	file:/home/jovyan/256_sampledata/249.yo-yo/249...	b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\...	png
196.spaghetti	[255, 216, 255, 224, 0, 16, 74, 70, 73, 70, 0,...	file:/home/jovyan/256_sampledata/196.spaghetti...	b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\...	png
249.yo-yo	[255, 216, 255, 224, 0, 16, 74, 70, 73, 70, 0,...	file:/home/jovyan/256_sampledata/249.yo-yo/249...	b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\...	png
249.yo-yo	[255, 216, 255, 224, 0, 16, 74, 70, 73, 70, 0,...	file:/home/jovyan/256_sampledata/249.yo-yo/249...	b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\...	png
196.spaghetti	[255, 216, 255, 224, 0, 16, 74, 70, 73, 70, 0,...	file:/home/jovyan/256_sampledata/196.spaghetti...	b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\...	png

Figure 3-12. Output example of the added new columns

Text Featurization Process

In the simplified example with extract and selector you learned how to use the `Tokenizer`, `Word2Vec, and other` tools. In this section, we will dive deeper into our Bot Or Not dataset to learn about the featurization process of free short text such as Twitter User Description in the BotOrNot dataset.

Extracting Features

Since our data is supervised, we can explore combinations of existing features with labels to develop new ones. Of course, this might result in features having higher correlation which means we need to think out of the box. It's true that having interpretable features and models is easier to debug than complex ones. However, they do not always lead to the most accurate model. Our dataset has a *description* column of plain text. Let's examine the text features we can extract out of it:

1. Bag of words
2. TF-IDF
3. N-Gram

4. Word2vec (shown before)

5. Others

Figure 3-9-13 shows how description and label columns look like in our data.

```
+--------------------+-----+
|         description|label|
+--------------------+-----+
|Contributing Edit...|    1|
|     I live in Texas|    0|
|Fresh E3 rumours ...|    0|
|''The 'Hello Worl...|    0|
|Proud West Belcon...|    1|
|Hello, I am here ...|    0|
|Meow! I want to t...|    0|
|I have something ...|    0|
|I have more than ...|    0|
|If you have to st...|   13|
|I am a twitterbot...|    1|
|Designing and mak...|    0|
|Host of Vleeties ...|    0|
|Benefiting Refuge...|    0|
|Access Hollywood ...|    0|
|Producer/Songwrit...|    0|
|CEO @Shapeways. I...|    0|
|Two division UFC ...|    0|
|Moderator of @mee...|    0|
|Tweeting every le...|    0|
+--------------------+-----+
only showing top 20 rows
```

Figure 3-13. Twitter Bot-Or-Not description and label columns

Bag of words

Bag-of-words is part of the bag-of-x methods for turning text into flat vectors and extracting term frequency vectors. During that process, we take in raw input and produce a bag of words vectors that contains a word and its frequency in the given text.

As an example, let’s assume we have the following Twitter account description:

Tweets on programming best practices, open-source, distributed systems, data & machine learning, dataops, Apache Spark, MLlib, PyTorch & TensorFlow.

While MLlib provides us with great functionality to calculate just that, like `HashingTF` and `CountVectorizer` that allow us to generate the term frequency vectors

There is no real reason to do that from this description as all word's frequency is 1. However, if we group descriptions of all accounts that are labeled as bots, we might find distinguished terms that appear more than once. This brings us to a second option - TF-IDF.

TF-IDF

TF-IDF is a method from the field of information retrieval that is highly used in text mining. It allows us to reflect on the importance of a term to a document in a corpus. In our example, the importance of a term in a description, given a whole corpus of descriptions. *Term Frequency* - `TF(t,d)` is the number of times that term `t` appears in document `d,` and *Document Frequency* - `DF(t, D)` is the number of documents that contain term `t.`

- d - a document
- t - a term
- D - corput

IDF is the inverse document frequency that provides us a numerical measure of how much information a specific term provides.

Hence TF-IDF is multiplying those outputs:

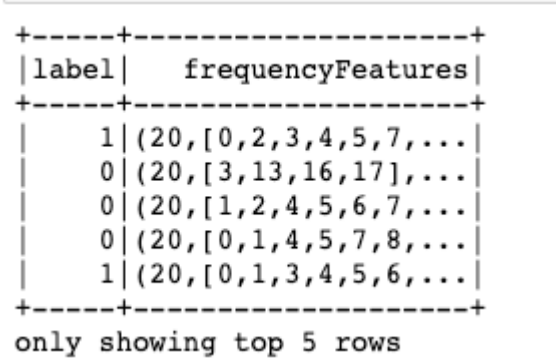
$$TFIDF(t,d,D)=TF(t,d)\cdot IDF(t,D)$$

Since TF-IDF functionality is separated in MLlib, that provides us with the flexibility of deciding how to work with them and if to use them together or separately.

Let's calculate TF, using the tokenizer to extract worlds and later the HashingTF:

```
[source, python]from pyspark.ml.feature import HashingTF, IDF,
Tokenizer
tokenizer = Tokenizer(inputCol="description", outputCol="words")
wordsData = tokenizer.transform(data)
hashingTF = HashingTF(inputCol="words",
outputCol="frequencyFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
```

In our code sample, we calculate the TF using hashingTF, which has created a new column named *frequencyFeatures* that contains a Sparse vector *of hashed words and their appearance as shown in Figure 3-14*.



```
+-----+--------------------+
|label|   frequencyFeatures|
+-----+--------------------+
|    1|(20,[0,2,3,4,5,7,...|
|    0|(20,[3,13,16,17],...|
|    0|(20,[1,2,4,5,6,7,...|
|    0|(20,[0,1,4,5,7,8,...|
|    1|(20,[0,1,3,4,5,6,...|
+-----+--------------------+
only showing top 5 rows
```

Figure 3-14. hashed term frequency

Our second step is to calculate the IDF. Since Spark IDF is an Estimator, we need to build it first using the *fit* method.

```
[source, python]
idf = IDF(inputCol="frequencyFeatures",
outputCol="features")idfModel = idf.fit(featurizedData)
```

As shown in the snippet code above, we created an `IDFModel` object instance named `idfModel` in our code. Now, we can transform the data using the model:

```
[source, python]rescaledData = idfModel.transform(featurizedData)
```

rescaledData DataFrame instance has a column named features with the calculated importance of a term in a description given the whole database.

Although nice to have information, TF-IDF is an unsupervised method that completely neglects the label of the data.

N-Gram

NGram in MLlib is part of the feature transformer functionalities. It allows us to take an array of strings(words) and convert it into an array of n-grams, where we can define the n in NGram function:

```
[source, python]
from pyspark.ml.feature import NGram
ngram = NGram(n=2, inputCol="words",
outputCol="ngrams")ngramDataFrame = ngram.transform(wordsData)
```

NGram can produce a nice feature if we know what to do with it later and the output provides value. For example, the output of:

```
[I, live, in, texas]
```

Given n=2 provide us with the following array of Strings:

```
[I live, live in, in texas]
```

As you can witness from the output, NGram works as a sliding window, where it repeats words. NGram is useful when we are building tools for autocompletion of sentences, auto spell check, grammar check, extracting topics, etc.

Others

There are many other techniques that you can leverage, such as topic extraction. Topic extraction is based on understanding the topic of a given document, by scanning for known impactful terms, or a combination of terms. This can be achieved, using the previous features shown. You can also leverage Frequent Pattern Mining MLlib functionality: FP-Growth and PrefixSpan. All those methods, at their core, are based on the frequency of terms, identifying patterns frequency in a corpus vs in a given document and NGram, bag-of-words, etc. To learn more, checkout the following reputable sources:

1. Text Analysis with Python

2. Natural Language Processing with Spark NLP
3. Practical Natural Language Processing

Enriching the Dataset

Often, our dataset will require more data points and new features.

Finding the right features is the beating heart of the machine learning workflow, and is truly a work of art. With our Bot or Not, Twitter Bots detection dataset we can leverage the Twitter API for extracting new and fresh tweets and updates. We can also leverage Transfer Learning.

Transfer Learning

> Transfer Learning is the process of leveraging knowledge gained while solving one problem and fitting it into a different problem.

How can we leverage Transfer Learning with Twitter Bots detection?

> Pulling data from other social media with similar *screen_name*. For example, LinkedIn. Learning if an account is real on LinkedIn can provide us with another data point to leverage.

FEATURE STORE: WHAT ARE THEY ALL ABOUT?

In 2017, Uber introduced the concept of Feature Store with Michelangelo. The main requirement of a feature store is to store the features in optimized, relatively fast storage, with comprehensible metadata to allow fast access and query by the machine learning algorithm. As we will discuss in chapter 7, there is also a notion of data types, caching and bridging multiple platforms where feature store can assist us. As for Michelangelo, Uber designed it to support more than just the featurization step. They covered those 6 requirements:

1. Manage data
2. Train models
3. Evaluate models
4. Deploy models
5. Make predictions
6. Monitor predictions

As you might guess by now, Michelangelo is a platform that goes beyond just features. It supports the whole machine learning workflow. As part of training the models, they refer to the Feature Store for offline, batch data. Which supports large-scale distributed training of decision trees, linear and logistic models, unsupervised models (k-means), time series models, and deep neural networks. Interestingly enough, they acknowledge that engineering *good features* is the hardest part of machine learning. Hence, sharing those features using a dedicated store, or storage layer, helped speed up the data science workflow. This is why we advise persisting the data during the training and feature extraction steps, make sure it has version control attached to it and it is shareable.

Summary

With many feature extraction functions implemented in Spark API and Spark MLlib, there might be other features that you might look into when building your model. Realizing that there are multiple features that can improve model accuracy. For example with images, you might want to look into a feature descriptor that only keeps the image outline and very unique characteristics of an image to differentiate one feature of the image from another. Bear in mind that you can leverage Spark generic functions to do anything that you need with code, but be mindful of the resources and the compute costs needed to do so. Domain knowledge is the key for a successful featurization process and it's important to handle feature engineering processes with great care. In Chapter 6 you will learn how to build a machine learning model.

Chapter 4. Training Models with Spark MLlib

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. The GitHub repo is available now at https://github.com/adipolak/ml-with-apache-spark.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jleonard@oreilly.com.

Now that you've learned about Managing Machine Learning experiments, Getting a feel for the data, and Feature Engineering, it's time to train the models.

What does that involve exactly? Training a model is the process of adjusting or changing model parameters so that its performance increases. The idea here is to feed your ML model training data that teaches it how to solve a specific task - for example, classify an object on a photo as a cat by identifying its "cat" properties.

In this chapter, you will learn how algorithms work in ML, when to use which tool, how to test your experiment, and, most importantly, how to automate the process with the Spark ML pipeline library.

Algorithms

Let's start with algorithms, the essential part of your model training activities. The input of a machine learning algorithm is sample data, and its output is a model. The algorithm's goal is to generalize the problem and extract a set of logic for making predictions and decisions without being explicitly programmed to do so. Algorithms can be based on statistics, mathematical optimization, pattern detection, and so on. Spark MLlib provides us with a distributed training implementation for the classic *supervised machine learning* algorithms such as Classification, Regression, and Recommendation. MLlib includes implementations for *unsupervised learning* as well, such as Clustering and Pattern Mining, which are often used to detect anomalies.

> **NOTE**
>
> It is worth noting that there is no feature parity between the MLlib RDD-based API and the MLlib Main library yet, so there are cases where the functionality which you need can be found in the RDD-based APIs. The Singular Value Decomposition (SVD) is an example of such a method. Be aware of that and take appropriate action.

How do you pick the right algorithm for the job? Your choice always depends on your objectives and data.

While this chapter will cover many algorithms and their use cases , the topics of Deep Learning, integration with PyTorch, and TensorFlow distributed strategies (with the exception of some particular examples such as the Multilayer Perceptron classifier) will be discussed in Chapters 7 and 8.

I would like to draw your attention to the fact that the MLlib model instance has dedicated functionality for parameters documentation.

The following code sample illustrates how, you can immediately access the documentation for the individual parameters once you have created an instance of the model:

```
[source,python]
import pprint
```

```
pp = pprint.PrettyPrinter(indent=4)
params = model.explainParams()
pp.pprint(params)
```

Figure 4-1:1 is the outcome of a *model.explainParams() function.* Since this is a GaussianMixture model, it contains the params that are relevant to it. GaussiamMixture is discussed in detail later in this chapter. The purpose of this example is to help you in your educational journey as you explore MLlib algorithms and learn about each one and its model outputs.

```
('aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)\n'
 'featuresCol: features column name. (default: features, current: '
 'selectedFeatures)\n'
 'k: Number of independent Gaussians in the mixture model. Must be > 1. '
 '(default: 2, current: 42)\n'
 'maxIter: max number of iterations (>= 0). (default: 100, current: 100)\n'
 'predictionCol: prediction column name. (default: prediction)\n'
 'probabilityCol: Column name for predicted class conditional probabilities. '
 'Note: Not all models output well-calibrated probability estimates! These '
 'probabilities should be treated as confidences, not precise probabilities. '
 '(default: probability)\n'
 'seed: random seed. (default: 4621526457424974748, current: 10)\n'
 'tol: the convergence tolerance for iterative algorithms (>= 0). (default: '
 '0.01, current: 0.01)\n'
 'weightCol: weight column name. If this is not set or empty, we treat all '
 'instance weights as 1.0. (undefined)')
```

Figure 4-1. Example of Pretty print model doc params of GaussianMixture

Now that we've covered the basics, our learning journey for this chapter starts with supervised machine learning. Let's dive in.

