Machine Learning Across Programming Paradigms

STEPHANIE CHEN, Stanford University

As the long-term importance of machine learning grows and as data processing becomes increasingly dependent on large-scale distributed systems, functional programming languages have become a popular alternative to the standard Python approach to machine learning and data sciences. Here, I implement a small machine learning library, consisting of a preprocessing transformer, a logistic regression classifier, and a pipeline/chaining structure in Python and Scala in order to (subjectively) evaluate the individual experience programming in these two languages in a machine learning context. I find that while Python's language and environment make it extremely easy to get started and quickly implement data transformations, its flexibility and some unintuitive handling of classes can lead to poorly designed code. On the other hand, while Scala is much more difficult to set up (especially outside an IDE) and difficult to start, its type system combined with functional paradigms come together to form a more cohesive overall structure.

1. INTRODUCTION

In recent years, as the amount of data produced, collected, and processed around the world has dramatically increased, the field of machine learning has exploded in popularity and influence in both academia and industry. The effects of this relatively sudden growth can be seen all through everyday life; machine learning systems are now in charge of predicting everything from the optimal route for food delivery drivers to the credit risk posed by loan applicants.

A major enabling factor in these advancements in large-scale data processing has been the development of effective distributed computing systems. As computing power and memory have gotten cheaper while improvements in technology for raw processing power have simultaneously slowed[2], distributed systems have been the key to scaling machine learning and data analysis at all levels.

There has also been a recent upswing in the popularity of functional programming languages, primarily due to this emergence of distributed computing as a standard. Functional programming languages in their relative statelessness are especially suited to parallel processing, compared to the more imperative languages used in machine learning's root fields of statistics and data science. In addition, as we hand over more control of data and decision making to machine learning systems, it's become increasingly important that models are engineered to be scalable and reliable in the long term, and statically typed functional languages seem to provide a built-in level of code stability. Either way, functional languages are likely here to stay in the world of machine learning; this project aims to explore some of the user-level experience of this paradigm shift.

1.1. Machine learning workflows

At its core, machine learning relies on series of transformations applied to data, whether that data is input training data, test data, or data from the internal state maintained by a model. Intuitively, this nature makes machine learning well-suited to both the imperative and functional paradigms — imperative languages allow us to efficiently access and manipulate data in memory, and functional languages allow us to cleanly compose sequences of the transformations applied to that data.

In this project, I focused on a simple, minimal workflow of three steps: feature extraction, model training, and evaluation. The first two steps are briefly discussed below.

1.1.1. Feature extraction. Feature extraction involves the data-processing stages performed on input data to create numerical input features for a model (not to be confused with feature selection, which refers to the techniques used in actually deter-

mining which features should be generated for a given purpose). For example, text corpuses are often transformed into sets of word or n-gram counts, categorical features (ex. "red," "blue") are encoded as some corresponding values, and missing values in datasets are usually filled in with some kind of placeholder. In this project, I implemented a simple scale-normalize feature extractor, in which input values for all given features are normalized to mean of 0 and standard deviation of 1:

$$X_j = \frac{X_j - \text{avg}(X_j)}{\text{std}(X_j)} \tag{1}$$

for column vector X_j corresponding to the data across all input samples for feature j.

1.1.2. Supervised learning. Supervised learning involves training a model on an input dataset of feature values as well as labeled target values representing the actual output of some function f of the input values, with the goal of learning that function f. Two of the most common uses of supervised learning algorithms are classification and regression, where classification involves separating each data sample into one of multiple classes and regression involves finding some parameters that represent the relationship between the input data.

For this project, I implemented logistic regression for binary classification, which involves minimizing the loss (error) on some learned hypothesis function. Determining this function is equivalent to determining some weight vector θ over the input features that minimizes the loss as follows[5]:

$$\underset{\theta}{\text{argmin}}\, J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \log(1 + \exp(-y^{(i)} \theta^T x^{(i)})) \tag{2}$$

for m samples in input x and target y.

I used a stochastic gradient descent approach, which performs the minimization above by "stepping down" the gradient of the loss function and updating θ accordingly for some number of iterations or until convergence.

1.2. Functional programming

As I'm not very familiar with functional programming, a large part of this project focused on learning standard structures and idiomatic expressions in functional programming. In "pure" functional programming, programs are compositions of what are effectively mathematical functions; programs in this style should maintain little to no state and should operate on immutable data abstractions. Languages like Haskell are considered "pure" functional languages, as they restrict the user to rules like the above; in this project, I used Scala, which runs on the Java virtual machine, is object-oriented, and can be written in an "impure" way with ex. statements with side effects.