Supervised Machine Learning

All supervised algorithms expect to have a label column as part of the data. This allows the algorithm to "validate" itself during the training phase and estimate how well it performs. So, the label column is what the algorithm uses to correct its decisions. In the testing phase, we use it to assess the quality of the algorithm by comparing the model predicted to the real outcome. The label can be a discrete/categorized variable, a specific value

among a set of all possible values - for example, an apple when you are categorizing between apples and oranges. Or it can be a continuous value, such as a person's height. The difference between them is the task we want our model to solve.

In some cases, the label itself might be a set of labels. That's what we will talk about in the next section.

Classification

Classification is the task of calculating the probability of discrete categories by examining input features. The output of that process is the probability of being attached to a certain category, also known as class. Many practitioners often confuse Regression and Classification due to the existence of Logistic Regression. While Logistic Regression outputs the probability of a discrete class, similar to classification, other Regression algorithms are used for predicting a continuous numeric value. Pay attention to this difference!

There are 3 types of classification:

Binary

One label, two classes to choose from: yes or no, true or false. Algorithms often expect an indexed label in the range of [0,1].

MultiClass

One label, more than 2 classes to choose from.

MultiLable

Each given input can have multiple labels in practice. For example, a sentence can have two sentiment classifications, happy and content. Spark does not support it out of the box. For that, you will need to train each classifier separately and combine the outcomes.

What also impacts the classification process is the distribution of classes in the training data.

Imbalanced

> Imbalanced refers to the available labeled data. Data labels are said to be unbalanced when the classes of the input data are distributed unevenly. You can often see it in tools for fraud detection and medical diagnosis, we would need to consider and weigh our features accordingly when possible when facing such scenarios.

Indexed labels and vectors

All those algorithms expect an indexed label and vector of indexed features. IndexToString, StringIndexer, VectorIndexer, and other transformators were discussed under *Feature Engineering* in Chapter 5.

After training, each algorithm produces a model and often has the class pattern name of: {name}ClassificationModel, {name}Classifier. For example: GBTClassificationModel and GBTClassifier classes. Or just the names of the algorithms, for example: NaiveBayes and NaiveBayesModel, depending on whether the name was shortened in the beginning or not.

API	Usage
LogisticRegression	Binary and multiclass classification, can be trained on streaming data with the RDD-based API.
DecisionTreeClassifier	Multiclass decision tree classifier. It expects an indexed table and vector of indexed features.
RandomForestClassifier	Multiclass. Part of the ensemble tree classifiers for discrete values. It expects an indexed table and vector of indexed features.
GBTClassifier	As of Spark 3.1.1, it supports binary Gradient Boost Tree Classifier. Similar to the Random forest classifier, it treats features with > 4 distinct values as continuous. As a result, it can be used for regression as well.
MultilayerPerceptronClassifier	Multiclass, based on feedforward artificial neural network, expects layers sizes, vectors of indexes as features, and indexed labels.
LinearSVC	Linear Support Vector Machine, binary classifier, expects vectors of indexes as features and indexed labels.
OneVsRest	Can be used for imbalanced data given a binary classifier. It expects a binary classifier, vectors of indexes as features, and indexed labels.
NaiveBayes	Multiclass classifier. Considered efficient as it runs only one pass over the training data. It expects a boolean for the weight of a data entry, indexed label,

	and vector of indexed features. When used, returns the probability for each label.
FMClassification	Binary Factorization machines classifier. It expects indexed labels and vectors of indexed features.

MLlib doesn’t support multilabel out of the box, what can I do about it?

Alright, since MLlib doesn’t support multilabel out of the box, there are multiple approaches we can take here:

1. Search for another tool that does and can train on a large data set.
2. Train binary classifiers for each of the labels and output multilabel by running relevant-irrelevant predictions for that label.
3. Think about another way how to leverage the existing tools and break the task into pieces to solve each independently and then combine them together using code.

There is good news for you! PyTorch and TensorFlow are both capable of supporting multilabel classification algorithms. As such, we can take advantage of their capabilities also for multilabel cases.

As for the second option, if you are an AI Engineer or an experienced Spark Developer, Spark provides a rich API you can use to execute the next steps:

1. Add multiple labels to the existing DataFrame that represents the label. DataFrame output example for multilabel image::images/ch03/ch03-withcolum-for-mutilable.png[DataFrame output example for multilabel]
2. Continue the featurization process for each dedicated label. In the book’s GitHub repo we used HashingDF, IDF, and others to reach this state, as shown in Example 6-1.

Figure 4-2. Example 6-1

DataFrame is ready for training the first classifier for the Happy label

Build a Binary classifier for each one of the labels. This code snippet shows you how to build LogisticRegression classifier after the transformations of adding columns and indexing:

```
[source,python]
from pyspark.ml.classification import LogisticRegression
happy_lr = LogisticRegression(maxIter=10, labelCol="happy_label")
happy_lr_model = happy_lr.fit(train_df)
```

Now we can do the same process to the rest of the labels.

Remember, there are more steps to the machine learning pipeline, like evaluating the outcome with testing datasets and using them together.

Take a look at Figure 4-2, which shows the output of testing a model:

```
Spearman correlation matrix:
DenseMatrix([[ 1.        , -0.41076061, -0.22354106,  0.03158624],
             [-0.41076061,  1.        , -0.15632771,  0.16392762],
             [-0.22354106, -0.15632771,  1.        , -0.09388671],
             [ 0.03158624,  0.16392762, -0.09388671,  1.        ]])
```

Spearman correlation matrix

This output is a DenseMatrix that holds the following columns as we discussed in chapter 4.

rawPrediction

Vector of doubles that are the measure of confidence to each possible label.

probability

The probability vector. Please note that not all models output accurate probabilities. As a result, they may not be precise and should be used with caution.

prediction

The prediction itself.

What about imbalanced class labels?

You might be wondering what the term "imbalanced data" means. It refers to datasets in which the target class has uneven numbers of observations. You can find them in financial fraud detection scenarios, healthcare areas like identification of a certain disease, or when the data is ingested into the system from multiple sources .

A one-class label with a very high number of observations and another with a very low number of observations can produce a biased model. The bias will be toward the class label with a high number of observations, as it is statistically more dominated in the training dataset.

It is possible to introduce bias in various phases of the model's development. Insufficient data, inconsistent data collection, and poor data practices can all lead to bias in the model's decisions. We are not going to delve into how to solve the existing model bias, but rather focus on the strategies of working with the existing dataset to mitigate the potential bias.

You can use multiple strategies to mitigate potential bias in the existing training set:

1. Filtering the more representative classes and sampling them to downsize the numbers of entries in the overall dataset.
2. Using dedicated algorithms such as GBTClassifier, GBTRegressor, RandomForestClassifier, and others. These are from the tree ensemble family discussed earlier in this chapter. During the training process, the algorithm has a dedicated feature subset strategy parameter featureSubsetStrategy with supported values of are auto, all, sqrt, log2 and onethird. With the default of "auto" parameter, the algorithm is choosing the best strategy to go with based on the given features. In every tree node, the algorithm is processing a random subset of features and users it to builds the next node. It iterates on the same procedure until it finishes using all the datasets. This is useful because of its random approach to the parameters, yet there still might be cases of bias, depending on the observation distributions. Imagine that you have a dataset with 99 apples and 1 orange. Suppose in a random process, the algorithm picks a batch of 10 units, and it includes 1 orange and 9 apples. The distribution remains heavily skewed towards apples. So, the results of the model will be that 100% of all predictions would be apples, which is correct for a given training dataset but might be completely off the mark in the real world. You can read more about it in the documentation.

Here is how to set the strategy:

```
[source,python]
from pyspark.ml.classification import RandomForestClassifier
# Train a RandomForestClassifier model with a dedicated feature
subset strategy
rf = RandomForestClassifier(labelCol="label",
featuresCol="features",featureSubsetStrategy="log2")
model = rf.fit(train_df)
```

Regression

It's time to learn about Regression! This is also known as Regression Analysis - the process of estimating variable relationships between dependent variables given an input of independent variables. To simplify it,

our independent features are used to determine the dependent variables. If that is not the case, you can use the API in featurization to select only the features that add value.

From a bird's-eye view, there are 3 types of regression:

Simple

> There is only one independent and dependent variable: one value for training and one to predict.

Multiple

> Here we have one dependent variable to predict using multiple independent variables for training and input.

Multivariate

> Similar to multilabel classification, there are multiple variables to predict with multiple independent variables for training and input. Accordingly, the input and output are vectors of numeric values.

For the simple and multiple cases, you might see that many algorithms show up for both regression and classification use cases. This is because they support both discrete and continuous numerical predictions.

There is no dedicated API available for Multivariate cases as of today. So, you would need to architect your system to support it. This process is similar to what we did for MultiLabel. Prepare the data, train for every variant independently, test and tune multiple modules, and later assemble the desired predictions.

To learn about Regression, we are going to try and predict a vehicle's CO2 emission using Kaggle's dataset. We are going to look at features such as company, car model, engine size, fuel type, consumption, and more!

As you will probably notice, working with data requires featurization, cleaning, and formatting it to fit into the algorithm.

There are 13 columns in the data. To speed up the process of indexing and hashing, we use FeatureHasher only on the continuous features with UnivariateFeatureSelector. That is because this selector asks us to specify the nature of the numeric features - discrete or continuous.

```
[source,python]
from pyspark.ml.feature import FeatureHasher
cols_only_continuous = ["Fuel Consumption City (L/100 km)", "Fuel
Consumption Hwy (L/100 km)",
        "Fuel Consumption Comb (L/100 km)"]
hasher = FeatureHasher(outputCol="hashed_features"
inputCols=cols_only_continuous)ƒthis i
co2_data = hasher.transform(co2_data)
```

Notice how nicely inputCols takes a list, this makes reproducing and developing cleaner code easier for us!

hashed_features is of type SparseVector:

```
+-----------------------------------------+
|hashed_features                          |
+-----------------------------------------+
|(262144,[38607,109231,228390],[0.0,9.9,6.7]) |
|(262144,[38607,109231,228390],[0.0,11.2,7.7])|
|(262144,[38607,109231,228390],[0.0,6.0,5.8]) |
|(262144,[38607,109231,228390],[0.0,12.7,9.1])|
|(262144,[38607,109231,228390],[0.0,12.1,8.7])|
+-----------------------------------------+
only showing top 5 rows
```

Figure 4-3. Hashed-features sparse vector

Look at Figure 4-3. Due to the complexity of the hashing function, we ended up with a 262144 size vector. This needs to improve since most of the vectors are sparse and might not be meaningful for us.

So, it's time to select the features automatically!

```
[source,python]
from pyspark.ml.feature import UnivariateFeatureSelector
selector = UnivariateFeatureSelector(outputCol="selectedFeatures",
featuresCol="hashed_features", labelCol="co2")
selector.setFeatureType("continuous")
selector.setLabelType("continuous")
model = selector.fit(co2_data)
output = model.transform(co2_data)
```

The selector reduced the number of features from 262,144 to 50.

Increased dimensions

Note that we actually increased the dimensions with FeatureHasher. This is because we didn't normalize the data at first to make it simpler for us to backtrack the experiment. For real cases, it is best to normalize the data before.

The next step is to build the machine learning model. When we look at the documentation and what MLlib provides, there are multiple algorithms we can choose from.

AFTSurvivalRegression AFT stands for accelerated failure time and can be used to discover how long a machine in a factory will last. Another one is the DecisionTreeRegressor - but don't be perplexed by the name regressor. It operates at its best on categorical features, which have a finite number of categories. As a result, it won't be able to predict unseen values like other regressors. GBTRegressor is an ensemble gradient boosting trees regressor that splits the train data into train dataset and validation dataset. The validation dataset is used in the boost loss function to remove the error on every iteration of the algorithm over the data.

You are probably curious how they differ from RandomTrees that we've seen in Classification? The main difference is that GBT builds one tree at a time that helps correct the previous one, while RandomForestClassifier

builds the trees randomly in parallel (where every subset of worker nodes builds their own tree that is being collected in the main nodes). In this instance, the main node assembles the worker's output to the final model. Both GBTRegressor and RandomForestClassifier support continuous and categorical features.

In the next example, we use GBTRegressor as it might perform better. While training might take longer due to its sequential nature, we're using it due to the optimization function.

```
[source,python]
from pyspark.ml.regression import GBTRegressor
# defining the classifier
gbtr = GBTRegressor(maxDepth=3, featuresCol="selectedFeatures",
labelCol="co2")
#building the model
model = gbtr.fit(data)
# using the model
test01 = model.transform(data)
```

Now that we have a model, we ingest it with the data we will use to train it. We need to validate that there is no overfitting. If test01 predictions are 100% accurate, that means overfitting. In ML Pipelines, we will learn about evaluating models. For now, we will look at a sample from prediction column output

Figure 4-4. Predicted vs. Actual CO2 Emission of vehicles

MLlib supports other machine learning algorithms that can solve this problem, for example, FMRegression - Factorized Machine Learning Regression. It is based on a gradient descent algorithm with a dedicated loss

function, also called an optimization function. Gradient Descent is an iterative optimization algorithm. It iterates over the data and together with the loss functions, searching for rules or definitions that provide minimum loss of accuracy. Presumably, it improves with every iteration.

The FMRegression max iteration default is set to 100 and we can tweak it using the setMaxIter functionality. The optimization function used here is SquaredError. SquaredError implements the MSE functionality that calculates the overall Mean Square Error in every iteration and the one the algorithm seeks to reduce. It is the error sum of squares of "distances" between the actual value and the predicted value in a given iteration. MSE is considered an unbiased estimator of error variance under standard assumptions of linear models.

If FM sounds familiar, it is because there is also an FMClssfier where the main difference between them is the loss function. The classifier uses LogisticLoss. This is sometimes referred to as entropy loss or log loss. The LogisticLoss functionality is used in LogisticRegression as well. We will not dive into the theoretical math behind it as there are many books that cover the introduction to machine learning. But it is important that you grasp the similarities between Classification and Regression algorithms.

Recommendation Systems

Recommendation systems are often taught on a movie dataset, such as MovieLens, where the objective is recommending movies to users based on what other users liked and/or user preferences like genres. You can find recommendation systems implemented in many online platforms - for instance, e-commerce systems such as Amazon or move streaming platforms like Netflix. They are based on *Association Rule Learning* where the algorithm aims to learn the association between the movies and users.

At a high level, we can split them into 3 categories depending on the data available (metadata and interaction matrix):

Content-based

Algorithms use the metadata of content and users, including what content the user has watched before and rated, and produce a recommendation based on it. This can be implemented with rule-based functionality and doesn't necessarily require machine learning. User rating, favorite genre, movie genre, and so on, are considered metadata.

Collaborative- filtering

In this instance, no metadata is available about the movies and users. We only have interaction matrix data between the users and movies. The algorithm would filter search for similarities between the user interactions to provide a recommendation.

Neural- Networks

Given metadata on the user and content together with the interaction matrix, you can leverage neural networks (since all this data exists).

ALS for Collaborative-filtering

MLlib provides a well-documented solution for the collaborative filtering algorithm called Alternating Least Square (ALS). Its goal is to fill out missing values in a user-item association matrix. It also provides a solution to cold start scenarios where the user is new to the system and there is no previous data. You can read more about it in Spark Docs.

Unsupervised Machine Learning

Unsupervised algorithms are used when the data doesn't have a label. Yet, we still want to automatically find interesting patterns, predict behaviors, or calculate resemblance without knowing the desired outcome. Those algorithms can be used interchangeably with supervised algorithms as part of the feature extraction procedure. Let's get started!

Frequent Pattern Mining

Frequent pattern mining belongs to the Association *Rule Mining* set of algorithms that are part of the *Association Rule Learning*. Since it follows the basics of *antecedent (if)* and *consequent (then)*.

MLlib provides two functionalities that can be used as preprocess procedures for the recommendation, like extracting meaningful patterns out of a corpus of text to detect user sentiment towards a movie. You can read about FPGrowth and PrefixSpan in PySpark 3.1.1 documentation.

Clustering

Clustering is a grouping technique for discovering hidden relationships between data points. Clustering is regularly used for customer segmentation, image processing and detection, spam filtering, anomaly detection, and more.

In the clustering process, every item is attached to a specific random group. The group is defined by its center. The likelihood of an item belonging to a group is calculated by its distance from the center. The algorithms usually try to optimize the model by changing the group centers.

k is a known word in clustering algorithms like k-nn and k-means. Its meaning depends on the algorithm itself. It often represents the number of predefined clusters/topics. MLlib algorithms have a default Integer K value, and you can set it using the method setK. Some algorithms require the data to have weighCol, specifically KMeans, GaussianMixture, PowerIterationClusting'and `BisectingMeans expects a nonnegative weighCol in the training dataset that represents the data points weights relate to the center of the cluster.

If a specific data point has a high weight and is relatively far from the cluster center, the "cost" on the optimization function would be high. The algorithm would prioritize moving the cluster center to optimize the overall clustering. Almost all require seed values (except for PowerIterationClusting). The seed value is used to initialize a set of cluster center points at random (think x and y coordinates). And with every

iteration of the algorithm, the centers are changing based on the optimization function.

Now that you know what clustering is, let's return to our CO2 Emission prediction objective and try to find commonalities between columns such as fuel type, consumption, cylinders, etc.

LDA

Exploring what MLlib provides, we can see multiple algorithms. One of them is Latent Dirichlet Allocation (LDA). This is a generic statistical algorithm used in evolutionary biology, biomedicine, and natural language processing.

The algorithm expects a vector representing counted words in a document. But since in our scenario we focus on fuel types, etc., LDA does not match our data. The classic and most known one is KMeans, which takes any unlabeled data and provides it with a predicted label, which is the group it belongs to. This can be a great option for us. Still, let's explore the other options.

GaussianMixture

GaussianMixture is often used to demonstrate the presence of a group within a bigger group. It can be useful when we want to get the car classes' presence inside each car manufacturer's group - like the compact car class in the Audi group vs. compact cars in the Bentley cars group.

However, GaussianMixture is said to perform poorly on high-dimensional data, making it hard for the algorithm to converge to a satisfying conclusion. High dimensional data is when the number of features/columns is significantly larger than the number of rows. For example, if I have 5 columns and 4 rows, my data is considered highly dimensional. In the world of large datasets, this is less likely to be the case.

KMeans

KMeans is the most popular algorithm for clustering, due to its simplicity and efficiency. It takes a group of classes, creates random centers, and starts iterating over the data points and the centers. Its quality depends on the k and the number of iterations.

BisectingKMeans

BisectingKMeans is based on the KMeans algorithm with the hierarchy of groups. It supports calculating distance in the euclidean way or the cosine way. The model can be seen as a tree, with leaf clusters. Where during training there is only a root node, and nodes are split to optimize the models. It's great if you want to represent groups and subgroups.

PowerIterationClustering (PIC)

PowerIterationClustering implements the Lin and Cohen algorithm. Note that this cannot be used in Spark pipelines as it does not yet implement the Estimator/Transformer pattern. More on that in the ML Pipeline later on.

Cool, cool, cool! Now that we understand our options, let's go with the GaussianMixture since our dataset has only 11 columns and much more data than that. We are going to use the transformations from before with all the columns, including the labels.

The number of k would be the number of car manufacturers in our data. To extract it, we use distinct count.

```
[source,python]
dataset.select("Make").distinct().count()
```

The result is 42. What an interesting number. :)

```
[source,python]
from pyspark.ml.clustering import GaussianMixture
gm = GaussianMixture(k=42, tol=0.01, seed=10,
featuresCol="selectedFeatures", maxIter=100)
model = gm.fit(dataset)
```

Now that we have the model, we can get the summary object that represents the model.

```
[source,python]
summary = model.summary
```

All clustering and classification algorithms have a summary object. In clustering, it contains the predictions, easy access to the cluster overall predictions, the cluster size - the number of objects in every cluster, and dedicated params based on the specific algorithm.

For example, we can understand how many groups the algorithm has converged to at the end by running a distinct count using the summary:

```
summary.cluster.select("prediction").distinct().count()
```

In our case, we received 17. That means we can reduce the number of k to get better convergence or increase the number of iterations and check if that number changes. Of course, the more iterations the algorithm does, the more processing time it takes. Remember that you are running on a large set of data, so be cautious with the amount. *How can I know how many iterations I need?* is a question of trial and error. Be sure to add it to your experiment testing together with the performance measurement.

Another way to measure is by looking at the logLikelihood that represents the statistical significance of the difference between the groups.

```
[source,python]
summary.logLikelihood
```

With 200 iterations, we received about ~508,076 likelihood scores. This score is not normalized and it is hard to compare it to others. All we know is that a higher score means a greater likelihood of instances being related to their cluster. You can see now that basing your model performance on likelihood won't give you the best measure for the overall problem, but the measure of one model vs. another one with the same data. So, make sure to set up the objectives of the experiments first. If you are curious to learn more about statistics and likelihood measurement, we recommend reading O'Reilly's Practical Statistics for Data Scientists book.

To continue exploring, we used maxIter=200 and received the same amount of distinct prediction - 17. Let's change *k* to 17.

We can examine the cluster size, to make sure there are no groups with zero data points in them:

```
[source,python]
summary.clusterSizes
```

Our clusterSizes output:

Cluster size of every group, where the indices represent the group index.

Figure 4-5. Cluster size of every group, where the indices represent the group index

Since clusterSizes is of type Array, you can use statistical tools such as numpy and metaploit to create a histogram.

Figure 4-6 shows the distribution of group/cluster sizes, using a histogram.

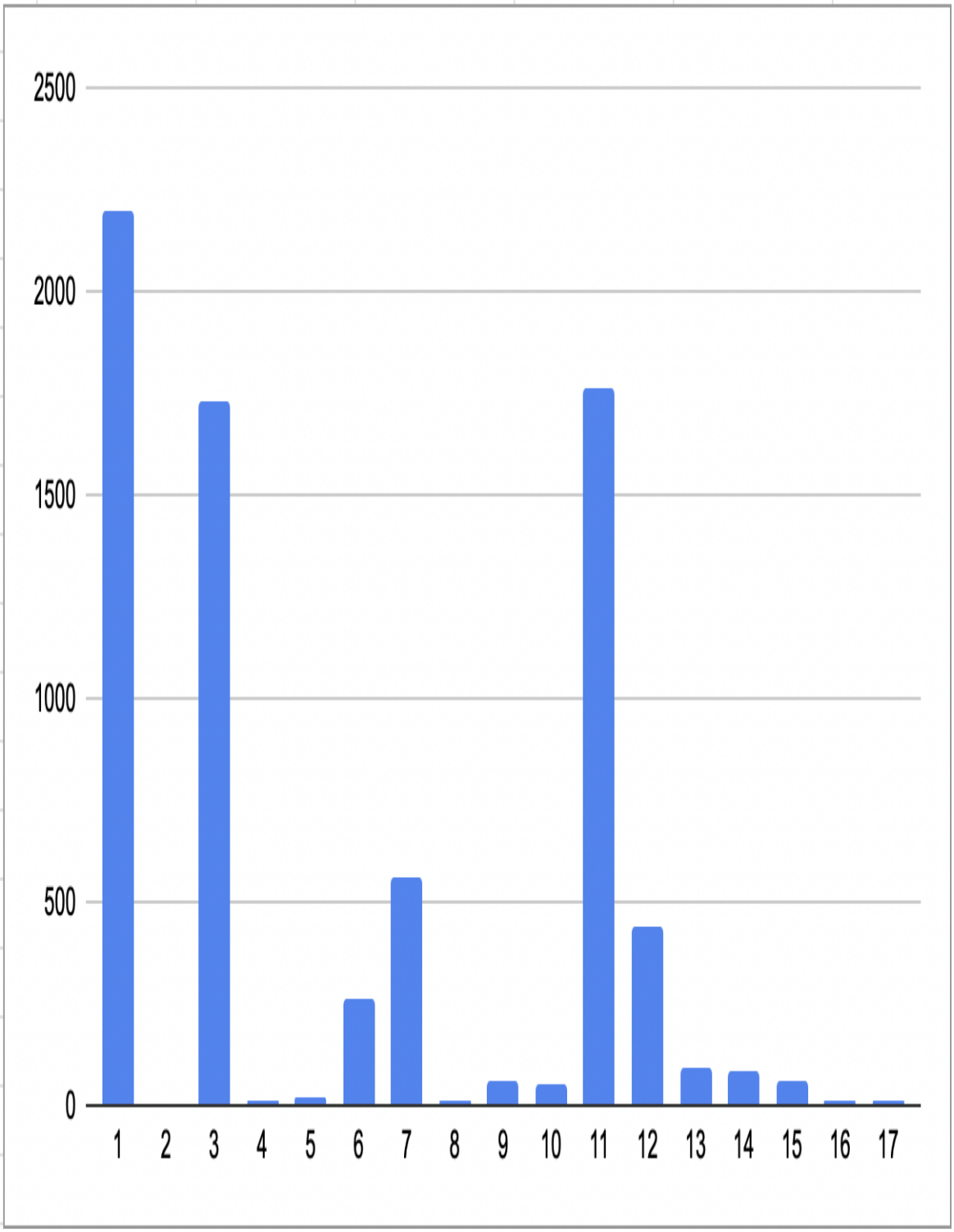


Figure 4-6. Histogram of clusters or group sizes

Evaluating

The evaluation phase is an essential part of the machine learning process. Evaluating is our way of estimating the model performance. Now that we know how to put everything together, it's time to discuss how to train and evaluate your model.