1.2.1. Category theory. Many of the common structures and paradigms in functional programming can be formalized through category theory, which is a branch of mathematics that I have no familiarity with and thus effectively do not understand at all, though I tried to learn some basic concepts for this project. Category theory is "a general mathematical theory of structures and of systems of structures"[4], which at the level that was practical for me to understand means a theory to formalize how "structures" (data structures, code structures, etc.) and transformations interact in a way that is overall consistent.

In this project, I wanted to learn more about monads, which in category theory is defined as follows[3]:

Definition 1.1 (*Monad*). A monad $T = \langle T, \eta, \mu \rangle$ on a category $\mathbb{C}$ consists of an endofunctor $T : \mathbb{C} \to \mathbb{C}$ on $\mathbb{C}$, a "unit" $\eta : \mathrm{Id}_{\mathbb{C}} \Rightarrow T$, and a "multiplication" $\mu : T^2 \Rightarrow T$.

In practice, a monad is a kind of design pattern, somewhat visible in code structure but more connected to the context of data structures, types, and operations. It's a sort of wrapper for a type that outlines certain behavior regardless of the type contained. In Scala, T above is built into the way a `Monad` type wraps another type ($\mathbb{C}$) without transforming it, η is usually implemented as a function called `pure` or `unit` that wraps $\mathbb{C}$, and μ is called `flatMap` or `bind` and defines the way a monad instance wrapped in itself (ex. a list of lists) resolves back to just a single "flat" instance of that type.

The monad structure was intuitively appealing for this project as it provides a way to chain together a wide variety of operations within some consistent outer scaffolding, which independently of its contents can track state and pass it down — much like what's needed for a model training workflow.

1.3. Status quo of programming languages in machine learning

The most popular language by far in machine learning and related fields is Python, mainly due to its extensive libraries as well as its ease of use in quickly prototyping and testing models. Libraries such as `numpy` for math and matrix operations, `pandas` for data storage and processing, and `scikit-learn` for machine learning (not to mention all the competing deep learning libraries currently in use) provide essentially all the functionality needed to quickly and effectively build and develop models, and as a scripting language, Python is incredibly easy to set up and starting using.

Other generally popular languages include R (familiar to nearly all users coming from a statistics background) and C++ (for speed). Scala is rising in popularity, probably because of the immense popularity of the Spark data processing framework, which is written in Scala and is now one of the most, if not the most, popular "big data" frameworks in use today.

2. METHOD

2.1. Python

The Python implementation was primarily intended to be a benchmark for both programming experience and model performance; it is modeled after the standard `scikit-learn` abstractions of estimator and transformer classes, though I did not reference any `scikit-learn` source code. `numpy` was the only external library used — it would have been interesting to also implement a standard-library-only version for timing benchmarks, but given `numpy`'s ubiquity, I chose to include it for the sake of realism and efficiency. All algorithms (scaling, logistic regression, stochastic gradient descent) were implemented manually, however.

The full implementation consists of a class each for scaling and logistic regression, as well as a pipeline abstraction for chaining transformations, built on three abstract base classes.

```
class LibScaler(baselib.LibTransformer)
class LibLogisticRegression(baselib.LibClassifier, baselib.LibTransformer)
class LibPipeline(baselib.LibTransformer, baselib.LibClassifier)
```

2.2. Scala

The Scala implementation was centered around a `ModelComponent` monad structure; the aim was to compose a workflow of functions (as opposed to the class instances used in the Python implementation). I used the `breeze` and `cats` external libraries for

math/matrix operations and category theory types, respectively — though I didn't need an external `Monad` type, it helped during development to have the type rules enforced by another library. All algorithms were again implemented manually.

The scaler and logistic regression classifiers were implemented as functions with the type signature `DenseMatrix[Double] -> ModelComponent[DenseMatrix[Double]]`. The central `ModelComponent` monad was implemented as follows below referencing [6]:

```
case class ModelComponent[A](data: A, state: Map[String,
    DenseVector[Double]])

trait ModelComponentInstances { self =>
  type MC[A] = ModelComponent[A]

  def flatten[B](xs: MC[MC[B]]): MC[B] = {
    ModelComponent(xs.data.data, xs.state ++ xs.data.state) }

  implicit val modelComponentInstance: Monad[MC] = new Monad[MC] {
    def pure[A](data: A): MC[A] = ModelComponent(data, Map.empty)

    override def map[A, B](fa: MC[A])(f: A => B): MC[B] = {
      ModelComponent(f(fa.data), fa.state) }

      def flatMap[A, B](fa: MC[A])(f: A => MC[B]): MC[B] = {
          self.flatten(map(fa)(f)) }

      def tailRecM[A, B](a: A)(f: A => MC[Either[A, B]]): MC[B] = {
        f(a) match {
          case ModelComponent(Left(nextA), _) => tailRecM(nextA)(f)
          case ModelComponent(Right(b), bstate) =>
              ModelComponent(b, bstate) } }
    } }

case object ModelComponent extends ModelComponentInstances
```

3. RESULTS AND DISCUSSION

3.1. Performance

I used two binary classification datasets — the breast cancer set from `scikit-learn` and a much larger credit card fraud set from Kaggle — and evaluated performance of the Python and Scala implementations of a scaler-logistic regression pipeline/workflow against the `scikit-learn` version on a 25%-75% test-train split. The `scikit-learn` `LogisticRegression` was initialized to match my implementation (ex. L2 regularization, stochastic gradient descent) in terms of parameters.