MLlib has 6 evaluators, all of which implement the Spark abstract class *Evaluator*. They can be roughly split into supervised and unsupervised.

Evaluator

A class that allows us to see how a given model performs according to specific machine learning evaluation criteria.

Supervised Evaluator

In supervised learning, we have the test dataset label so we can compare the actual outcome with the label and produce multiple metrics for estimating performance. To do so, the estimator first calculates the confusion matrix.

What is the confusion matrix?

This is a matrix that compares the predicted label and the actual label.

For a binary, it would look like this, with each box having a range of [dataset size,0].

Binary confusion matrix

True and False stands for the accuracy of the prediction, and Positive and Negative stands for the binary prediction (can also be 1 and 0):

True Positive (TP): # of positive labels where prediction is also positive.
True Negative (TN): # of negative labels where prediction is also negative.
False Positive (FP): # of negative labels but the prediction is positive.
False Negative (FN): # of positive labels but the prediction is negative.

Based on those values, estimators provide many matrices, and you can find them all in the documentation.

When we want to use the same process for *multiclass* and *multilabel* classification, the confusion matrix will grow accordingly to capture all of the labeling possibilities. In the case of *imbalanced data*, you would probably need to write your own evaluator. You can do that using the Spark extensive API.

BinaryClassificationEvaluator

Binary evaluator that expects input column rawPrediction, label, and an optional weight column. It has a special functionality of outputting an area under Receiver Operating Characteristic (ROC) and Precision-Recall (PR).

MulticlassClassificationEvaluator

Evaluator for Multiclass Classification, which expects input columns: prediction, label, weight (optional), and probabilityCol (only for logLoss). It has dedicated metrics such as precisionByLabel, recallByLable, etc, and a special matrix hammingLoss that calculates the fraction of wrong labels versus the total number of labels. When comparing binary and multiclass models, remember that Precision, Recall, F1-Measure are designed for the binary class, therefore it is better to compare the hammingLoss to accuracy.

MultilabelClassificationEvaluator

This evaluator for Multilabel Classification is relatively new and was added in Spark 3.0. It is currently in an experimental phase[1]. It expects two input columns: prediction and label vector. It also has dedicated metrics such as microPrecision, microRecall, etc. that are an average on the aggregation of all prediction classes.

RegressionEvaluator

It expects input columns prediction, label, and an optional weight column. And produces metrics such as mse - root mean square error of the distances between the predictions and the actual labels, rmse - the root of the previous value and others.

RankingEvaluator

Added in Spark 3.0. This evaluator expects prediction and label columns. It often allows for evaluating the ranking of search results. It has the variable k that you can set to get the matrix averages by the specific K first results. Think about that movie recommendation system where the output can be 5 or 10 recommendations. The average would change and evaluating the results can help you make an informed decision. This evaluator output is based on MLlib RDD-based API - RankingMatrix, you can read more about it in its docs.

Unsupervised Evaluator

For unsupervised learning, there are various methods you can choose from. We could focus on what is available in MLlib, however, bridging to TensorFlow, PyTorch, or other libraries that can process Spark DataFrames would present more out-of-the-box evaluators.

ClusteringEvaluator

Evaluator for Clustering results, which expects two input columns: prediction and features, and an optional weight column. It computes the *Silhouette measure*, where you can choose between two distance measures: squaredEuclidean and cosine. Silhouette is a method to evaluate the consistency within the cluster and how similar the data is in every cluster. It does that by calculating the distance between each data point to other data points in the cluster and also from other points out of the cluster. The output is the mean Silhouette values of all points based on points weight.

Reflecting on our classification example with GaussianMixture and KMeans, we can evaluate the models to make better decisions. The full code snippet is in the book GitHub repository in ch_03_clustering.

```
[source,python]
from pyspark.ml.evaluation import ClusteringEvaluator
evaluator = ClusteringEvaluator(featuresCol='selectedFeatures')
```

```
evaluator.setPredictionCol("prediction")
print("kmeans: "+str(evaluator.evaluate(kmeans_predictions)))
print("GM: "+ str(evaluator.evaluate(gm_predictions)))
```

The default distance compute method is the square euclidean, it gave the following output:

```
kmeans: 0.7264903574632652
GM: -0.1517797715036008
```

How can we tell which one is better? The evaluator has a dedicated functionality named isLargerBetter() that can help us determine the output:

```
[source,python]
evaluator.isLargerBetter()
```

In our case, it returned True. We are not done yet, let's take a look at the cosine distance output:

```
[source,python]
evaluator.setDistanceMeasure("cosine")
print("kmeans: "+str(evaluator.evaluate(kmeans_predictions)))
print("GM: "+ str(evaluator.evaluate(gm_predictions)))
```

output:

```
kmeans: 0.05987140304400901
GM: -0.19012403274289733
```

Now, this is not an indicator yet, considering that the models themselves are built with specific distance measures in clustering. For example, K Means was built with euclidean distance and has most likely performed better when evaluating based on square euclidean distance. For adding more clarity, we can combine the evaluating process with multiple tuning of the test-train data sets, the algorithms, and the evaluators. Time for tuning!

Hyperparameters and Tuning Experiments

What if I told you that there are tools for running multiple experiments, producing numerous models, and extracting the best one? This is precisely what we are going to learn in this section!

It's true that all machine learning processes require tuning and experimenting to reach good results. Tuning experiments are defined by splitting the dataset into multiple trains and test sets and/or tweaking the algorithm parameters.

It's time to learn how to do it comprehensively, grid params, and split functionality.

Take a look at this:

```
[source,python]
from pyspark.ml.tuning import  ParamGridBuilder
grid = ParamGridBuilder().addGrid(kmeans.maxIter, [20,50,100]) \
        .addGrid(kmeans.distanceMeasure,
['euclidean','cosine']).build()
```

In this code, we built a grid param using ParamGridBuilder() by using addGrid for defining multiple max iterations for building k-means models.

Before continuing, let's tweak the params of our evaluator as well by adding a dedicated grid for it:

```
[source,python]
grid = ParamGridBuilder().addGrid(kmeans.maxIter, [20,50,100]) \
        .addGrid(kmeans.distanceMeasure, ['euclidean','cosine']) \
        .addGrid(evaluator.distanceMeasure,
['euclidean','cosine']).build()
```

Notice that grid itself is a Python array of maps, and ParamGridBuilder is just a tool that allows us to build it faster. You can use it with every *MLlib* functionality that takes in an array of params.

Simple Split of Train-Test Sets

Next, let's define a TrainValidationSplit that evaluates each param combination once.

```
[source,python]
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit(estimator=kmeans,
estimatorParamMaps=grid, evaluator=evaluator,
collectSubModels=True, seed=42)
tvs_model = tvs.fit(data)
```

TrainValidationSplit splits the data to one set of *train set* and *test set* with the default ratio of 75% training, 25% testing. You can change it by setting param trainRatio on initialization. TrainValidationSplit is an *Estimator* so it implements fit and outputs a *Transformer*. tvs_model represents the best model that came out after validating various params combinations. We can also specify the TrainValidationSplit to collect all the submodels that perform poorly. We do it by setting the collectSubModels param.

Use collectSubModels with caution. Consider this option for stacking multiple machine learning models as part of your workload or when the evaluation matrix holds similar numbers. Or when you get similar results for the validation metrics and want to keep all the models for continuing the experiment. More on that soon!

How can you tell? Let's check the validation matrix output:

```
[source,python]
tvs_model.validationMetrics
```

This gives us a view of how the various experiments performed given our evaluator.

```
[0.04353869289393124,
 0.04353869289393124,
 0.6226612814858505,
 0.6226612814858505,
 0.04353869289393124,
 0.04353869289393124,
 0.6226612814858505,
 0.6226612814858505,
 0.04353869289393124,
 0.04353869289393124,
 0.6226612814858505,
 0.6226612814858505]
```

Figure 4-7. —Validation Metrics output

Remember that every param in the process can be added to ParamGridBuilder. You can also set the specific columns for labels and features using it. If you have a very big table with many columns, you can use it to tune your experiments using the baseOn functionality.

How can I access various models?

Good question. Remember we set *collectSubModels=True*? This saved all the sub-models, and you can access them from the subModels instance, using this snippet code. But bear in mind that this operation will brain all the information from the executors to the driver and if you are working with a large set of data, might cause out-of-memory exceptions. So for educational purposes, this is fantastic—but *better avoid using* this feature for real-world cases.

```
[source,python]
arr_models = tvs_model.subModels
```

arr_models is a Python array instance that holds our models and some metadata. The array index corresponds to the index of the ValidationMatrics we've seen before.

```
: [KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=euclidean, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50,
   KMeansModel: uid=KMeans_0695dc19e9ae, k=2, distanceMeasure=cosine, numFeatures=50]
```

Figure 4-8. Shows how an output of Sub Models Array would look like

While tvs_model instance itself has access to all the submodels. When we use the instance itself for testing or predicting, it picks the best performing model out of all the sub models produced. When multiple models performed equally, as in our case, it takes the first one in the list.

Cross Validation: A Better Way to Test Your Models

You might have noticed that we cannot tune the TrainValidationSplit to try out multiple combinations of data split. For that, MLlib provides us with

CrossValidator. CrossValidator is an *Estimator* that implements k-fold cross-validation, a technique that splits the dataset into a set of non-overlapping randomly partitioned pairs/folds used separately as [train,test] sets. This operation is considered more computation-heavy since when using it, we train k times parameters maps models. In MLlib k is named folds, so if my numFolds is 3, and I have one parameter grid with 2 values, I train 6 machine learning models.

With numFolds=3 and the previous param grid (for the evaluator and the algorithm), I would train numFolds times grid size which is 12. evaluator.distanceMeasure: 2 options, kmeans.distanceMeasure is 2 and kmeans.maxIter is 3 options hence, grid size is $(\2*2*3\)$. Overall $(\numFolds*gridSize = 36\)$.

This is how we define it:

```
[source,python]
from pyspark.ml.tuning import CrossValidator
cv = CrossValidator(estimator=kmeans, estimatorParamMaps=grid,
evaluator=evaluator,
                           collectSubModels=True,  parallelism=2,
numFolds=3)
cv_model = cv.fit(data)
```

This interface is similar to the previous example.

Similar to TrainValidationSplit and validationMetrics, CrossValidatorModel has avgMetrics params that holds the average evaluations across the grid pram. For every grid params combination, it holds the average evaluation of the numFold. parallelism is used for evaluating models in parallel, when set to 1, its sequential evaluation. parallelism has the same meaning in TrainValidationSplit.

Executing cv_model.avgMetrics would result in: Model Training Average evaluation Metrics output: image::images/ch03/ch03-model-avgMetrics.png[Model Training Average evaluation Metrics output]

Every cell in the array avgMetrics is being calculated using the next equation:

```
$$\begin{equation} {averageEval = \frac{sum{EvalForFold}
{NumOfFold}}} \end{equation}$$
```

ML Pipelines

In this section, you are going to learn how to provide structure to the machine learning pipeline building blocks: featurization, model training, model evaluation, algorithm tuning and persistence. Figure 4-9 is part of O'Reilly's Spark: The Definitive Guide and visualizes the process well. Inserting data from multiple sources, preprocessing, cleaning, transforming, building and tuning the model, and evaluating it.

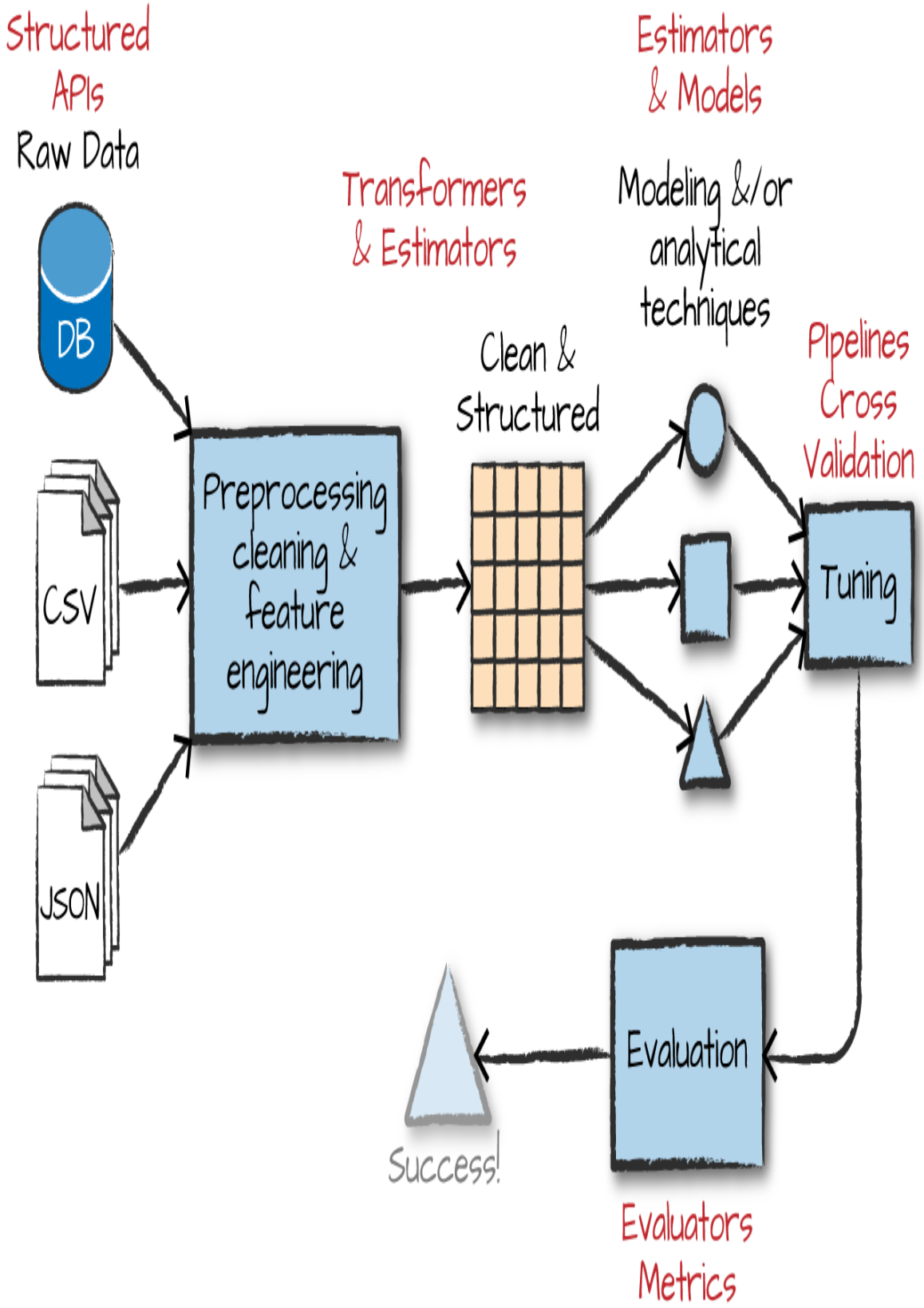


Figure 4-9. ML workflow in Spark, courtesy of O'Reilly's Spark: The Definitive Guide

Pipelines provide a uniform set of API built on top of *DataFrame* and *DataSets*:

Transformer

A function that converts data in some way. It can be a machine learning model or feature transformer that takes in a *DataFrame* and outputs a new *DataFrame* with an appended column. It can take in Spark *Parameters* as well, like in the next example. *Parameters* allow us to influence the function itself, including the names of input and output columns. Providing temporary parameters enables us the flexibility of evaluating and testing multiple variations using the same object.

How to identify a Transformer?

Transformer implements the function Transformer.transform().

Estimator

An object that abstracts the concept of any algorithm that fits or trains on data. An Estimator outputs a model or a Transformer. For example, a learning algorithm such as GaussianMixture is an Estimator, and calling `fit() " - it trains a GaussianMixtureModel, which is a Model and thus a Transformer. Same as the Transformer, the Estimator can take *Params* input.

How to identify an Estimator ?

An Estimator implements the function Estimator.fit().

Going back to our CO2 example, our UnivariateFeatureSelector is an Estimator. It takes in column hashed_features and produces a new *DataFrame* with appended updated columns called selectedFeatures as we specified in outputCol and featureCol.

```
[source, python]
selector = UnivariateFeatureSelector(outputCol="selectedFeatures",
featuresCol="hashed_features", labelCol="CO2")
```

```
model_select = selector.fit(data)
transformed_data = model_select.transform(data)
```

Calling fit() on the Estimator created an instance of UnivariateFeatureSelectorModel that we assigned to model_select. model_select is now a *Transformer* that we can use to create new *DataFrame* with appended column selectedFeatures

Both *Estimators* and *Transformers* on their own are stateless. This means that once we use them to create a model or another Transformer instance, they do not change or keep any information about the input data. They solely retain the parameters and the model representation.

For machine learning, that means that the model state does not change with time. Hence, for online/adaptive machine learning, where new real-time data becomes available in sequential order and is used to update the model, you can use PyTorch, TensorFlow, or other frameworks.

Constructing

In machine learning, it is common to run multiple algorithms in a sequential order to produce and evaluate a model. MLlib provides a dedicated *Pipeline* object for constructing sequential stages.

Constructing machine learning stages in sequential stages. A *Pipeline* object is an *Estimator* with a dedicated stages param.

Looking at our previous example, we had a hasher, selector, and Gaussian mixture algorithm. Now, we are going to construct them together by assigning an array to pipeline stages.

```
[source,python]
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[hasher,selector, gm])
# Fit the pipeline to training data.
model = pipeline.fit(data)
```

Remember to set the input and output columns accordingly to the stage's sequence! For example, the hasher output columns are a selector input, and/or one of gm input. Try and produce only the columns you would use.

How Does Split Work with Pipeline API?

Since the pipeline instance is an Estimator, we can pass the pipeline as an Estimator. So all the splits we learned earlier, like CrossValidator, are available for us here as well.

This is how we can construct the two together:

```
[source,python]
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[hasher,selector, gm])
cv = CrossValidator(estimator=pipeline, estimatorParamMaps=grid,
evaluator=evaluator,
                          collectSubModels=True, numFolds=3)
```

Pretty straightforward, since it is based on the basic concepts of *Estimator* and *Transformer*.

Persistence

One part of the machine learning pipeline process is persisting the output by saving it to a file system. Later on, the output can be deployed to the engineer's staging/production environment, shared with colleagues for collaborative work, or just persisted to disk for the sake of keeping the model for future references. MLlib provides persist to disk functionality using .write().save(model_path) for its *Models*, including its *PipelineModel* and featurization learners models, such as *MinMaxScalerModel*.

```
[source, python]
path = "/cv_model"
cv_model.write().save(path)
```

For loading the MLlib model from disk, you are required to know the model class used for saving it. In our case, CrossValidator produces the CrossValidatorModel and this is what we use to load the model using the load(path) function:

```
[source, python]
from pyspark.ml.tuning import CrossValidatorModel
read_model_from_disk = CrossValidatorModel.load(path)
```

Now, we loaded into memory a model and it is ready for use.

You can also export your model to a portable format like ONNX, and then use the ONNX runtime to run the model. But not all MLlib models support it. We will discuss this and more formats in chapter 8.

Summary

The chapter provided an introduction to MLlib, machine learning algorithms, how to mitigate potential problems, and how to combine the machine learning pipeline building blocks using ML Pipeline functionality for collaborative, structured work. It contained a lot of wisdom and insights into working with MLlib that you might want to revisit as you learn more about machine learning and Spark.

The next chapters are going to show you how to leverage all the work thus far and extend the machine learning capabilities by bridging into other frameworks such as PyTorch and TensorFlow.

---

1 A feature in the Experimental phase is usually still under development and not ready to be implemented in the actual software or product yet. In most cases, these features are limited to just a few modules in the software and are incorporated mainly to gain knowledge and gather comprehensive insights for the future development of the software. Open-source technologies are likely to introduce experiment features that can be leveraged in production code. Be aware that these features might change in future versions of the software, and the contributors are not committing to continue supporting them or building a well polished feature.

Chapter 5. Bridging Spark and Deep Learning Frameworks

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. The GitHub repo is available now at https://github.com/adipolak/ml-with-apache-spark.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jleonard@oreilly.com.

So far, the main focus of the book has been on leveraging Spark's capabilities for machine learning workloads. Despite this, Spark is often a natural choice for scalable analytics workload, and in many organizations, data scientists can take advantage of the existing teams supporting it. Which makes ML infrastructure a shared responsibility between data scientists, data engineers, machine learning engineers, and analytics engineers. Using a scalable, generic tool such as Apache Spark enables this collaborative work.

While Spark is a powerful general-purpose engine with rich capabilities, sometimes, it lacks critical features to fully support scalable deep learning workflows. This is the natural *curse of development frameworks*. In the distributed world, every development framework needs to make decisions in the infrastructure level, which later limits the possibilities of the API, and its performance. Spark limitation is mostly lack of implementations for

deep learning algorithms. Which sometimes makes it hard to data-scientist working with deep learning fully take advantage of.

Specifically, with Natural Language Processing and Image Processing, which rely heavily on deep learning algorithms. These systems can also be trained on large datasets with various technologies, which means that it is likely that to effectively develop deep learning models, you will have to rely on a broader range of algorithms.

Why will this chapter discuss how to bridge to deep learning frameworks rather than completely using other tools? In organizations, when there is a well-supported distributed system for processing and digesting data, it is a best practice to work with what exists first and take full advantage of it. Many times, introducing new distributed frameworks, take months to years. This is, of course, depends on the team size, the workloads, importance of the task to business goals, and long-term investment. Google's article about Hidden Technical Debt in Machine Learning Systems taught us that training machine learning is only one pieces of the puzzles. Take a look at Figure 5-1, where the ML Code box speaks for the training code itself, and the bigger boxes surrounding it are the support it needs from the rest of the system. These require more engineers, frameworks, and overall organizational investment. These challenges of creating the buy-ins of engineering support for distributed clusters, can be done if you first leverage what already exits. In this case, this is Apache Spark which supports data collection, verification, features extraction, analysis tools, etc. The concept of bridging discuss how to build the system to connect Spark capabilities to other frameworks. By doing so, enabling you and your team to take advantages of algorithms that exist in other frameworks while securing the buy-ins from the rest of the organization.

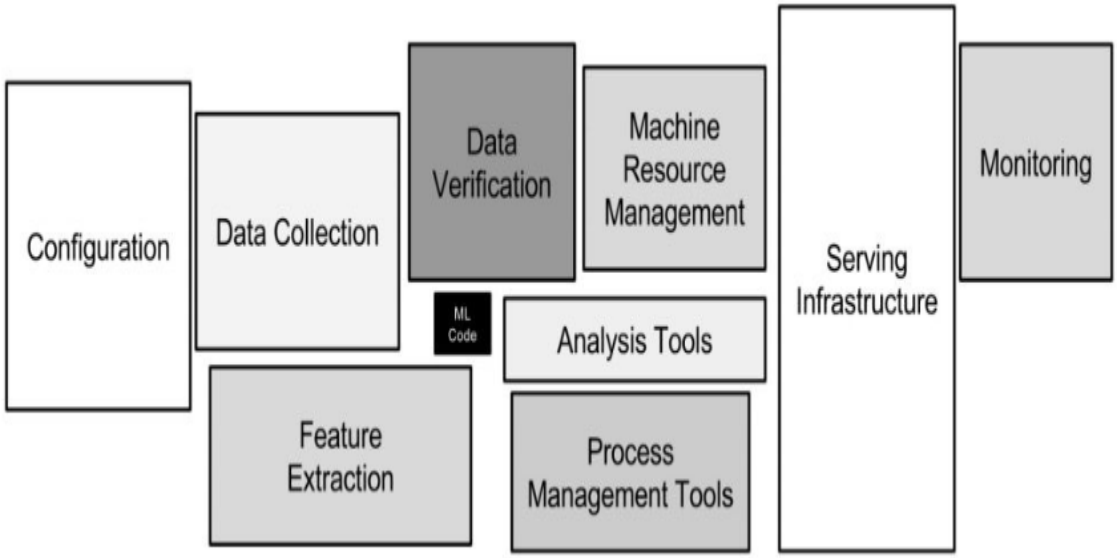


Figure 5-1. Hidden Technical Debt in Machine Learning System by Google, Inc.