Both implementations were effectively as accurate as the benchmark, and for some reason also non-negligibly faster than the stochastic gradient descent `scikit-learn` implementation that should have matched all parameters, including max iterations. Given that the standard `scikit-learn` implementation is still much faster as expected, it's possible that parameters that didn't have options to set in `scikit-learn` (ex. step size) could account for the difference.

Table I. Performance

Model	Accuracy (cancer)	Time (cancer)	Accuracy (credit card)	Time (credit card)
`sklearn`	0.9091	0.0265	0.9989	19.16
`sklearn` (standard params)	0.9371	0.0198	0.9991	5.942
Python	0.9161	0.0487	0.9983	12.92
Scala	0.9371	0.1029	0.9791	14.29

3.2. Experience

Here, I explain some of the language and environment features that stood out to me in each language during the development experience.

3.2.1. Setup and environment. For the Python implementation, setting up a virtual environment and installing packages via `pip` went smoothly as usual; I've always liked the `pip` package manager and have rarely run into issues with ex. conflicting environments. I then, however, spent an inordinate amount of time trying to set up a working module system, having only previously worked off scripts or in existing codebases; Python makes it so easy to import files in the same directory that setting up a multi-directory module system was surprisingly unintuitive. The last environment feature that stood out was the `pdb` debugger; I've generally taken it for granted after using `gdb` in classes, but it stood out here when compared to the difficulty of debugging runtime errors in Scala.

Scala environment setup was difficult — I chose to use `sbt` on the command line, as it's what I had used in my limited experience, but setting up a standard project and all its build files and subdirectories felt like it would have been much easier with an IDE. The abundance of build-related files (`build.properties`, `build.sbt`, `Dependencies.scala`) that come with a new `sbt` project were confusing. `sbt` is also absurdly slow compared to ex. a C compiler, but I do like the feeling of being able to compile to catch errors before testing (because runtime debugging in Scala is near impossible).

3.2.2. Design. In the Python implementation, I worked for the first time with the `abc` abstract base class module in the standard library after realizing that standard Python classes don't enforce inheritance rules on instantiation (ex. child classes are allowed to instantiate without implementing any parent class methods) — this was unintuitive. There were some more interesting questions when designing the library base classes: `scikit-learn` implements transformer and classifier features as mixins (ex. transformers are instances of `BaseEstimator with TransformerMixin`, but I chose to use regular multiple inheritance instead to simplify subclass definition (ex. my transformer was just a `LibTransformer` instance).

Figuring out how to work with monads in Scala took far more time than expected — it's hard to understand an extremely abstract concept without anything concrete in front of you. I think I had the general intuition of composability, chaining transformations, etc. in mind early on, but it took a while to understand how all that fit into actual types in code. The abundance of similar-sounding constructs (classes, abstract classes, traits, case classes, objects, case objects) didn't help. I did find, however, that after I had a correctly compiled base structure, it was much easier to add edits and extensions, and on the whole, my Scala code felt like it formed a more cohesive structure than my Python implementation. I think because Scala (and other strongly typed languages) depends so much on the relationships between types, it's harder to start working from scratch, but projects are also less liable to fall apart into spaghetti code. On top of that, being able to have things run correctly on the first try (for the most part) once compiled was highly enjoyable.

REFERENCES

Hermann, J., Del Balso, M. 2017. Meet Michelangelo: Ubers Machine Learning Platform. https://eng.uber.com/michelangelo/.

Simonite, T. 2016. Moores Law Is Dead. Now What? https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/.

Turi, D. Category Theory Lecture Notes. http://www.dcs.ed.ac.uk/home/dt/CT/categories.pdf.

Stanford Encyclopedia of Philosophy. 2014. Category Theory. https://plato.stanford.edu/entries/category-theory/.

Duchi, J. 2016. CS229 Supplemental Lecture Notes. http://cs229.stanford.edu/extra-notes/loss-functions.pdf.

Yokota, E. 2014. Herding cats. http://eed3si9n.com/herding-cats/making-monads.html