Now that there is a better understanding of the why, let's explore the technologies that enables us to execute a distributed training workflow and has a good experience and industry traction that shows they are trusted by reputable organizations. Off the shelf, both TensorFlow and PyTorch provide capabilities to ingest and preprocess large datasets, but they are often more tricky to achieve and require learning dedicated frameworks that are solely ML focused which collide with the knowledge of the team you are working with.

Think about it, even if these teams would agree, the request from peer teams to fully shift their data ingestion, data processing, feature extraction, etc., to a new framework would require a massive change in the infrastructure for the rest of the workloads, which impact your ability to execute on the task at hand. That's why we're tasked with figuring out how to combine what already exists with what we would like to introduce solely for ML training purposes to achieve more. Throughout this chapter you will learn about the two clusters approach, file format challenges and how to bridge multiple technologies, Petastorm, why caching is crucial for ML, project Hydrogen, hardware accelerator aware, and Horovod.

Two Clusters Approach

When there is a need for deep learning algorithms that are not implemented in MLlib, an approach called the two clusters approach can come in handy. In this approach, there is a dedicated cluster for running all Spark workloads, such as data cleaning, preprocessing, processing, and feature engineering. As shown in Figure 5-2, the data is later saved into a distributed file system (such as Hadoop) or an object storage (such as S3 or Azure Blob) and is available for the deep learning cluster to load it and use it for building and testing models.

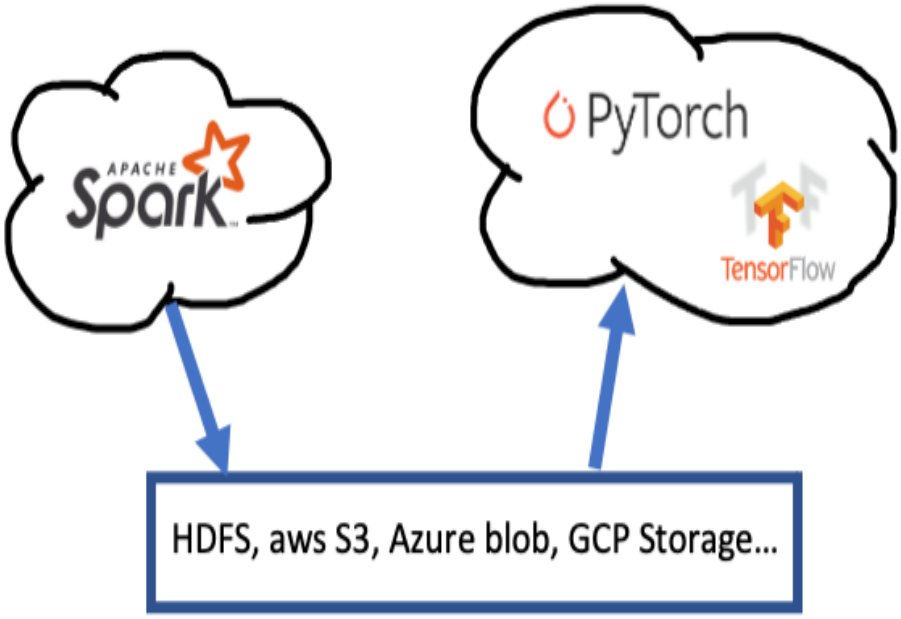


Figure 5-2. The two clusters approach: a dedicated cluster for Spark and a dedicated cluster for PyTorch and/or TensorFlow with a distributed storage layer to save the data to

When ingesting the data into a TensorFlow cluster, there is a dedicated *Data API* where you can create a *dataset* object and tell it where to get the data from. It can read text files (such as CSV files), binary files with fixed-size records, and a dedicated TFRecord format for records of varying size. All are row-oriented formats. Moreover, TFRecords are the default data format for TensorFlow and are optimized using Google's Protocol Buffers.[1] While having an optimized row file format is great, with Spark and Analytics, best practice is to work with a columnar format. On top of that, it might limit us to working with TensorFlow. There is also an open question of working with multiple ML tools. Is TensorFlow always enough? What can happen if as a data scientist I need to implement algorithms from another library? What about PyTorch?

To solve these questions, we need to re-think data formats and adjust them to fit each platform independently. Moreover, we'll need to adjust the data types. Just like you learned in Chapter 4, Spark has dedicated MLlib data types! With both PyTorch and TensorFlow, we might experience type and format mismatch. For example, saving an MLlib sparse vector into parquet and later on directly trying to load it into the PyTorch equivalent data type. While we can absolutely overcome it with arrow pyarrow.parquet [2]and build our own translation layer it will require us to understand how arrow works, define the batches and handle it ourselves. These generic churns can rapidly turn into grueling and error prone tasks.

This is why we should consider introducing an independent translation/data access layer that supports parquet and will simplify the data management[3] aspect by unifying file formats across different types of machine learning frameworks.

Dedicated Data Access Layer

Microsoft introduced the concept of the data access layer (DAL). DAL means having dedicated software that provides simplified access to data stored in some kind of storage. The actual storage can vary and the layer can support multiple storage connectors as well as provide different features

such as data translation, caching etc. Of course, you can use the DAL concept outside of Microsoft environments as well.

DAL is not responsible for the *reliability* of the storage itself, but the *accessibility*, making it accessible for different apps to consume the storage. So, we are able, as users, to consume data that was written with other tools leveraging a higher level of abstraction. This kind of model could potentially retrieve or write records to or from a file system. You don't see the complexity of the underlying data store because the DAL hides it. DAL can help us close the gap of data types, formats etc, for working with multiple ML training platforms.

Our dedicated access layer should be scalable and support distributed systems. Which is to say it should be able to save data to a distributed storage and also leverage existing distributed server's architecture to write and read data in a distributed manner. It should also support the various data types to bridge the gap between Spark and distributed Machine Learning frameworks. It is nice to have a rich software ecosystem, in case there is a new emerging ML framework and there should be some notion of caching the data, since machine learning algorithms take many cycles over the data for improving the model accuracy and reducing loss. We'll discuss more on caching in a moment.

If we filter on specific fields in our ML experiments, it should be able to support columnar, and if we use machine learning algorithms that sample the data during training cycles for example, the LSTM, it should allow for row filtering as well!

To summarize:

Distributed systems

Leverage existing systems to support scale.

Rich software ecosystem

Knowing that the data access layer will evolve with the demand from data scientists for new machine learning frameworks. On top of that, the

tool support will continue, bugs will be fixed, and new features will be developed.

Columnar Support

Columnar file formats store data by column, not by row, which allows us better efficiency when filtering on specific fields during the training and testing process. Sometimes, we would want to create one table which everyone is using, so we can execute various experiments while flirting only with the desired columns for each experiment and not filtering columns for all experiments. Autonomous driving datasets are a good example, where data is coupled with sensors information such as Radar and LiDAR that allow to actively project signals and measure their response. We might not want to load sensors information for every training iteration and columnar support allows us to be more efficient.

Row Filtering

Machine Learning is unique in the sense that some algorithms sample the data during training cycles. For example, LSTM - Long Short Term Memory algorithm is part of the deep learning algorithms family and is used for time series forecasting. With LSTM there is a series of observations and a model is required to learn from the series of past observations to predict the next value in the sequence. Here there might come a requirement to filter the rows in each iteration based on timestep in the time series.

Selecting a Data Access Layer

So, now that we know we need a DAL, how do we pick one?

There is a range of solutions available, from closed source to open source. In this book, we focus solely on open-source solutions. Our chosen tool will support bridging from Spark workloads to enrich the machine learning algorithms capabilities with TensorFlow and PyTorch. There are two open sources that come to mind, Alluxio and Petastorm.

Can Alluxio fit the bill?

Alluxio is an open-source data orchestration layer for data analytics. It has a Unified Data layer for speeding access to the data and makes data available across multiple clouds, on-prem and more.

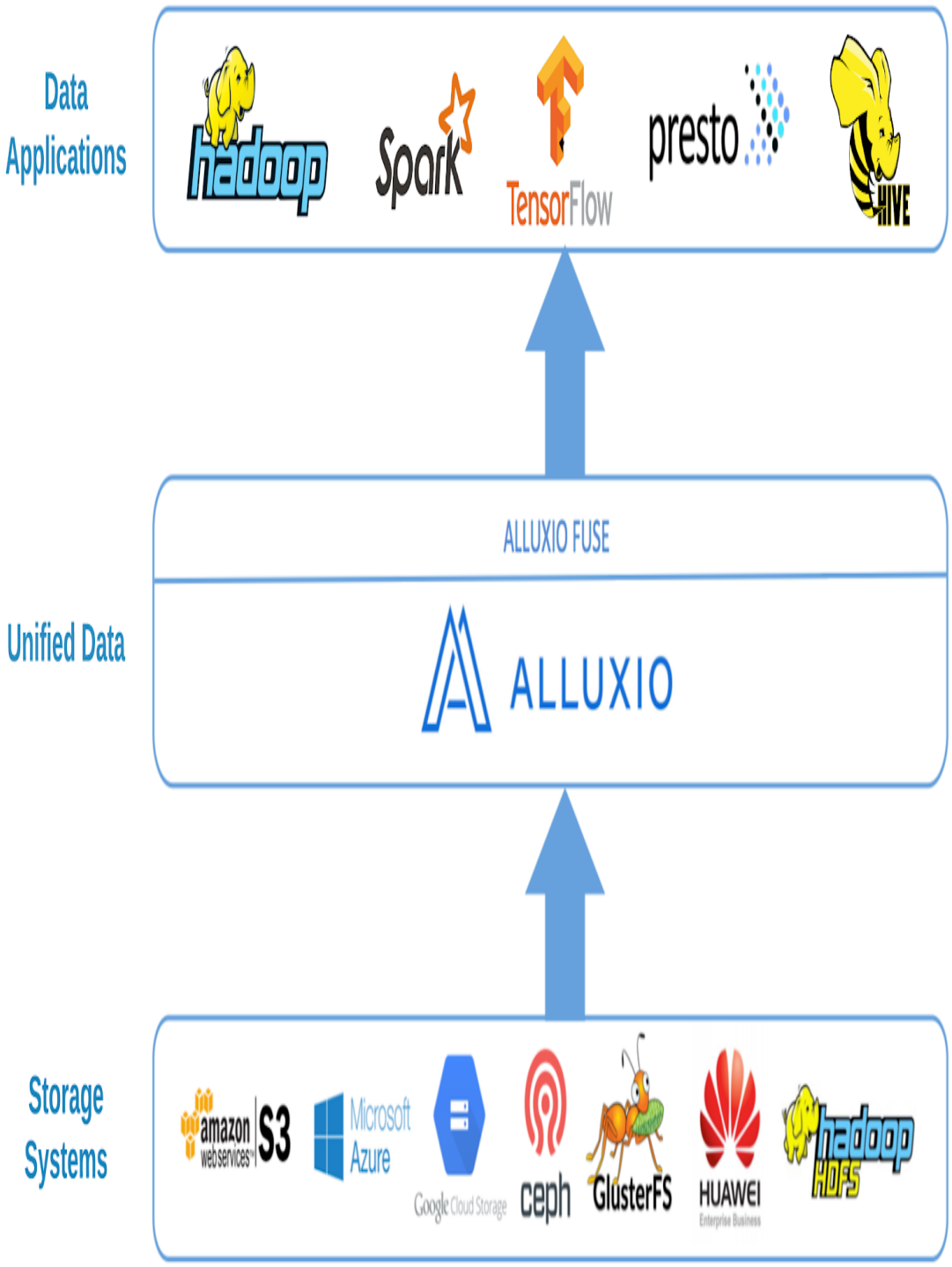

Data Applications
hadoop
Spark
TensorFlow
presto
HIVE
ALLUXIO FUSE
Unified Data
ALLUXIO
Storage Systems
amazon web services S3
Microsoft Azure
Google Cloud Storage
ceph
GlusterFS
HUAWEI
Enterprise Business
hadoop HDFS

Figure 5-3. Alluxio high level architecture with unified data. Figure from Alluxio site

Alluxio looks like an interesting solution that can serve us beyond bridging the gap, yet from looking at their code and docs, as of the time this book is being written, it seems like a black box that is very hard to optimize and manage. This is why I suggest looking into Petastorm. This is also the technology used in the book hands-on tutorial and code samples.

What about cache? Is it critical for training ML?

The caching mechanism is known for improving performance when data is used more than once and in a relatively sequential manner. For example, loading the first 10 rows of specific columns from a dataset and iterating on them multiple times before moving to the next batch of rows. Same goes for ML. In ML the goals of the algorithms are to optimize the loss function to increase accuracy while providing a generic enough model that can be used in the real world data, beyond the test dataset. For that, in many cases, the algorithms are interested in the data multiple times depending on the number of iterations provided, desired accuracy or specific callbacks and epochs mechanism like we discussed in Chapter 3. Storing the data in dedicated, fast or in memory storage has the potential of speeding up the machine learning training process. When looking at cloud storage options, caching is often a criteria we need to define as customers. Not all storage allows for caching out of the box. For example, with AWS S3, we might need to leverage a dedicated caching service. While it is important to understand the concepts, we are not going to cover cloud caching in this book since it often requires a more specific solution.

What Is Petastorm?

PetaStorm is an open-source data access library developed at Uber. This library enables us to train and evaluate deep learning models directly from multi-terabyte datasets in Apache Parquet format. It does it by enabling us to read and write parquet storage with TensorFlow, PyTorch, and other

Python-based ML training frameworks. Several features of Petastorm support the training of deep learning algorithms, including efficient implementations of row filtering, data sharding, shuffling, access to a subset of fields, and time-series data support. Right out of the box, it seems to work. So let’s break it down a little as shown in Figure

7-4.

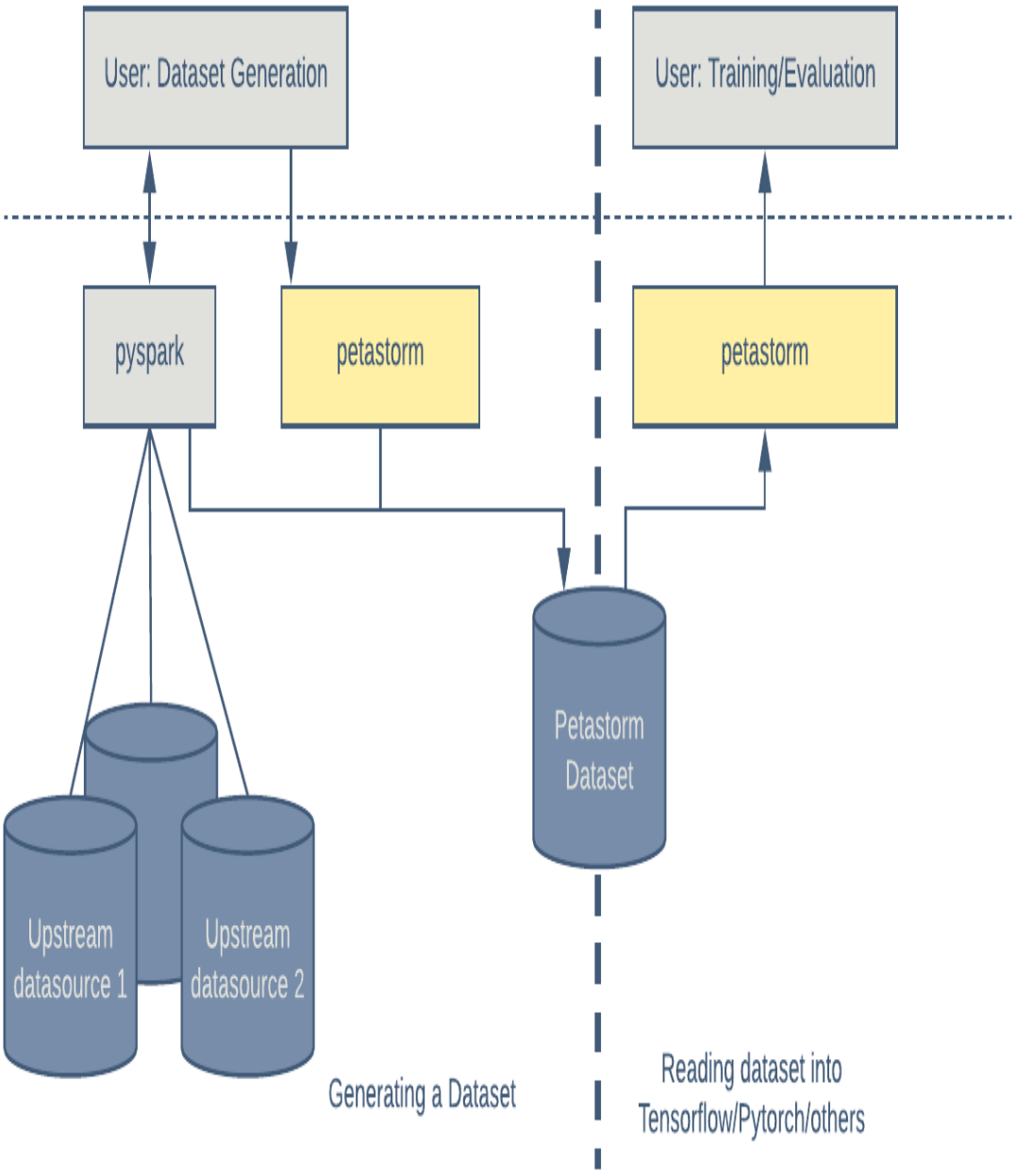


Figure 5-4. Datasets processed with pyspark, combined and used multiple times for model training and evaluation available in parquet columnar format for PyTorch and TensorFlow. Image by Uber

This figure actually tells the story of two alternative integrations. The first one is to leverage petastorm as converter logic alone and keeps the data in

strict parquet format. The second approach is integrating petastorm as a dedicated format with the Apache Parquet store.

Depending on how we use the dataset, both options can be good. Choosing between them depends on how complex our system is, If TensorFlow, PyTorch and Spark are the only frameworks we use, having petastorm as a store might make sense, but if our data system is more complex, it might be best to keep it in a non-petastorm store and leverage petastorm as a converter alone.

Let's examine the first approach of training a model from existing "non-petastorm" parquet stores using Petastrom SparkDatasetConverter.

Petastrom SparkDatasetConverter

SparkDatasetConverter is an API that can do the "boring" work for us, of save, load, and parsing the intermediate files so we can focus on the unique part of the deep learning project.

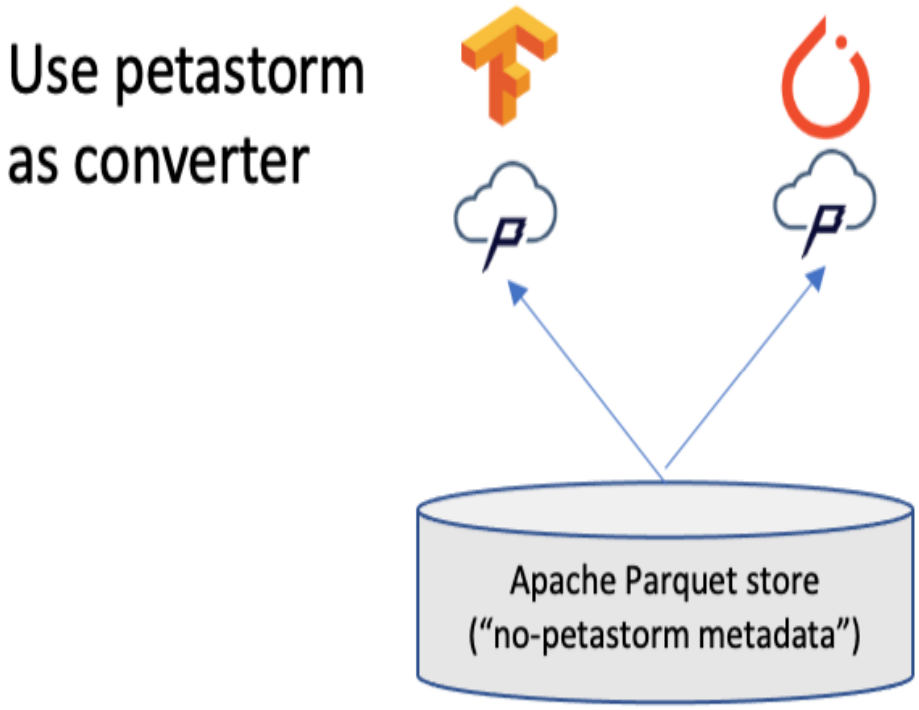


Figure 5-5. First alternative - petastorm as converter

How does it work? It takes the data from a Spark DataFrame, checks if it's already cached and persisted in the distributed file system, and persists it if not. Then the converter will load the persistence file into the TensorFlow dataset or PyTorch data loader. In practice, we first need to define the cache path to provide a directory for the intermediate files:

```
[python,code]
from petastorm.spark import SparkDatasetConverter
# Set a cache directory on DBFS FUSE for intermediate data
spark.conf.set(SparkDatasetConverter.PARENT_CACHE_DIR_URL_CONF,cach
e_path)
```

The instance cache_path is of type String and defines the location for the caching.

Later, SparkDatasetConverter (on its own), is able to recognize if a DataFrame was cached or not by analyzing Spark's DataFrame query plan. At the end the data will be persisted in the path with this directory format: `{datetime}-appid-{spark_application_id}-{uuid4}`. Data time string format is `'%Y%m%d%H%M%S'` which represents the time of dataframe materialziation. `spark_application_id` is the application ID, you might be familiar with this `.sparkContext.applicationId` - where we can pull the application ID directly from the running Spark session, and `{uuid4}` is a random number.

Using this converter has many benefits. It caches intermediate files and also cleans the cache when the program exists. Additionally, it converts Spark's unique MLlib vector into 1D arrays automatically. This approach removes the decoupling from needing to use Spark and specifically MLlib for training machine learning models.

After setting the cache for the converters, it's time to create a converter itself.

The converter takes in:

`parquet_row_group_size_bytes`

> This is a critical component that defines the performance of the converter itself, how fast it operates, and also can prevent out-of-memory executions. It is of type Int and defines the bytes size of a row group in parquet after materialization. The official parquet documentation recommends a row group size of 512 to 1024 MB on HDFS. There will be different block sizes optimized for cloud and on-prem storage. It also depends on what type of data you have. On storage clouds like S3 or Azure Blob, the object sizes themselves are often optimized for a smaller block, with Azure it is between 64KB to 100MB and S3 is about 5MB to 100MB. But these numbers can change and it is always a good practice to check with the cloud providers which sizes they are optimizing for. With images, for instance, you'll want to figure out how much space they take up on disk. Remember that this

parameter is in bytes, so optimizing for HDFS with 512 MB is equal to `512 * 1024 * 1024` bytes. If we ran our example in the cloud, we might want to use 1000000 bytes which are 1 MB, as they work nicely with our images data.

`Compression_codec`

As we discussed in Chapter 3, Parquet has multiple compression codecs, and since Petastorm works with Parquet, it allows us to define it as well, the default is `None`. Don't mistake image data compression codec(jpeg, png), the `Compression_codec` as part of the Spark converter functionality, sperefers to parquet compression.

`dtype`

dtype defines how to assess floating points in our data. In machine learning, when we convert data from one state to another, there is always a risk of losing information. Especially if we convert strings to numbers and later round them up or change their representation again. The converter allows us to be very specific with this definition as well with the default type of `'float32'`.

> **NOTE**
>
> [TIP]Typically, all the above configurations should be in a global variable so it is both easier to remember and easier for a team to collaborate. You can define a separate configuration file for that case.

After understanding the configurations, the code itself is straightforward:

```
[python, souce]
# TIP: Use a low value for parquet_row_group_bytes.
# The detafault of 32 MiB can be too high for our images case if we
run it in the cloud
# converting the train data frame
converter_train = make_spark_converter(df_train,
parquet_row_group_size_bytes=32000000)
```

```
# converting the test data frameconverter_test_val =
make_spark_converter(df_val, parquet_row_group_size_bytes=32000000)
```

At this stage, the dataframe has NOT YET materialized. Petastorm enables us to define additional preprocessing functions using `TransformSpec`. All the transformations we define here are going to be applied to each row on the Spark worker thread/process. We need to pay attention to the columns we want to keep, the types of data, the order of the columns and the final schema.

Let's take a look at a code example of how to define TrabsformSpec:

```
[python, code]
# The output shape of the `TransformSpec` is not automatically
known by petastorm,
# so you need to specify the shape for new columns in `edit_fields`
and specify the order of
# the output columns in `selected_fields`.
transform_spec_fn = TransformSpec(
  func=transform_row,
  edit_fields=[('features', np.uint8 , IMG_SHAPE, False)],
  selected_fields=['features', 'label_index']
)
```

In this code snippet, we defined a func that is going to operate in Spark workers, named transform_fun . This callable function performs the transformation of *pre-transform-schema* to *post-transform-schema*.

In our example, we use this function to prepare the data to be injected into a dedicated TensorFlow MobileNetV2 neural network (more on MobileNetV2 in chapter 8):

```
[python, code]
def preprocess(grayscale_image):
  """
  Preprocess an image file bytes for MobileNetV2 (ImageNet).
  """
  image = Image.open(io.BytesIO(grayscale_image)).resize([224,
224])
  image_array = keras.preprocessing.image.img_to_array(image)
  return preprocess_input(image_array)
def transform_row(pd_batch):
  """
  The input and output of this function are pandas dataframes.
  """
```

```
  pd_batch['features'] = pd_batch['content'].map(lambda x:
preprocess(x))
  pd_batch = pd_batch.drop(labels=['content'], axis=1)  return
pd_batch
```

In our case using python's `Pandas dataframe Map` functionality, `transforw_row` is going over the data in the 'content' column and translates it into processed image data with the size and preprocessing requested by MobileNetV2. It won't always be a Pandas dataframe row. This is only the case for our example. If you remember, in Chapter 4 we used Spark's Pandas APIs to extract features and this is when we introduced the pandas dataframe instance. Additionally, it's a good example of how the machine learning model we use can have an effect – no missing values are allowed, and resizing the images is necessary since their sizes in the original dataset vary and most algorithms would request a uniform size image shape.

If not defined correctly, we might encounter the next error:

```
[error code]
File "/opt/conda/lib/python3.9/site-
packages/petastorm/arrow_reader_worker.py", line 176, in
_check_shape_and_ravel
    raise ValueError('field {name} must be the shape {shape}'
ValueError: field features must be the shape (224, 224, 3)
```

Also, take a look at the pythonic way in which we handle the Pandas Dataframe. In the code snippet, we process them to become *tuples*[4], and not *namedtuples*[5] since this is what TensorFlow expects in our scenarios. As a result, the selected_fileds dictate the order and names.

The above error won't show up until you run the converter and create desired datasets. That means that we need to decide which training framework we want to use and convert to the desired instance. Here is a code snippet that demonstrates how we make the actual TensorFlow data set by calling the make_tf_dataset function and providing it with the transform spec, and batch size:

```
[python, source]
with
converter_train.make_tf_dataset(transform_spec=transform_spec_fn,
```

```
                                          batch_size=BATCH_SIZE) as
train_dataset, \

converter_test_val.make_tf_dataset(transform_spec=transform_spec_fn
,
                                       batch_size=BATCH_SIZE) as
val_dataset:
```

The above code snippet creates *train_dataset* and *val_dataset* in TF dataset format which can be downstream into a Keras or a TensorFlow algorithm.

Petastorm as a Parquet store

The second alternative is to build a petastorm store with parquet as the data format. This is the classic variation of the data access layer. It requires us to introduce petastorm code in all our services. It couples using petastorm with all data consumers as shown in Figure 5-6.

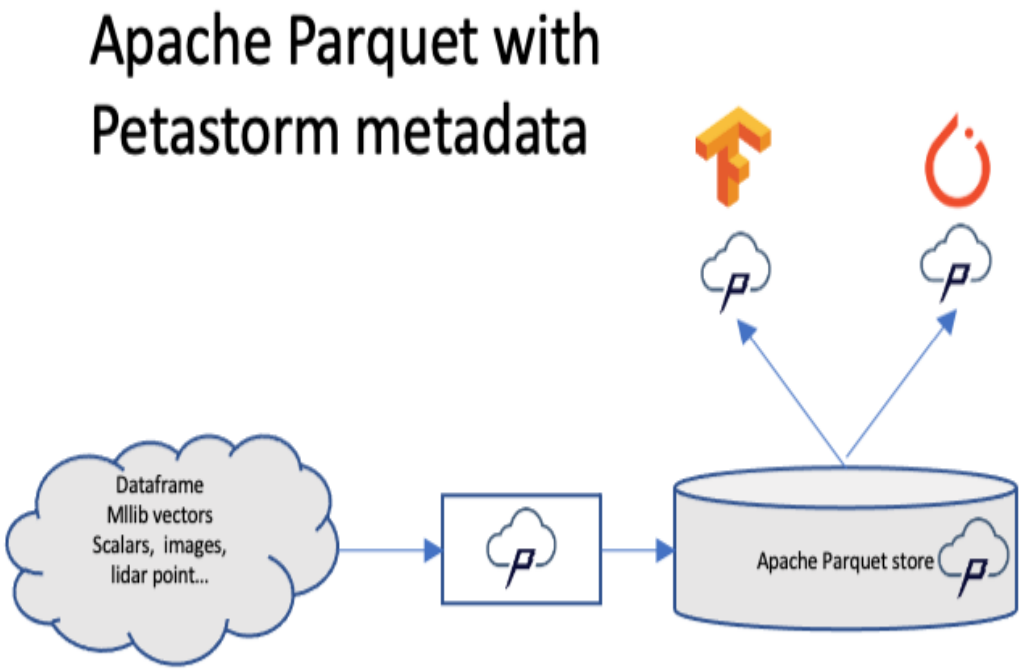


Figure 5-6. A model demonstrating petastorm as a dedicated store with metadata on top of the parquet

Petastorm on its own is not a complete store. It enables tensor support through saving additional metadata of the fields translated into numpy data types. To create one, you can define a new schema by creating a Petastorm Unischema instance with dedicated fields and leverage `dict_to_spark_row` functionality.

```
[python, source]
from petastorm.unischema import Unischema, UnischemaField
imageSchema = Unischema('caltech156schema', [
    UnischemaField('content', np.uint8, (224, 224, 3) ,False),
    UnischemaField('label_index', np.int32, (),
ScalarCodec(LongType()), False)])
```

UnischemaField is a type used to describe a single immutable field in the schema. You must provide it with the name, numpy_dype and shape. Codec definitions for encode/decode during a serialization process Petastorm asses on its own.

Let's take a look at our fields. The content field is of BinaryType Spark SQL types and the label_index field is of LongType Spark SQL types. While the latter is straight forward, the first one isn't. BinaryType in Spark is a Scala array of Scala Byte. When choosing a numpy_dype, we need to go back and assess the origin of the data. Content field is based on images. Their numeric representation range is [0,255]. For that range, `Np.uint8` is a good fit. `uint` stands for unsigned integer type, only positive range of numbers.

After defining the schema, you can leverage `dict_to_spark_row` functionality with Spark rdd to verify that the data confirms with unischema definition types and encodes the data using the codec specified. In our example we provided a `ScalarCodec`. Later write the data into the store with Spark write functionality.

Spark Hydrogen Project

Project Hydrogen is a community-driven project to improve Apache Spark's support for deep learning / neural network-distributed training. Previously, the clusters were separated and we had clusters for processing data with Spark and another dedicated cluster for deep learning. The reason is that the Spark Map-Reduce scheduler approach didn't fit Deep Learning training with a cyclic training process. The algorithm tasks must be coordinated and optimized for communication and support *backpropagation* and *forward propagation*.

Barrier Execution Mode

In a neural network, backpropagation means "backward propagation of errors." In every iteration over a subset of the data, it calculates the gradient

of a loss function with respect to the weights in the network. Later, it propagates the error back into the previous network layer and recalculates that layer. The opposite behavior is happening with forward propagation, sometimes referred to as feedforward. Check out Patterson and Gibson's Deep Learning book to learn more about the mathematics and behavior of deep learning. To solve the scheduler challenge, hydrogen introduced the *Barrier Execution Mode*. It allows us to define a specific code block, where the barrier execution is being used.

In order to work with barrier mode, we need to use barrierRDD with barrier context and Spark's rdd functionality, such as Map Partitions. Barrier RDD signals that all tasks over this rdd are going to run with a Barrier context. Barrier context within the task enables us to decide which operation should be coordinated, hence "*barrier*". First, define the stage logic itself with BarrierTaskConext:

```
[python, source]
from pyspark import BarrierTaskContext
def stage_logic(row):
    context = BarrierTaskContext.get()
    # some logic that needs to be coordinated
    context.barrier()
    return row
```

Second, define the barrier rdd and call on stage_logic with mapPartitions:

```
[python, source]
barrierRdd = df.rdd.barrier()
rdd = barrierRdd.mapPartitions(lambda x: stage_logic(x))
```

That's it. All the functionality we defined in the stage logic until the context.barrier() call is going to take place in the barrier mode. The barrier mode leverages the MPI programming model discussed in Chapter 1 to allow for better communication and a coordinated cyclic training process over the Spark cluster.

Now that we are able to define stages and barriers, let's level up and understand the next challenge Project Hydrogen aims to solve.

Accelerator-Aware Scheduling

Just the way we read it, this functionality allows us to expose the cluster GPU addresses. We already know that for preprocessing the data, executors that run on CPUs are sufficient, whereas, for training, we need GPUs. To pragmatically find the GPUs, we must configure Spark properties with the amount of GPUs and a discoverability script:

Table 5-1. Configuration for gpu schedd

u
l
i
n
g
a
w
a
r
e
n
e
s
s

```
spark.executor.resource.gpu.amount  5
```

```
spark.executor.resource.gpu.discoveryScript  /home/ubuntu/getGpusResources.sh
```

```
spark.driver.resource.gpu.amount  1
```

```
spark.driver.resource.gpu.discoveryScript  /home/ubuntu/getGpusResourcesDriver.sh
```

```
spark.task.resource.gpu.amount  1
```

In addition to the examples above, the discoveryScript functionality can be used to configure NVIDIA Rapids GPUs as a Spark plugin/addon. Rapids

itself is configured to have better performance for operations such as joins, aggregations, etc. While Rapids is an important piece of the puzzle, it's role is actually about optimizing the hardware itself. To learn more, check out the enabling NVIDIA Rapids with Spark Cluster tutorial. Notice that Kubernetes clusters might behave differently from a cluster where the resources are controlled by Yarn or standalone clusters.

With accelerator-aware scheduling, we source the GPU addresses from within a Task that runs in an Executor using TaskContext:

```
[python, source]
Context = TaskContext.get()
Resources = Context.resources()
Gpus = resources['gpu'].addresses
# feed the gpu addresses into a dedicated program
```

From the driver, we can leverage SparkContext with similar logic to the one before:

```
[python, source]
sc = spark.sparkContext
gpus = sc.resources["gpu"].addresses
# feed the gpu addresses into a dedicated program
```

We've seen how we can source the GPUs addresses themselves, now we can feed them into a TensorFlow, or other AI program.

To further optimize resources checkout the pyspark resources documentation.

Introducing Horovod's Estimator API

To tie all of that together in an automated way, we can leverage Horovod. Horovod, similar to Petastorm, is an open-source developed by Uber. It was donated to Linux's LF AI & Data Foundation. Its core goal is to allow for single GPU training to be distributed into multiple GPUs. Due to the great use of Spark in Uber, they also introduced the concept of the Estimator API. The Horovod Estimator hides the complexity of glueing Spark DataFrames to a deep learning training script. This is *yet another* tool that helps us to

read the data into a format interpretable by the training framework, and distribute the training itself using Horovod. As users, we need to provide a TensorFlow/Keras or PyTorch model, and the Estimator does the work of fitting it to the DataFrame. After training the model, the Estimator returns an instance of Spark Transformer representing the trained model. Later, the model transformer can be used like any Spark ML transformer to make predictions on an input DataFrame as we discussed in Chapter 6. Horovod overall helps us configure GPUs and define distributed training. Working with Horovod (even as an exercise) requires dedicated hardware that goes beyond the scope of this book.

Summary

This chapter discussed bridging data from Spark DataFrame formats into TensorFlow datasets and Pytorch dataloader. We covered the two cluster approach, data access layer, petastorm, and more as tools you have in your arsenal to bridge Spark and DL clusters. It's important to remember that this world is still evolving and is going through a massive transformation. The basic concepts and requirements will stay the same, but the technology itself may change. As a result, we learned to assess criteria for a dedicated data management platform. Even though this book won't get into every aspect of data management and hardware environment, it's important to remember that code+data+environment goes hand in hand, as mentioned in Chapter 3.

---

1 Protocol Buffers, also known as Protobuf are a language, platform, neutral extensible mechanism for serializing structured data open source by Google similar to JSON format.

2 Spark heavily leverages arrow, but it is abstracted and we rarely work directly with it.

3 The process of inputting, storing, organizing and maintaining the data that an organization creates and collects is known as data management.

4 Tuples - tuples are 1 of 4 python data types to store collections of data. Tuples store the data in order, unchangeable and allows for duplicate values.

5 Namedtuple - also known as dictionaries with key and value.

About the Author

Adi Polak is a senior software engineer and developer advocate in the Azure Engineering organization at Microsoft. Her work focuses on distributed systems, big data analysis, and machine learning pipelines. In her advocacy work, she brings her vast industry research & engineering experience to bear in educating and helping teams design, architect, and build cost-effective software and infrastructure solutions that emphasize scalability, team expertise, and business goals. Adi is a frequent presenter at world-wide industry conferences and O'Reilly courses instructor. When Adi isn't building Scalable Machine Learning Pipelines or thinking up new software architecture, you can find her hiking and camping in nature.